FOSDEM 2011

arduino/AVR: interactive development with **amforth**

Date: 2011-01-22
Author: Erich Wälde
File: Fosdem2011-talk-amforth.tex

Contents

1 Introduction

Are you tired of the *edit, assemble, flash, reset and watch* cycle when writing code for your AVR microcontroller? There is an alternative: **amforth** [1]. Yes, it needs to be assembled and flashed, too. But after that you hook the controller to a serial port and `minicom` or similar and you are greeted with the **ok**–prompt. After that you are free to change pin directions, pin levels, or to configure the available periphery in any way you want. You can try out little bits and stick them together, defining new words (functions) as you go. The new words can be used immediately, no need to reassemble and reflash. You have every bit of the controller under direct control. **amforth** is designed to run on the Atmel **atmega**–family of controllers.

amforth is an incarnation of Forth [4]. It is a stack based system, it is virtually free of syntax, and it might change your opinions on microcontroller programming a whee bit. **amforth** is GPLv2 software.

This paper replaces any slides for the presentation at FOSDEM 2011. The presentation demonstrated the ”interactive” trait of **amforth** as well as some concepts special to Forth–Land.

2 The Toolchain

As for working with any microcontroller, there is a fairly long list of items that need to work. If you are used to working with microcontrollers you may skip the first part of this list.

Part 1

Controller
Obviously you need a controller, a supporting board or bread board, some supporting devices like a crystal, a converter for the serial port connection, a connection for the programmer, power supply etc. Choose your favourite board from a surprising variety, including the arduino boards like the duemilanove [13]. My favourite board happens to be the Mega-32 Protoboard by Embedit [14].

Editor
Choose your favourite editor to work on assembly and forth files. Emacs can load the forth-mode that comes with gforth, providing syntax highlighting and indentation.

Assembler
You need software to convert the programm you or someone else wrote, into a format that is understood by the controller. The controller comes with a specific assembly language created by the manufacturer. Atmel provides AVR Studio, a windows based and closed source development tool including the assembler and a lengthy list of include files. The assembler will run nicely with wine [22] on Linux.

avra [18] is an open source implementation of the assembler (not the IDE). It can work with amforth (see section 6.1).

Programmer
To flash the **amforth**–system (or any other program) to the controller, you need some

connection from your PC to the ISP–Port of the controller. There is a range of programmers available ranging from the do-it-yourself sp12 dongle [15] all the way to the usbasp [16] or the AVR MKII programmer from Atmel. There are more.

Flash Tool
The programmer alone is a dead piece of hardware. It needs a piece of software to send the assembled program to the controller. **avrdude** [19] does a very nice job, but there are probably others.

Datasheet
Download the datasheet for you specific controller from the manufacturers site. Sure, datasheets are long and hard to read initially. Some day you will be suprised about the details written down in them.

Part 2

The items so far are neccessary for any microcontroller. Using **amforth** needs still more things:

Amforth Source Code
amforth is available as tarballs or from its subversion repository at SourceForge [1]. **amforth** comes with useful documentation, it's website is also worth inspecting.

However, you need Atmel's include files as well, which have to be obtained from their website [10]. Unfortunately, you need to feed the "Register" web form with something that convinces the website to *let you in*. Then download AVR Studio 4, which contains said include files. The file is a self extracting archive which does cooperate with wine. Install it into a convenient location. The directory you need is called `AvrAssembler2`.

am/forth documentation
amforth hosts a mailing list [2] with archives.

Most notably there is a reference card [3] available listing all commands with stack effect and a one line description.

If you have never heard of Forth before, there are two books by Leo Brodie available online: *Starting Forth* [6] and *Thinking Forth* [7]. The first is still worth a read, despite its age.

`gforth` comes with a detailed tutorial, most of which applies for any Forth.

serial connection
To make this fun truely interactive, you need a cable and some sort of serial port (RS232). USB–serial–dongles tend to work fine.

upload tool
Once your **amforth** source code exceeds a few lines, you want to edit them into a file and upload that file somehow to the the controller. For uploading you can use the python tool `amforth-upload.py` or some utility built into the terminal programm. But note, that timing is important. The upload tool should wait for the echo character to appear and should wait for the prompt to appear after sending a newline. ok and error should be diagnosed and handled. I have had good success with am4up, a tool for minicom written by Al Williams [20], see section 6.4.

With all of the above we are set to go. Next you need to configure a working **amforth**, that will be flashed to the controller. The following list provides the absolute minimum

of information required:

- exact controller model, e.g. `atmega32`
- exact crystal frequency in Hz, e.g. `11059200`
- exact fuse settings. These are neccessary to tell the microcontroller to actually use the external crystal as its clock source — among other things. A brand new controller will run on its internal RC–oscillator at 1 MHz, no matter what you configured in **amforth**. Inline with this example, I use `lfuse=0xee` and `hfuse=0x89`.
- exact baud rate, e.g. `1105200`
- exact settings of your programming dongle, e.g. using an AVR MKII dongle, the flags for avrdude are `-c avrispmkII -P usb`. There are more examples in the makefile.

For a more detailed walk through see section 6.2.

3 Contact!

For now, we assume to have flashed a working **amforth**–system to the controller. We have also connected the controller board to a serial interface, started a terminal programm such as `minicom` and configured the port settings (115200 baud, 8N1, no flow control). If we have done everything correctly, and as soon as the controller comes out of its reset state, we are greeted at this point with the following lines (press the `Return`–key a few times):

serial terminal

```
1 amforth 4.2 ATmega32
2 >
3  ok
4 >
```

Line 1 shows the output of the word (command, function) `ver` , lines 2 and 4 show the input prompt, and line 3 shows the result of our typing the `Return`–key, namely `ok`. We are talking directly to the controller, allthough we have so far not written a single line of code. Well, someone else has, obviously. We have entered interactive Forth–Land at this point.

4 amforth Concepts

4.1 Words

In Forth all *functions* or *commands* or *subroutines* are simply called *words*. To list the available words, we ask **amforth** to list them for us, with the word `words` unsurprisingly:

serial terminal

```
1 > words
2 i@ (i!) i! e@ e! not s>d up! up@ >< cmove> unloop i sp! sp@
3 rp! rp@ +! rshift lshift 1- 1+ xor or and 2* 2/ invert um*
4 um/mod m* + - log2 d< d> 0> u> u< true 0 0< > < 0= = <> r@
```

```
>r r> rot drop over swap ?dup dup c@ c! ! @ execute exit
-int +int -jtag -wdt wdr spirw 1ms set-order set-current
fill u0.r 0<> show-wordlist +usart baud tx? tx rx? rx order
get-order get-current environment? environment end-code code
abort abort" [char] immediate recurse user constant variable
[ ] ; :noname : does> create ?do leave +loop loop do again
until repeat while begin then else if literal int!
applturnkey is Rdefer Edefer words s" ." .s u. dinvert d- d+
d2* init-user ee>ram ee-user tib d2/ cmove dnegate dabs d>s
j * defer@ defer! icompare find search-wordlist to value
unused noop ver ?stack interpret depth rp0 sp sp0 cold pause
quit place word /string source cscan parse 2swap >number
number char refill accept cskip throw catch handler ' type
count spaces space cr icount itype s, digit? ud/mod ud.r ud.
. d. .r d.r sign #> #s # <# hold hld within max min abs mod
/ negate u/mod */ /mod */mod turnkey bl hex decimal bin ['] 
, compile ( \ allot here edp dp /key key? key emit? emit pad
#tib >in cell+ cells base state f_cpu ok
>
```

More surprisingly this list is rather long, and many words look quite odd, like `i@` or `(i!)`. It should be noted, that in Forth words are delimited by white space and can contain any character in their name, e.g. the parentheses in `(i!)` are part of the name.

All availabe words are organised in a simply linked list. Whatever we type in will be looked up in the list and executed if found. We get an error indication, when the word was not found:

serial terminal

```
> verrr
 ?? -13 6
>
```

Can we define a new word? Yes we can:

serial terminal

```
> : hi  ." Hello, FOSDEM!" cr ;
 ok
>
```

The colon (`:`) starts the compiler — yes the compiler is onboard! Colon creates a new entry in the wordlist, it will consume the next token on the input as the new name (`hi`). Then the commands that come after the name are looked up in the wordlist. If found, the address of their executable part is compiled into the new word. This works for the word `cr`, which will simply output a newline. The word dot-quote (`."`) will consume the input stream up to the next double quote, store the characters in flash and when run, output them again. The semicolon (`;`) signals the end of our definition and adds some code to return to the calling word. So, if we call the new word, this is what happens:

serial terminal

```
> hi
Hello, FOSDEM!
 ok
>
```

We just used the new word. No re–assembling, no re–flashing, no reset. Just use the newly defined word. Isn't that cool?

Admittedly, this is not a classic `hello world` program for a microcontroller, because there is no blinky LED stuff. So let's get that done. Assume we have connected LEDs to +5V at the anodes and to the pins of port B on the cathode side.

serial terminal

```
1 > $38 constant PORTB
2  ok
3 > $37 constant DDRB
4  ok
5 > $ff DDRB c!       \ make PortB output
6  ok
7 > $f1 PORTB c!      \ make it show 1111 0001
8  ok
```

Lines 1 and 3 store the unobvious addresses of registers PORTB and DDRB (data direction register B) into constants with respective names. In order to switch the pins of port B to output, we need to write 1s into the data direction register. The word c-store (`c!`) will take one byte (a char) and store it into the address given before (`DDRB`). Lines 5 and 7 will also show that backslash (with a following space) will start a comment, which continues until the end of the line.

At this point all LEDs will light up, because the reset state of the port pins is low (zero). So in line 7 we set some pins to high (one), and the corresponding LEDs should go dark.

Almost unnoticed three important concepts have crept in now, which demand further explanation: the data stack, postfix notation, and the base of numbers. And by the way: **amforth** is case sensitive!

4.2 The Data Stack

Numbers are stored on the data stack. The meaning of these numbers is not known to amforth, there are no types. The stack takes entries of two byte width, which are called *cells*. Words, which deal with numbers, pop their arguments from the stack, work with them and push the results back there. For example, we put two numbers on the stack, multiply them and show the result. Between the steps, we inspect the content of the stack with the word dot-s (`.s`). Dot-s lists the content of the stack without touching it:

serial terminal

```
1  > 4 7 .s
2  0 2059 7
3  1 2061 4
4   ok
5  > * .s
6  0 2061 28
7   ok
8  > .
9  28  ok
10 > .s
```

```
 ok
>
```

Multiply (*) will do its work, but it will not show the result. If we want to output the result, we need to call dot (.), which will take the topmost entry from the stack and print it as a formatted string assuming signed integer value.

In order to deal with the stack there is a group of words to manipulate the content of the stack, like dup (duplicate the topmost entry), drop (remove the topmost entry), swap exchange places of the two top entries, and many more.

serial terminal

```
> 1 2 3 .s
0 2057 3
1 2059 2
2 2061 1
 ok
> swap dup .s
0 2055 2
1 2057 2
2 2059 3
3 2061 1
 ok
>
```

The data stack completely eliminates any argument lists found in other languages. The handling of the stack is so important, that words are documented with their effect on the stack like so

file

```
: plus  ( n1 n2 -- n ) + ;
```

The parenthesis are surrounded by white space. They indicate a comment stretching to the closing paren, not to the end of the line. The *stack comment* says, that plus will consume two entries from the stack, and place one entry on the stack as a result.

There is a second stack, called the *return stack*. It is there to organize, what's happening after a word has completed execution.

4.3 Postfix Notation

Tightly connected to the data stack is the use of postfix notation. For multiply we have seen the notation above, and it doesn't seem too strange. However, postfix notation is used for control structures as well:

file

```
: odd          ( n -- t/f ) $0001 and  if   -1  else  0  then ;
: countup      ( n -- )     0 do  i . loop ;
: wait-for-key ( -- )       begin  noop  key? until key drop ;
```

The traditional value for true is -1, the one for false is 0.

4.4 Base

Anything dealing with numbers is using a numerical base, which can be changed on the fly. There are three words to set base to known values: `hex`, `decimal`, and `bin`. To inspect the content of base in decimal output, the word `.base` is useful. But base can be set to anything. Closely related are the numerical prefixes `$`, `&`, and `%`, which denote the number as being written in base 16, 10, and 2 respectively.

serial terminal

```
> : .base ( -- )   base @ dup decimal . base ! ;
 ok
> $ff hex .
FF  ok
> $ff decimal .
255  ok
> $ff bin .
11111111  ok
> &3 base ! .base
3  ok
> &26 .
222  ok
>
```

The numerical base is stored in a variable named `base`, it can be read and written with the words fetch (`@`) and store (`!`).

5 Worked Example: Talking to I2C-Device

I will present a more complex worked example to show, how **amforth** can be used. I want to talk to a device connected via the I2C–interface (aka TWI, two-wire-interface). For purely didactic reasons I choose to ignore the builtin twi interface.

In order to talk to the connected PCF8574 device, an 8-bit IO controller with LEDs connected, the following things need to be done:

- connect the two signals `SDA` and `SCL` to freely chosen pins
- configure the pins as output, set them high
- generate a clock pulse
- get the next data bit and clock it out
- clock out a Byte, MSB first
- generate start and stop conditions
- read ack back from bus
- scan the bus for devices
- send a byte to the device

5.1 Preparing a Comfortable System

To make programming simpler I included a few more words (assembly files) into the **amforth**–system via the file `dict_appl.incl`:

- words/notequalzero.asm
- words/uzerodotr.asm
- words/fill.asm
- words/set-current.asm
- words/set-order.asm
- words/no-jtag.asm
- words/1ms.asm
- words/spirw.asm
- words/wdr.asm
- words/no-wdt.asm

I also uploaded these files (amforth code) to the **amforth**–system on the controller, all of which come with **amforth**. To upload, a tool like `amforth-upload.py` or `am4up` is needed. The file is sent via the serial interface just like you would type it in yourself.

- lib/misc.frt
- lib/bitnames.frt
- lib/ans94/marker.frt
- core/devices/atmega32/atmega32.frt
- lib/ans94/postpone.frt
- lib/dumper.frt

The exact list of words to include into your system mainly depends on the intented use of the controller. Not everything listed above is strictly needed.

5.2 Marker

At some point, you want to clean the added words off the controller and start over. Obviously, we can reflash the controller. But there is a more clever way. The word `marker` will consume the next token in the input stream an create a word of that name. When called, this word will reset the book keeping in such a way, that the entries compiled after and including the marker, are gone.

main.fs

```
1 marker --start--
```

5.3 Configuring IO Pins

Next we configure the used IO pins and give them meaningful names. There are three pins connected to the remains of a fnordlight [12] to illuminate the audience. The pins corresponding to the colors blue, green, and red are connected to pins 0, 1, and 2 on port B.

For the i2c bit bang connection I chose pins 3 and 4 on port B for the bus signals SDA and SCL respectively.

main.fs

```
1 PORTB 0 portpin: blue   \ sda
2 PORTB 1 portpin: green
3 PORTB 2 portpin: red    \ scl
4
```

```
5 PORTB 3 portpin: sda
6 PORTB 4 portpin: scl
```

`portpin:` is a defining word. Calling `addr pos portpin: nameX` will create a new word `nameX`. Calling `nameX` in turn will execute code that was given to it by `portpin:`. `nameX` will push the address of the IO register and the bit mask `(1<<pos)` to the data stack. Words like `pin_output`, `high`, or `low` will consume exactly this information and clear or set the corresponding bits in the port and data direction registers. The pins need to be initialized:

main.fs

```
1   red   pin_output red   high
2   green pin_output green high
3   blue  pin_output blue  high
4   sda   pin_output sda   high
5   scl   pin_output scl   high
```

This sequence will be stored into a word `init` below.

5.4 Setting levels

Once the pins have been configured, we want to see them in action. So we define a set of words to set the bus lines high or low. Setting the bus lines alone is somewhat boring, we want to *see* them in action somehow. So I decided to add switching on and off the blue leds for the SDA signal and the red ones for the SCL signal. Also note, that setting a color pin to high will actually switch off the leds, so the logic is reversed.

main.fs

```
1 : sda0   sda low    blue high ;
2 : sda1   sda high   blue low  ;
3 : scl0   scl low    red  high ;
4 : scl1   scl high   red  low  ;
```

5.5 Waiting . . . more or less

Well, like in (almost?) every microcontroller programm waiting is achieved by wasting cycles. We need to wait a little when creating a clock pulse. So we could just insert a `noop` (this is a Forth word, not the assembly instruction) at these places. But a noop is still pretty short and hard to see with our eyes. Wouldn't it be great, if we could make a word `wait`, which can wait for various amounts of time somehow?

We can. And it illustrates something like *function pointers*, which are called *deferred words* in Forth. To illustrate the effect, we define two words `wait-long` and `wait-short`, which will wait for 500 milli-seconds and 50 milli-seconds, respectively. These are just arbitrary choices.

main.fs

```
1 : wait-long  &500 0 do 1ms loop ;
2 : wait-short  &50 0 do 1ms loop ;
3 \ "function pointer" wait
4 Rdefer wait
5 : slow  ['] wait-long is wait ;
```

```
: fast  ['] wait-short is wait ;
slow
```

The *function pointer* `wait` is defined as `Rdefer`. It stores the address of the code to execute in RAM (as opposed to EEPROM or flash memory). Calling `wait` will retrieve this address and execute it. So we define two more words to store the address of the executable portion of `wait-long` and `wait-short` in wait.

In order to test correct behaviour, we create a word to generate a number of pulses. The number is taken from the data stack, the pulse is generated with the help of `wait`:

main.fs

```
: pulses  ( n -- )
  0 ?do
    red low wait red high wait
  loop
;
\ test "function pointer"
: test-wait
  slow  &5 pulses
  fast &25 pulses
;
```

5.6 I2C Bit Bang

Clocking out a byte on the i2c bus requires a few small pieces of code. We need to generate a clock pulse similar to the inner part or `pulses` above. Just before the pulse, we assert the data signal according to a given bit.

main.fs

```
\ clock a given data bit out
: bit>i2c ( bit -- )
  if sda1 else sda0 then \ set data
  scl1 wait scl0 wait    \ clock it out
;
```

To transfer a byte, we need to extract the bit at a given position. It is sufficient to know, whether the bit was set or not. It is not neccessary to shift the bit to the least significant position.

main.fs

```
\ see if bit at pos is set
: get.bit ( byte pos -- bit )
  1 swap lshift    \ -- byte bitmask
  and              \ -- bit
;
```

Now we can wrap the above words into a loop. Please note that i2c transfers the MSB bit first, so the position is `7 i -` rather than `i`:

main.fs

```
\ clock one byte out, MSB first!
: byte>i2c ( byte -- )
  8 0 do
```

```
    dup 7 i -  \ 7 6 5 ... 0: MSB first!
    get.bit
    bit>i2c
  loop
  drop
;
```

These words can be tested using `slow` without an oscilloscope or other more complex means.

5.7 I2C Start and Stop Condition, Reading ACK

We need three more things to enable writing to an i2c–device: sending a start condition or a stop condition and reading the ACK on the bus. A start condition is produced by setting SDA low while SCL is high and then setting SCL low after a wait. Similarly, a stop condition sets SCL high while SDA is still low, and SDA high later. Please consult the data sheets for exact timing information.

main.fs

```
\ create start, stop conditions
: i2c.start  sda0 wait scl0 wait ;
: i2c.stop   scl1 wait sda1 wait ;
```

The addressed device on the i2c bus signals its presence by asserting SDA low during a nineth clock cycle. We need to switch the SDA pin to input, generate a clock cycle and sample SDA before setting SCL back to low.

main.fs

```
\ read ack|nack from bus
: ack<i2c ( -- bit )
  sda pin_input
  scl1 wait
  sda pin_low?
  sda pin_output
  scl0 wait
;
```

5.8 I2C Putting it together

We can now send a byte to a known address by putting all of the above together:

main.fs

```
\ make it really fast!
' noop is wait

\ send a byte to pcf8574
: >8io ( x -- )
  $40  \ addr
  i2c.start
  byte>i2c ack<i2c drop
  byte>i2c ack<i2c drop
```

```
  i2c.stop
;
```

This can be tested interactively, of course. And we do now have everything needed to scan the bus for connected devices. A loop sends all even numbers from `0` to `$FE` and collects the ACK. If ACK was read (as opposed to high level called NACK), the address is listed:

main.fs

```
: i2c.scan
  $FF 0 do
    i2c.start
    i byte>i2c
    ack<i2c  \ -- ack|nack
    i2c.stop
    if
      i . cr
    then
  2 +loop

;
```

I happen to have connected two devices on the i2c bus of my demo board. So calling `hex i2c.scan` will produce this output

serial terminal

```
> hex i2c.scan
40
96
 ok
>
```

We now wrap this up in two more words to initialize everything and to run a nice blinky demo. For enhanced illumination I'll connect pin 0 of the PCF8574 with the green channel of the fnordlight.

main.fs

```
: init
  blue pin_output blue high
  red  pin_output red  high
  sda  pin_output sda  high
  scl  pin_output scl  high
  \ alternatively
  \ $ff DDRB c! $ff PORTB c!
  ['] noop is wait
;

: ms 0 ?do 1ms loop ;
variable N   0 N !
: run
  init
  $00 >8io
  &1000 ms
```

```
  begin
    N @ invert >8io
    1 N +!
    &1000 ms
  key? until
  key drop
;
```

We are in business. Remember, there was no re-flashing of the controller. This is not production code, since there is no error checking whatsoever. For real stuff, you want to use the builtin TWI interface anyway. There is a library module `lib/twi.frt` for that purpose, so use that.

Have the appropriate amount of fun!

6 Appendix

6.1 avra

Pulling the newest version from avra's repository (`avra.git.sourceforge.net/git/gitweb.cgi?p=avra/avra` and compiling will result in a version that will assemble **amforth**. There is one bug open: the directives `.nooverlay` and `.overlay` are not handled. You need to hack the file `core/devices/\$(MCU)/device.asm` and remove the directives and any interrupt service routine calls, which are already defined before (e.g. usart rx complete, usart data register empty).

bash

```
$ tar xf avra-b609ce5.tar.gz
$ cd avra-b609ce5/src
$ make -f ./makefiles/Makefile.linux LDFLAGS=''
$ ./avra --version
AVRA: advanced AVR macro assembler Version 1.3.0 Build 1 (8 May 2010)
...
```

The empty LDFLAGS directive was neccessary on my amd64 system to prevent stripping, which caused an *unexpected reloc type*–error. On a i686 system everything worked with stripping.

6.2 configuring amforth-4.2 for atmega–32 from scratch

You need the assembler (avra or AvrAssembler2), programmer and flash tool (avrdude) working at this point.

bash

```
mkdir Forth
cd Forth
svn co https://amforth.svn.sourceforge.net/svnroot/amforth
cd amforth/releases/4.2
ln -s $HOME/wine/AvrAssembler2 Atmel # edit for your location
cd appl
```

```
cp -a template 2011-fosdem-32
rm -fR 2011-fosdem-32/.svn/ 2011-fosdem-32/words/.svn/
cd 2011-fosdem-32

find . | sort
.
./build.xml
./dict_appl_core.inc
./dict_appl.inc
./makefile
./template.asm
./words
./words/applturnkey.asm


vi makefile
# edit MCU LFUSE HFUSE CONSOLE MKII BURNER DIR_ATMEL
# add sudo before avrdude

vi template.asm
# edit F_CPU BAUD
# add USART0 definitions ".equ TXEN0 = TXEN" ...
# .include usart.asm not usart_0.asm

vi dict_appl.inc
# add useful stuff for demo

# if you want to work with patched avra, you need to edit
# make a local copy of ../../core/devices/atmega32/device.asm
# and edit it.

make
make install
minicom -ow atmega-s0 (115200,8N1,no flow control)

ln -s ../../lib lib
../../tools/amforth-upload.py -t /dev/ttyS0 \
lib/misc.frt lib/bitnames.frt lib/ans94/marker.frt \
../../core/devices/atmega32/atmega32.frt lib/ans94/postpone.frt \
lib/dumper.frt
```

6.3 configuring amforth-4.2 for atmega–328p (duemilanove) from scratch

Along the same lines outlined in the previous section, you can configure **amforth** for an arduino duemilanove (or other).

board: arduino duemilanove

controller: atmega328p at 16.000 MHz
fuses: low,high,extended: 0xff, 0xd9, 0x05
serial connection: 9600, 8N1, no flow control
onboard led is connected to PortB.5 and GND

bash
```
cp -a arduino 2011-fosdem-328p
find 2011-fosdem-328p -type d -name '\.svn' | xargs rm -fR {} \;
cp template/makefile 2011-fosdem-328p
cd 2011-fosdem-328p
vi dict_appl.inc # add set-current, set-order for marker
vi makefile      # TARGET=duemilanove
                 # AMFORTH=../../core
                 # LFUSE=0xFF
                 # HFUSE=0xD9
                 # EFUSE=0x05
                 # DIR_ATMEL=...
                 # BURNER=...
                 # AVRASM=wine $(DIR_ATMEL)/avrasm2.exe -I $(DIR_ATMEL)/Appnotes
make install
```

6.4 using am4up with minicom

am4up was announced on the amforth mailing list `http://www.mail-archive.com/amforth-devel@lists.sourceforge.net/msg00252.html` including instructions.

check: am4up does not recognize errors?

7 References

Forth, amforth

1. **amforth** home page: `amforth.sourceforge.net`
2. mailing list: `amforth-devel@lists.sourceforge.net`
3. reference card: `amforth.sourceforge.net/refcard.pdf`
4. Forth Inc. `www.forth.com`
5. `tayeta.com` ???
6. *Leo Brodie — Starting Forth* Online Transkript `home.iae.nl/users/mhx/sf.html`
7. *Leo Brodie — Thinking Forth* Online Transkript `thinking-forth.sourceforge.net`
8. `de.wikipedia.org/wiki/Forth_(Informatik)`
9. `en.wikipedia.org/wiki/Forth_programming_language`

hardware

10. Atmel home page: `www.atmel.com`
11. rumpus board: `www.lochraster.org/rumpus`
12. fnordlight: `www.lochraster.org/fnordlicht`
13. arduino duemilanove ??? `www.arduino.cc`
14. embedit mega-32 proto board: `www.embedit.de`

15. sp12 programmer: www.xs4all.nl/~sbolt/e-spider_prog.html
16. usbasp programmer: www.fischl.de/usbasp

software

17. AvrStudio4 www.atmel.com/dyn/resources/prod_documents/AVRStudio4.18SP3.exe
18. avra: sourceforge.net/projects/avra
19. avrdude: savannah.nongnu.org/projects/avrdude
20. am4up: dl.dropbox.com/u/360343/am4up.c
21. minicom: freshmeat.net/projects/minicom
22. wine: www.winehq.org

other resources

23. www.forth.cz
24. www.mikrocontroller.net
25. www.roboternetz.de
26. www.forth-ev.de Forth Gesellschaft e.V.
27. Forth Interest Group Netherlands www.forth.hccnet.nl/w
28. Forth Interest Group UK www.fig-uk.org
29. Forth Interest Group USA www.forth.org