Forth for AVR ATmega microcontroller

Matthias Trute

Forth for AVR ATmega microcontroller
by Matthias Trute

Published 2007

Table of Contents

List of Tables

Overview

amforth is a Forth system for the AVR Atmega micro controller family. It works on the controller itself and does not depend on any additional hard- or software.

This document covers version 2.5

amforth implements a large subset of the Forth standard ANS94. Most of the CORE and CORE EXT words and a varying number of words from the other word sets are implemented. It is very easy to extend or shrink the actual word list for a specific application by just editing the dictionary include files.

The dictionary is located in the flash memory. The built-in compiler extends it directly.

amforth provides full access to all interrupts. The interrupt handler routines are forth words.

amforth is published under the GNU General Public License version 2.

The name amforth has no special meaning.

amforth is a new implementation. The first code was written in the summer of 2006. It is written "from scratch" using assembly language and forth itself. It does not have a direct relationship to any other forth system.

The first lines of code were written with the AVR Studio and it's simulator. Soon the switch to real hardware (an evaluation board) and to linux based development was done.

Chapter 1. First Steps

The first steps require a working ATmega microcontroller with an RS232 connection to an PC or a terminal like the VT100 or similar hardware. A customization may change these requirements.

1.1. User Interface

amforth has a simple user interface. Connect your system to a serial terminal (or a PC) and you get the forth prompt > .

```
amforth 2.5 ATmega32
>
words
cell+ cells abort abort" [char] ...
>
```

Chapter 2. Hardware

2.1. Fuses

Amforth uses the self programming features of the ATmegas to extend the dictionary. It is ok to use the factory default settings plus (!) changes for the oscillator settings. It is recommended to use a higher CPU frequency to meet the timing requirements of the serial terminal.

Chapter 3. Source Organisation

3.1. Overview

The source code can be processed with both the AVR Studio and the linux avr assembler avra.

amforth needs an assembler to generate the hex files for Flash and EEPROM. That does not mean that every word is actually written in assembly language however. Most of the words are written in forth itself, but are precompiled into assembly syntax. This solves btw the chicken-and-egg problem too: how to compile the compiler words.

amforth consists of a great number of relatively small source files. Nearly all words are coded in their own source file. A number of files are organized with include files, called `dict*.inc` . Currently 3 such files exists: `dict_minimum` , `dict_high.inc` and `dict_compiler.inc` .

There are two additional files: `amforth.asm` and `macros.asm` . The first one is the master file and the only one the application needs to include. The file `macros.asm` contains some useful assembler macros that make the source code easier to read.

3.2. Core system

The file `amforth.asm` is the core of amforth. Here is the startup code for the microcontroller, and the forth inner interpreter with the interrupt service routine. It includes the dictionary files.

3.2.1. Dictionary files

The dictionary files have two tasks: First they include the word definition files. Second, they determine each word's location in the resulting flash layout. The file dict_high.inc contains all words for the NRWW flash section, Since the word **I!** cannot write to this address range, no new words can be compiled to this section at runtime. Thus it is advisable to include as many words as possible in `dict_high.inc` if the amount of writable dictionary space is an issue.

A useful forth system needs at least the file `dict_minimum.inc` , which includes the serial IO and the forth interpreter words.

An almost complete forth system with a compiler needs the third include file: dict_compiler.inc.

There are a few words left out from the dictionary lists. These words are either not always needed or are some variants of existing words or simply cannot be included in the core system due to size limitations in the NRWW section with smaller atmegas.

3.2.2. Device Settings

Every Atmega has it own specific settings. They are based on the official include files provided by Atmel, which define the most important settings for the serial IO port (which port and which parameters), the interrupt vectors and some macros.

The last setting is a string with the device name in clear text. This string is used within the word VER.

3.3. Application Code

Every build of amforth needs an application. There are a few sample applications, which can be used either directly (such as the AVR Butterfly) or serve as a source for inspiration (template application).

The structure is basically always the same. First the file macros.asm has to be included. After that some definitions need to done: The size of the Forth buffers, the CPU frequency, initial terminal settings etc. Then the device specific part needs to be included and as the last step the amforth core is included.

For a comfortable development cycle the use of a make utility such as make itself is recommended. The assembler needs quite a few settings and the proper order of the include directories. The sample applications use the standard make, but others such as ant or maven can be used as well.

Chapter 4. Architecture

4.1. Overview

amforth is a 16 bit Forth implementing the indirect threading model. The flash memory contains the whole dictionary. Some EEPROM cells are used to hold initial values and the dictionary pointers. The RAM contains buffers, variables and the stacks.

The compiler is a classic compiler without any optimization support.

amforth uses most of the CPU registers to hold vital data structures: The data stack pointer, the instruction pointer, the user pointer, and the Top-Of-Stack cell. The hardware stack is used as the return stack. Some registers are used for temporary data in primitives.

4.2. CPU -- Forth VM Mapping

The default Forth registers are mapped as follows

Table 4-1. Register Mapping

Forth Register	ATmega Register(s)
W: Working Register	R26:R27
IP: Instruction Pointer	XH:XL
RSP: Return Stack Pointer	SPH:SPL
PSP: Parameter Stack Pointer	YH:YL
UP: User Pointer	R4:R5
TOS: Top Of Stack	R6:R7
X: temporary register	ZH:ZL

In addition the register pair R0:R1 is used internally e.g. to hold the the result of multiply operations. The register pair R2:R3 is used as the zero value in many words. These registers cannot be used by user programs and should never be changed.

The registers from R8 to R15 are currently unused. The registers R16 to R25 used as temporary registers and can be used freely within one module. They are overwritten by the primitives.

The forth core uses the T bit in the machine status register SREG for signaling an interrupt.

4.3. Core System

4.3.1. Threading Model

amforth uses the classic indirect threaded variant of forth.

4.3.2. Inner Interpreter

For the indirect threading model an inner interpreter will be needed. The interpreter core is responsible for the interrupt handling too.

4.3.2.1. EXECUTE

This operation reads the cell the IP currently points to and uses the value as the destination of a jump.

4.3.2.2. NEXT

The NEXT routine is the core of the inner interpreter. It performs two flash accesses, the second one is accessed as EXECUTE.

The very first step in NEXT is to check whether an interrupt needs to handled. It is done by looking at the **T** flag in the machine status register. If it is set, the code jumps to the interrupt handling part. If the flag is cleared the following normal NEXT routine runs.

4.3.2.3. NEST

NEST (aka DO_COLON) first pushes the IP (which points to the next word to be executed when the current word is done) to the return stack. It then increments W by one flash cell, so that it points to the body of the (colon) word, and sets IP to point to same location. Then it continues with NEXT, which begins executing the words in the body of the colon word

4.3.2.4. UNNEST

The code for UNNEST is the word **EXIT** in the dictionary. It pops the IP from the return stack and jumps to NEXT.

4.3.2.5. DO_DOES

This code is the runtime that is used by the code compiled by the forth word **DOES** . It it closely tied to the action performed by the code compiled by DOES. That code pushed the current address in the data field to the returnstack and jumps to DO_DOES. DO_DOES gets that address back, saves the current IP and sets the forth IP to the address it got from the stack. Finally it continues with NEXT.

4.3.3. Stacks

4.3.3.1. Data Stack

The data stack uses the CPU register pair YH:YL as its data pointer. The Top-Of-Stack element (TOS) is in a register pair. Compared to a straight forward implementation this approach saves both code space and gives higher execution speed (approx 10-20 %). Saving more stack elements does not really provide a greater benefit (much more code and only little speed enhancements).

The data stack starts at a configurable distance below the return stack (RAMEND) and grows downward.

4.3.3.2. Return Stack

The Return Stack is the hardware stack of the controller. It is managed with push/pop instructions. The return stack starts at RAMEND und grows downward.

4.3.4. Interrupts

amforth routes the low level interrupts into the forth inner interpreter. The inner interpreter switches the execution to a predefined word if an interrupt occurs. When that word finishes execution, the interrupted word is continued. The interrupt handlers are completely normal forth words without any stack effect.

The processing of the interrupts takes two steps: The first one is responsible for the low level part. It is called whenever an interrupt occurs. The code is the same for all interrupts. It takes the number of the interrupt from its vector address and stores this in a RAM cell. Then the low level ISR sets the **T** flag in the status register of the controller. The inner interpreter checks this flaf every time it is entered and if it is set it switches to interrupt handling at forth level. This approach has a penalty of 2 CPU cycles for checking and skipping the branch instruction to the isr forth code.

The ISR at forth level is a RAM based table much like the low level interrupt table of the execution tokens associated with the interrupt number.

Interrupts from hardware sources (such as the usart) may not work as expected. The reason is that the interrupt source is not cleared within the generic ISR. This leads to an immediate re-interrupt. There is currently no solution but a custom ISR that clears the interrupt source and calls the main ISR. This code has to be run within the interrupt and cannot be (easily) turned into forth code, since the forth inner interpreter is not reentrant.

4.3.5. Multitasking

amforth does not really implement multitasking. It only provides the basic functions for it. Within the IO words the word **PAUSE** is called whenever possible. This word is as a deferred word that is initialized to do nothing (NOOP).

The Library contains a cooperative multitasker (`multitask.frt`). It is possible to use a timer interrupt to call the command **pause** and turn the cooperative multitasker into a preemptive one. The latency is in the worst case that of the longest running primitive forth command: **1ms** . For this kind of task switcher a lot more tools like semaphores may be needed.

4.3.6. Exception Handling

amforth implements the CATCH/THROW exception handling. The outermost catch frame is located at the interpreter level in the word QUIT. If an exception with the value -1 or -2 is thrown, QUIT will print a message and re-start itself. Other values silently restart QUIT.

4.3.7. User Area

The User Area is a special RAM based storage area. It contains the USER variables and the User deferred definitions. Access is based upon the value of the user pointer UP. It can be changed with the word UP! and read with UP@. The UP itself is stored in a register pair.

The size of the user area is defined at compile time in the device definition section. This may change in future versions.

The User Area is used to provide task local information. Without an active multitasker it contains the starting values for the stackpointers, the deferred words for terminal IO, the BASE variable and the exception handler.

A multitasker can use the first 2 cells for own purposes. In that situation the user area is/can be seen as the task control block.

4.4. Memory Layout

4.4.1. Flash

The flash memory is divided into 5 sections. The first section, starting at address 0, contains the interrupt vector table for the low level interrupt handling and a character string with the name of the controller in plain text.

The next section is the initialization code block. It is executed whenever the controller starts. The code sets up the basic infrastructure for the forth interpreter. This step is finished by calling the forth interpreter with the word COLD as the entry word.

The 3rd section contains the low level interrupt handling routines. The interrupt handler is very closely tied to the inner interpreter. It is located near the first section to use the faster relative jump instructions.

The 4th section is the first part of the dictionary. Nearly all colon words are located here. New words are appended to this section. This section is filled with FFFF cells when flashing the controller initially.

The last section is identical to the boot loader section of the ATmegas. It is also known as the NRWW area. Here is the heart of amforth: The inner interpreter and most of the words coded in Assembly language.

The reason for this split is a technical one: to work with a dictionary in flash the controller needs to write to the flash. The ATmega architecture provides a mechanism called self-programming by using a special instruction and a rather complex algorithm. This instruction only works in the boot loader/NRWW section. amforth uses this instruction in the word I!. Due to the fact that the self programming is a lot more then only a simple instruction, amforth needs most of the forth core system to achieve it. A side effect is that amforth cannot co-exist with classic bootloaders. If a particular boot loader provides an API to enable applications call the flash write operation, amforth can be restructured to use it. Currently only very few and seldom used bootloaders exist that enable this feature.

Atmegas can have more than 64 KB Flash. This requires more than a 16 bit address, which is more than the cell size. For one type of those bigger atmegas there will be an solution with 16 bit cell size: Atmega128 Controllers. They can use the whole address range with an interpretation: The flash addresses are in fact not byte addresses but word addresses. Since amforth does not deal with bytes but cells it is possible to use the whole address range with a 16 bit cell. The Atmegas with 128 KBytes Flash operate slightly slower since the address interpretation does need more code to access the flash (both read and write).

The technique described above does not work for the Atmega256x. These controllers definitely need a bigger cell size: 17 bits (or more).

4.4.1.1. Flash Write

The word performing the actual flash write operation is I! (i-store). This word takes the value and the address of a single cell to be written to flash from the data stack. The address is a word address, not a byte address!

The flash write strategy follows Atmel's appnotes. The first step is turning off all interrupts. Then the affected flash page is read into the flash page buffer. While doing the copying a check is performed whether a flash erase cycle is needed. The flash erase can be avoided if no bit is turned from 0 to 1. Only if a bit is switched from 0 to 1 must a flash page erase operation be done. In the fourth step the new flash data is written and the flash is set back to normal operation and the interrupt flag is restored.

This write strategy ensures that the flash has minimal flash erase cycles while extending the dictionary. In addition it keeps the forth system simple since it does not need to deal with page sizes or RAM based buffers for dictionary operations.

4.4.2. EEPROM

The built-in EEPROM contains vital dictionary pointer and other persistent data. They need only a few EEPROM cells. The remaining space is available for user programs. The easiest way to use EEPROM is the use of forth VALUEs. There intended design pattern (read often, write seldom) is like that for the typical EEPROM usage.

Another use for EEPROM cells is to hold execution tokens. The default system uses this for the turnkey vector. This is an EEPROM variable that reads and executes the XT at runtime. It is based on the DEFER/IS standard. To define a deferred word in the EEPROM use the Edefer defintion word. The standard word IS is used to put a new XT into the vector.

Low level space management is done through the the EDP variable. This is not a forth value but a EEPROM based variable. To read the current value an `e@` operation must be used, changes are written back with `e!`. It contains the highest EEPROM address currently allocated. The name is based on the DP variable, which points to the highest dictionary address.

4.4.3. RAM

The RAM address space is divided into three sections: the first 32 addresses are the CPU registers. Above come the IO registers and extended IO registers and finally the RAM itself.

amforth needs a few (real) RAM locations for its internal data structures. The biggest part are the buffers for the terminal IO. RAM Memory is managed by the words VARIABLE and ALLOT.

With amforth all three sections can be accessed using their RAM addresses. That makes it quite easy to work with words like **C@** . The word **!** implements a LSB byte order: The lower part of the cell is stored at the lower address.

For the RAM there is the word **Rdefer** which implements a deferred word, placed in RAM. As a special case there is the word **Udefer** , which sets up a deferred word in the user area. To put an XT into them the word **IS** is used. This word is smart enough to distinguish between the various xDefer definitions.

Chapter 5. Forth Implementation

5.1. ANS Words

amforth implements most or all words from the ANS word sets CORE, CORE EXT, EXCEPTION and DOUBLE NUMBERS. The words from the word sets LOCALS, BLOCKS, FILE-ACCESS and FLOATING-POINT are dropped completly.

5.1.1. Core and Core EXT

From the CORE word set only the words >NUMBER, C, CHAR+, CHAR, ENVIRONMENT?, EVALUATE, MOVE are missing. From the CORE EXT the words C", COMPILE, , CONVERT, EXPECT, SPAN, PICK, RESTORE-INPUT, ROLL are not implemented.

The following words have non-standard behavior

words created with **:** are immediately visible. An earlier definition with the same name will never be accessible. Work around may be done with **DEFER / IS** .

loop counters are checked on signed compares.

5.1.2. Block

amforth does not currently support block related words. Implementing them is on the roadmap.

5.1.3. Double Number

Double cell numbers do work as expected. Not all words are implemented. Entering them directly using the dot- notation does not work currently.

5.1.4. Exception

Exceptions are fully supported. The words ABORT and ABORT" use them internally.

The THROW codes -1, -2 and -13 work as specified.

The implementation is based upon a variable HANDLER which holds the current return stack pointer position. This variable is a USER variable.

5.1.5. Facility

The basic system uses the KEY? and EMIT? words as deferred words.

The word MS is implemented as the word 1MS which busy waits almost exactly 1 millisecond. The calculation is based upon the frequency specified at compile time.

The words TIME&DATE, EKEY, EKEY>CHAR are not implemented.

To control a VT100 terminal the words AT-XY and PAGE are written in forth code. They emit the ANSI control codes according to the VT100 terminal codes.

5.1.6. File Access

amforth does not have filesystem support. It does not support any words from this word set.

5.1.7. Floating Point

amforth does not currently support floating point numbers.

5.1.8. Locals

amforth does not currently support locals.

5.1.9. Memory Allocation

amforth does not support the words from the memory allocation word set.

5.1.10. Programming Tools

Variants of the words .S ? and DUMP are implemented or can easily be done. The word SEE won't be supported since amforth highly uses an optimization strategy to strip forth headers whenever possible.

The other reason for dropping SEE is that amforth is OpenSource software. If your vendor does not disclose the full source, let me know. He violates the GPL.

STATE works as specified.

The word **WORDS** does not sort the word list and does not take care of screen sizes.

The words **CODE ;CODE** and **ASSEMBLER** are not supported, amforth does not have an assembler.

CS-ROLL , **CS-PICK** and **AHEAD** are not implemented. The compiler words operate with the more traditional **MARK / RESOLVE** word pairs.

FORGET is implemented but does not fully reset the dictionary state. The better way is using **MARKER** .

An EDITOR is not implemented.

[IF] , **[ELSE]** and **[THEN]** are not implemented.

5.1.11. Search Order

amforth does not support word lists, so no words from the search word set are implemented.

5.1.12. Strings

SLITERAL , **CMOVE>** and **/STRING** are implemented.

-TRAILING , **BLANK** , **CMOVE** , **COMPARE** and **SEARCH** are not implemented.

5.2. amforth extensions

5.2.1. MCU Access

amforth provides wrapper words for the microcontroller words sleep and wdr (watch dog reset). To work properly, the MCU needs more configuration. amforth itself does not call these words.

5.2.2. Memory

Atmega microcontrollers have three different types of memory. RAM, EEPROM and Flash. The words @ and **!** work on the RAM address space (which includes IO Ports and the CPU register), the words **e@** and **e!** operate on the EEPROM and **i@** and **i!** deal with the flash memory. All these words transfer one cell (2 bytes) between the memory and the data stack. The address is always the native address of the target storage: byte-level for EEPROM and RAM, word-level for flash. Therefore the flash addresses 64KWords == 128 KBytes address space.

External RAM shares the normal RAM address space after initialization (which needs to be done within the turnkey action). It is accessible without further action.

For RAM only there is the special word pair **c@ / c!** which uses the lower byte of the Top-Of-Stack at transfer. The upper byte is either ignored or set to 0 (zero).

All other types of external memory need special handling, which may be masked with the block word set.

5.2.3. Input Output

amforth uses simple terminal io. A serial console is used. All IO is based upon the standard words **EMIT / EMIT?** and **KEY / KEY?** . In addition the word **/KEY** is used to signal the sender to stop. All these words are deferred words in the USER area and can be changed with the **IS** command.

The predefined words use an interrupt driven IO with an buffer for input and output. They do not implement a handshake procedure (XON/XOFF or CTS/RTS). The default terminal device is USART0 (if more than one USART port is available).

The basic words include a call of the **PAUSE** command to enable the use of multitasking.

Other IO depend on the hardware connected to the microcontroller. Code exists to use LCD and TV devices. CAN, USB or I2C are possible as well. Another use of the redirect feature is the following: consider some input data in external EEPROM (or SD-Cards). To read it, the words **KEY** and **KEY?** are redirected to fetch the data from them.

5.2.4. Strings

Strings can be stored in two areas: RAM and FLASH. It is not possible to distinguish between the storage areas based on the addresses found on the data stack, it's up to the developer to keep track.

Strings are usually stored as counted strings. Strings in flash are compressed: two consecutive bytes are placed into one flash cell. The standard word **S"** copies the string from the RAM into flash using the word **S,** .

Chapter 6. Tools

6.1. Host

There a few number of tools on the hostside (PC) that are specifically written to support amforth. They are written in script languages like perl and python and should work on all major operating systems.

6.1.1. Documentation

The tool **makerefcard** reads the assembly files from the `words` subdirectory and creates a reference card. The resulting LaTeX file needs to be processed with latex to generate a nice looking overview of all words available in the amforth core system.

The command **make-htmlwords** creates the linked overview of all words on the amforth homepage.

6.1.2. Uploader

To transfer forth code to the microcontroller some precautions need to taken. During a flash write operation all interrupts are turned off. This may lead to lost characters on the serial line. One solution is to send very slowly and hope that the receiver gets all characters. The program **ascii-xfer** can do the job:

ascii-xfr -s -c $delay_char -l $delay_line $file > $tty

This works but the upload of longer files needs a very long time.

Another solution is **amforth-upload.py** . It was initially created by user *pix* (http://pix.test.at/) . His algorithm checks for the echo of every character sent to the controller. At line ends the uploader waits for the ok prompt to continue with the next line.

This algorithm works very fast without the risk of lost characters. An extension of this script provides limited library support. In the source files a command

#include filename

is used to upload the content of `filename` instead of the two words. The sources will only work with this uploader utility, others will trigger the "word-not-found" exception on the microcontroller unless they recognize the #include syntax (similar to the c preprocessor).

Chapter 7. Todos And Roadmap

7.1. ANS Words

Support for Blocks may be useful. It is not trivial to implement a standard 1KB block buffer on an Atmega with only 1KB RAM. It can be useful to deploy block sizes smaller than 1KB to match the native block sizes of the attached storage devices: serial EEPROM have e.g. 64 bytes, SD-Cards have 512 bytes.

7.2. More Controller Types

amforth can run on the whole range of Atmegas. The only limiting factor is the flash size: amforth needs 7 KB for itself and can address 128 KB. The ATmega256x may be supported with a change in the cell size from 2 to 3 bytes. The other possible devices are the XMega MCU, that Atmel may publish in the near future. ATtiny devices are not supported since they lack both flash size and a few instructions that amforth uses.