amforth 3.1

Technical Documentation

Matthias Trute

amforth 3.1Technical Documentation

by Matthias Trute

Published 2008

Table of Contents

List of Tables

Overview

amforth is a Forth system for the AVR ATmega microcontroller family. It works on the controller itself and does not depend on any additional hard- or software. It places no restrictions on using external hardware.

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 can be code or 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.

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 3.1 ATmega32
>
words
d2/ s>d up! up@ 0 1ms >< cmove cmove> i! i@ unloop i sp! ...
>
```

Chapter 2. Hardware

2.1. Fuses

Amforth uses the self programming feature of the ATmega microcontrollers to work with the dictionary. It is ok to use the factory default settings plus the 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

amforth is written using the standard Atmel AVR 8 bit assembly language. 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 the assembler syntax. This solves the chicken-and-egg problem: how to compile the compiler words.

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

amforth consists of a great number of small source files. Nearly all words are coded in their own source files. These files are organized with include files, named after the pattern `dict*.inc` . Currently 4 such files exists: `dict_minimum` , `dict_mcu` , `dict_core.inc` and `dict_compiler.inc` . The order in which the files are included defines the search order and there location within the flash memory. Most words can be moved from one include file to another to optimize the flash usage.

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_core.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_core.inc` if the amount of writable dictionary space is an issue.

A useful forth system needs in addition to the above at least the file `dict_minimum.inc` , which includes the forth interpreter words.

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

Some words have their source files within the `core/words` directory but have to be included via the `dict_appl.inc` file. These words define the hardware dependecies to access the amforth system. The serial line terminal is an example.

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. They are usually included by the application specific include file(s).

3.2.2. Device Settings

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

Adapting another ATmega microcontroller is as easy as copy and edit an existing file from a similiar type.

The last definition 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 (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 build utility such as **make** or **ant** is recommended. The assembler needs a few settings and the proper order of the include directories.

Chapter 4. Architecture

4.1. Overview

amforth is a 16 bit Forth implementing the indirect threading model. The flash memory contains the whole dictionary. A few 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: 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	R24:R25
IP: Instruction Pointer	XH:XL (R27:R26)
RSP: Return Stack Pointer	SPH:SPL
PSP: Parameter Stack Pointer	YH:YL (R29:R28)
UP: User Pointer	R4:R5
TOS: Top Of Stack	R22:R23
X: temporary (scratch pad) register	ZH:ZL (R31:R30)

Table 4-2. Extended Forth VM Register Mapping

Forth Register	ATmega Register(s)
A: Index and Scratch Register	R6:R7
B: Index and Scratch Register	R8:R9

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 must never be changed.

The registers from R10 to R13 are currently unused, but may be used for the VM extended registers X and Y sometimes. The registers R14 to R21 are used as temporary registers and can be used freely within one module as temp0 to temp7.

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

4.3. Core System

4.3.1. Threading Model

amforth implements the classic indirect threaded variant of forth.

4.3.2. Inner Interpreter

For the indirect threading model an inner interpreter is needed. The inner interpreter does the interrupt handling too.

4.3.2.1. EXECUTE

This operation reads the cell the IP currently points to and uses the value read as the destination of a branch. This EXECUTE is not the forth word EXECUTE. The forth EXECUTE sets the IP from the data stack TOS element.

4.3.2.2. NEXT

The NEXT routine is the core of the inner interpreter. It consists of 4 steps which are executed for every forth word.

The first step in NEXT is to check whether an interrupt needs to be 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.

The next step is to read the cell the IP points to and stores this value in the W register. For a COLON word W contains the address of the code field.

The 3rd step is to increase the IP register by 1.

The 4th step is to read the content of the cell the W register points to. The value is stored in the scratch pad register X. The data in X is the address of the machine code to be executed in the last step.

This last step finally jumps to the machine code pointed to by the X scratch pad register.

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 that value. Then it continues with NEXT, which begins executing the words in the body of the (parent) colon word.

4.3.2.4. UNNEST

The code for UNNEST is the forth word **EXIT** in the dictionary. It reads the IP from the return stack and jumps to NEXT. The return stack pointer is incremented by 2 (1 flash cell).

4.3.2.5. DO_DOES

This code is the runtime part of the forth word **DOES** . It pushes the current address of the MCU IP register onto 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 code space and gives higher execution speed (approx 10-20%). Saving even 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 assembler instructions. The default 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 completly normal forth colon 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 flag 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 if no interrupt occured.

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 when the ISR is left. 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 implement multitasking directly. It only provides the basic functions. Within IO words the deferred word **PAUSE** is called whenever possible. This word is initialized to do nothing (**NOOP**).

4.3.6. Exception Handling

amforth implements the **CATCH** and **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 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.

The multitasker uses the first 2 cells to store the status and the link to the next entry in the task list. In that situation the user area is/can be seen as the task control block.

4.4. Memory Layout

4.4.1. Flash

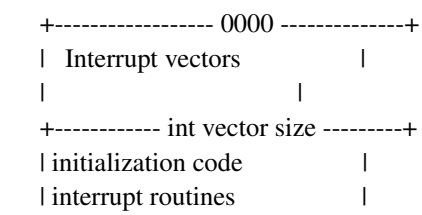
The flash memory is divided into 5 sections. The first section, starting at address 0, contains the interrupt vector table for the low level interrupt handling and a character string with the name of the controller in plain text.

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.



```
|    "colon" dictionary                |
|                                 |
|LATEST:                          |
|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|
+------------- nrww section -----------+
|   inner interpreter (docol, next)    |
| "primitive" dictionary               |
| Link to "colon" dictionary           |
+----------- flashend -----------------+
```

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 to 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 trick: 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 needs more code to access the flash (both read and write). The source code uses assembly macros to hide the differences.

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

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. The whole process takes a few milliseconds.

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 it.

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 very little (real) RAM space 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** .

Figure 4-1. RAM Structure Overview

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 defines 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 Words

5.1. ANS Words

amforth is not fully ANS94 compatible. The main difference comes from the fact that the AVR ATmegas use a Havard architecture (separate code and data address space) that amforth does not hide. amforth gives full and unmodified access to the whole address space.

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, FILE-ACCESS and FLOATING-POINT are dropped completly. The others are partially implemented.

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 arounds may be done with **DEFER** and **IS** .

Loop counters are checked on signed compares.

5.1.2. Block

amforth has limited block support with I2C/TWI serial eeprom chips with 2 byte addresses.

5.1.3. Double Number

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

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 in the USER area.

The word **MS** can implemented with 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 contain 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 the 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** and **ASSEMBLER** are not supported. amforth has a loadable assembler which can be used with the words **CODE** and **END-CODE** .

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** from the library.

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 instructions **SLEEP** and **WDR** (watch dog reset). To work properly, the MCU needs more configuration. amforth itself does not call these words.

Microcontrollers supporting the JTAG interface can be programmed to turn off JTAG at runtime. Similiar the watch dog timer can be disabled. Since both actions require strict timing they need to be implemented as primitives: **-JTAG** and **-WDT** .

5.2.2. Assembler

Lubos Pekny has written an assembler for amforth. To support it, amforth provides the two words CODE and END-CODE. The first creates a dictionary entry and sets the code field to the data filed address. The interpreter will thus jump directly into the data field assuming some machine code there. The word END-CODE places a JUMP NEXT into the data field. This finishes the machine instruction execution and jumps back to the forth interpreter.

5.2.3. Memory

Atmega microcontroller 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-based for EEPROM and RAM, word-based for flash. Therefore the flash addresses 64KWords or 128 KBytes address space.

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

For RAM only there is the special word pair **c@** / **c!** which operate with the lower half of a stack cell. 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.

5.2.4. 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 a 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).

These basic words include a call to 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?** can be redirected to fetch the data from them.

5.2.5. 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 stored as counted strings with a 16 bit counter value (1 flash cell) 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. Library

Amforth does not have a formal library concept. Amforth has a lot of forth code that can be seen as a library of words and commands.

6.1. Hardware Access

In the `device/` subdirectory are the controller specific register definitions. They are taken directly from the appnotes from Atmel. The register names are all uppercase. It is recommended to extract only the needed definitions since the whole list occupy a lot of flash memory.

Some commonly used lowlevel words can be included with the `dict_mcu.inc` include file at compile time.

6.2. Software Modules

6.2.1. Multitasking

The Library contains a cooperative multitasker in the file `multitask.frt` . It defines a command **multitaskpause** which can assigned to **pause** : ' multitaskpause is pause

The multitasker has the following commands

onlytask (--)

Initialize the task system. The current task is placed as the only task in the task list.

alsotask (tid --)

Append a newly created task to the task list. A running multitasker is temporarily stopped. Make sure that the status of the task is sleep.

task (dstacksize rstacksize -- tid)

Allocate RAM for the task control block (aka user area) and the two stacks. Initializes the whole user area to direct IO to the serial line. The task has still no code associated and is not inserted to the task list.

task-sleep (tid --)

Let the (other) task sleep. The task switcher skips the task on the next round. When a task executes this command for itself, the task continues until the next call of **pause** .

task-awake (tid --)

The task is put into runnable mode. It is not activated immediately.

activate (tid --)

Skip all of the remaining code in the current colon word and continue the skipped code as task when the task list entry is reached by the multitasker.

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 uninterruptable forth commands: **1ms** , **e!** and **i!** . For a preemptive task switcher a lot more tools like semaphores may be needed.

6.2.2. TWI / I2C

The file `twi.frt` contains the basic words to operate with the hardware TWI module of the microcontroller. The file `twi-eeprom.frt` uses these words to implement a native block buffer access for I2C EEPROMs with 2byte addresses.

The word **+twi** initializes the TWI hardware module with the supplied parameters. **-twi** turns the module off. The start-stop conditions are sent with the **twi.start** and **twi.stop** words. Data is transferred with the three words **twi.tx** for transmitting (sending) a byte, **twi.rx** for reading a byte (and sending an ACK signal) and **twi.rxn** for reading a byte and sending a NACK signal.

The command **twi.status** fetches the TWI status register, the command **twi.status?** compares the status with a predefined value and throws the exception -14 if they do not match.

The command **twi.scan** scans the whole (7 bit) address range and prints the address of any device found.

6.2.3. I2C EEPROM

I2C EEPROMs can be used in varios ways. The file `twi-eeprom.frt` defines words to access the EEPROM at byte address level and at block level. A page is the native block size of the eeprom device, that is stored in the **VALUE twi.ee-b/blk** . The hardware (i2c-) device address is stored in the value **twi.ee-addr** . Currently EEPROM devices with 2byte addresses are supported.

Byte level access is done with the words **twi.ee-c!** and **twi.ee-c@** . They transfer one byte from/to the eeprom address given. The stack diagram is exactly the same as for the RAM **c@ / c!** . Every store operation performes an full EEPROM erase/write cycle.

To transfer more bytes the block level words can be used. The transfer a whole EEPROM page to/from RAM. The first page is at address 0, page 1 starts at address **twi.ee-b/blk** .

Chapter 7. Tools

7.1. Host

There a few number of tools on the host side (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.

7.1.1. 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.

7.1.2. 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: $delay_char can be 1 or 2 ms, $delay_line around 800 ms.

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 8. Final Remarks

8.1. More ANS94 Words

There are a few missing words from the standard CORE word set. Many of them are related with string parameters, like **evaluate** and **environment?** . The difficulty arises from the fact, that the storage location of a string cannot be determined by simply lookaing at the address. A solution may be a state smart implementation with some helper words. If running interactivly, these words may use RAM addresses, if called within a compiled word they use flash addresses. Not really smart however..

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. Some rather simple code can be used from the library for I2C/TWI EEPROM modules with native block sizes.

8.2. More Controller Types

amforth can run on the whole range of Atmegas. The only limiting factor is the flash size: amforth needs ca 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. ATtiny devices are not supported since they lack both flash size and a few instructions that amforth uses.

8.3. Contributors

amforth would not be the system it now is without the feedback and help from its users. I would like to thank all of them. The following people made an outstanding work to improve amforth (in no particular order): Milan Horkel, Ullrich Hoffmann, Michael Kalus, Karl Lunt, Bruce Wolk, Lubos Pekny. But there are many more that helped by simply asking how to do some tasks.

8.4. Support

Amforth is not a commercial software. I hesitate to call it a product. Since you get all the source code for the system, you should be able to solve all problems yourself. On the other side I'm more than interested in any use of amforth and want to know what you're doing with it. If you find anything strange or faulty don't hesitate to mail it to the mailing list (mailto:amforth-devel@lists.sourceforge.net) .