amforth User's Manual
(for amforth v2.6)

Document contributed to the amforth project on SourceForge.net.

Introduction

This manual describes the amforth programming language and provides details on how to customize the standard release for use on your target platform. This document focuses on developing amforth applications in the Windows environment, using Atmel's freeware AVRStudio4. For information on developing amforth applications in the Linux/Unix environment, consult the Web.

amforth is a variant of the ans94 Forth language, designed for the AVR ATmega family of microcontrollers (MCUs). amforth was developed and maintained by Matthias Trute; the amforth project is hosted and maintained on SourceForge.net (http://sourceforge.net/projects/amforth).

You create your amforth application by creating a custom template file, perhaps modifying some of the files included in the distribution set, assembling them with AVRStudio4, then moving the resulting object file into flash in the target hardware. Your amforth application resides in flash, using very little RAM or EEPROM.

amforth is fully interactive. You can connect your target hardware to a serial port on your host PC, run a comm program such as Hyperterm, and develop additional Forth words and applications on the target. Each word you create interactively is also stored in flash and is available following reset or power-cycle of the target. You can select one of your amforth words to be the boot task, which will run automatically on the next reset or power-cycle. You can also write interrupt service routines (ISRs) in amforth for handling low-level, time-critical events.

This manual assumes familiarity with Atmel's AVRStudio4 development suite (Windows OS); some knowledge of AVR assembly language programming is helpful but not necessary for basic use of the amforth language. You can download the free AVRStudio4 suite from Atmel (http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725). You can use either version 4.12 or 4.13; note that you need to fill out a fairly simple registration page to complete the download.

This manual also assumes a working knowledge of the Forth programming language. If you are new to Forth, or if you need a quick refresher or a reference page, you can find a full Web version of Leo Brodie's marvelous book, “Starting Forth,” online at http://home.iae.nl/users/mhx/sf.html.

About this document

I created this document to support Matthias' work on amforth. He has designed an excellent Forth that I enjoy using, but I thought newcomers to amforth could use a bit more detailed explanation on setting up an application. As my contribution to his amforth project, I'm providing this user's manual.

This document was written using OpenOffice version 2.1. You can download a copy of OpenOffice at the project's website: www.openoffice.org

I have contributed this document to Matthias for inclusion in his project, with the hope that others will edit and expand this text, to make amforth even better. If you modify this document, please include this section (with suitable additions) and include my name in the list of contributors.

Karl Lunt, Bothell, WA USA, 23 September 2007

This document was originally created for version 2.3, this document is kept in sync with the current versions of amforth.

Starting amforth

amforth is available as two different downloads; the core (amforth-<version>.zip) and a small set of library words (am-lib-<version>.zip). Download both files into your working folder (I'll use c:\amforth throughout this document) and unzip them, preserving the subdirectory structure. You should end up with the following subdirectory layout:

```
c:\amforth\amforth-2.6\                <-- main directory, holds your project files
                       \words          <-- holds amforth words (assembler source)
                       \devices        <-- holds declarations for different Atmel MCUs
c:\amforth\am-lib-2.6\                 <-- library directory
```

You will develop your custom applications by creating simple assembly language source files in the main directory, assembling your source files and the amforth source files with AVRStudio4, then downloading the resulting .hex file into your target with AVRStudio4.

Note that although you will be creating assembly language source files, the files will be little more than a few macro invocations. Generally, you will not need to know any AVR assembly language to develop your applications.

The following sections describe the various subdirectories and files found in the initial installation of amforth.

words/ subdirectory

The words/ subdirectory holds a large collection of amforth words, each defined in a separate .asm file. Each file in this subdirectory is a complete word definition, ready to assemble into the final application. For example, here is the entire contents of equal.asm, the source file for the word **=**, which compares two values on the top of the stack and leaves behind a flag that is TRUE if the values match or FALSE if they don't.

```
; ( n1 n2 -- flag ) Compare
; R( -- )
; compares two values
VE_EQUAL:
    .db $01, "="
    .dw VE_HEAD
    .set VE_HEAD = VE_EQUAL
XT_EQUAL:
    .dw PFA_EQUAL
PFA_EQUAL:
    ld temp2, Y+
    ld temp3, Y+
    cp tosl, temp2
    cpc tosh, temp3
PFA_EQUALDONE:
    movw zl, zerol
    brne PFA_EQUAL1
    sbiw zl, 1
PFA_EQUAL1:
    movw tosl, zl
    rjmp DO_NEXT
```

This source file gives an excellent view of the layout for each amforth word. It isn't necessary that you understand how a word is laid out inside the dictionary in order to use amforth. However, if you ever need to define your own amforth words, you will need to follow the layout shown here to make sure your words properly integrate into the dictionary.

devices/ subdirectory

The files in the devices/ subdirectory define a target MCU. For each target MCU defined, you will find a .asm and a .frt file.

The .asm file is intended to be included as part of your application, and provides assembly language definitions for MCU-specific parameters, such as where RAM starts, the number and layout of the interrupt vectors, and where high flash memory starts.

The .frt file is intended to be included as part of any amforth applications you might create with your new executable, and provides Forth-style definitions of MCU-specific parameters, such as IO register names and addresses.

If you need to develop support files for a different Atmel MCU, you can copy an existing pair of files for similar device and use these copies as a starting point for your efforts. Rename the files to match the target MCU, then refer to the appropriate Atmel reference manual to determine the proper values for the various registers and parameters. Where possible, ensure that the assembly language names for the ports match those in the existing .asm file. If the names do not match, or if you need to add new names to provide support for a new subsystem (such as the CANBus ports on an AT90CAN128), you may need to edit one or more existing amforth source files.

amforth.asm

amforth.asm contains a small amount of assembly language source that:

- defines the Forth inner interpreter,
- declares the starting address of the Forth kernel,
- allocates the memory for the final system,
- sets up the small area of EEPROM used by the Forth system,
- declares the interrupt service routine (ISR) support,
- adds the words to be assembled into low flash memory (dict_minimum.inc),
- adds the words to be assembled into high flash memory (dict_high.inc),
- adds the words specific to your application, if any (dict_appl.inc).

The amforth.asm included in your original download is essentially complete, in that when assembled it will create most of an amforth system. However, key information about the target hardware is missing and must be supplied by you in what is known as a template file. Details on what this template file contains and how you use the template file to describe your target hardware are contained in a later section.

Note that amforth.asm refers to a turnkey application (see XT_APPLTURNKEY in the .eseg segment at the end of the file). This execution token is assumed to be an amforth word defined somewhere in your application. The default turnkey application, defined in the file applturnkey.asm, provides a typical Forth interactive environment. You can customize applturnkey.asm as needed; see the section below on applturnkey.asm.

dict_high.inc

This file lists all amforth core words that will be included in that part of amforth's dictionary stored in high flash memory (also known as the bootloader area). The following excerpt from the standard dict_high.inc file shows how the file fits into the amforth system.

```
; this part of the dictionary has to fit into the nrww flash
; section together with the forth inner interpreter
.include "words/int-on.asm"
.include "words/int-off.asm"
.include "words/int-restore.asm"

.include "words/exit.asm"
.include "words/execute.asm"
.include "words/dobranch.asm"
.include "words/docondbranch.asm"

.include "words/estore.asm"
.include "words/efetch.asm"
```

Each line in dict_high.inc is either a comment or an .include statement that adds an assembly language source file from the words/ folder. Thus, the words listed in this file and the order in which they appear define the high-memory portion of your amforth project dictionary.

In general, the words in dict_high.inc provide flash read and write capability, though other primitives, such as branch operations and the inner interpreter, are also included. The amforth design puts these flash read/write operations in high flash memory because the AVR MCU does not allow instructions in one area of flash memory to modify that same area of flash memory. Putting the flash read/write primitives in high flash memory allows amforth words in low flash memory to use these primitives to modify the dictionary, which is kept in low flash.

The amount of high flash memory varies based on the type of AVR device. Some small devices may have so little high flash memory that you won't be able to fit all of the words in this file into the available memory. Should this happen, you may be able to move some of the .include statements from this file into dict_minimum.inc (see below). Note, however, that this must be done carefully, to keep the flash read/write words (at least) in high flash memory.

For nearly all applications, you can simply leave dict_high.inc unchanged.

dict_minimum.inc

This file was called dict_low.inc in previous versions of amforth. Like dict_high.inc, it contains a series of .include statements that add a list of core amforth words to the final system.

The words included in dict_minimum.inc are stored in the target system in low flash memory. The entire dictionary in low flash memory that is created when you build an amforth system becomes your application's dictionary. The last word in dict_minimum.inc (j), will be the last word created in your application, assuming you don't add any other words as you build your application.

You could, if you chose, edit dict_minimum.inc to change the order or content of your application's dictionary. However, this really should be avoided. Instead, use dict_minimum.inc as provided and add your own custom .include file that defines your application. That file, named dict_appl.inc, is discussed below.

dict_appl.inc

The file dict_appl.inc contains those amforth words that you want to add to your application above and beyond the words added by dict_minimum.inc and dict_high.inc. Note that the files dict_minimum.inc and dict_high.inc do not include all of the amforth words available in the words/ folder. For example, they do not include the string operations words, such as **s"**, nor do they include the word **words**, used to list all words in the dictionary.

You create your own dict_appl.inc file using a standard text editor, such as AVRStudio4's text editor. Here is a simple dict_appl.inc file that I use for creating a typical amforth development system. It provides words for printing unsigned numbers, listing the dictionary, and displaying parts of memory.

```
;
;  dict_appl.inc    add application-specific words
;

.include "words/udot.asm"
.include "words/words.asm"

.include "dict_compiler.inc"  ; <-- if you want an extendible system
```

You are free to add whatever words from the words/ folder you want in your application, using your custom dict_appl.inc file. Note that if you accidentally include a duplicate word, assembling the resulting application will generate an error; you will need to edit out the duplicate word and reassemble.

applturnkey.asm

This file contains the assembly language source for a turnkey application, as called out in the file amforth.asm (see appropriate section above). The file applturnkey.asm usually contains amforth words, written in assembly language, in the form shown in the words/ subdirectory description above.

The following sample applturnkey.asm file shows the default turnkey application. This application is suitable for most amforth systems and should be included when you build your system.

```
; ( -- ) System
; R( -- )
; application specific turnkey action
VE_APPLTURNKEY:
    .db 11, "applturnkey"
    .dw VE_HEAD
    .set VE_HEAD = VE_APPLTURNKEY
XT_APPLTURNKEY:
    .dw DO_COLON
PFA_APPLTURNKEY:
    .dw XT_BAUD0
    .dw XT_USART0
    .dw XT_TOUSART0
    .dw XT_INTON
    .dw XT_VER
    .dw XT_EXIT
```

The above simple application sets the baud rate for the serial terminal to the baud rate defined in the template file, initializes USART0 for use as a serial port, turns on interrupts, and displays amforth's version information on the serial terminal.

The template file

You define the characteristics of your target hardware and your application in a template file. The template file is an assembly language source file you create with a standard ASCII text editor, such as AVRStudio4's text editor. Although this is called an assembly language source file, you will typically not write any true assembly language instructions. Instead, the contents of your template file will largely consist of .include and .equ statements.

Here is a typical template file for defining an amforth system running on an ATmega168 with a clock frequency of 8 MHz.

```
; This is a template for an amforth project.
; The amforth system generated works on atmegas
; with fuse factory default settings. At least
; for the ATmega16
;
; The order of the entries must not be changed since
; it is very important that the settings are in the
; right order
;
; first is to include the macros from the amforth
; directory
.include "macros.asm"

; amforth needs two essential parameters
; cpu clock in hertz, 1MHz is factory default
.equ cpu_frequency = 8000000

; size of return stack in bytes
.equ rstacksize = 80

; initial baud rate of terminal
.equ baud_rate = 38400

; the application specific dictionary can
;   - not be included, set to 0 (zero)
;   - be included in the rww section: set to 1 (one)
;   - be included in the nrww (bootsector) area: set to 2 (two)
; this  dictionary can be quite large so putting
; it into the bootsector area (NRWW) does make sense for the bigger
; atmegas only (ATmega32 and up) only
.set dict_appl=1



; include the amforth device definition file. These
; files include the *def.inc from atmel internally.
.include "devices/atmega168.asm"

; change these settings only if you know what you do.
  .set heap = ramstart
  .set VE_HEAD = $0000

;  Include the source for your turnkey application.

.include "applturnkey.asm"
```

```
; include the whole source tree.
.include "amforth.asm"
```

The following paragraphs describe the key elements of the above file.

Your template file must include the file macros.asm, and this should be the first statement (except for comments, of course) in your template file. The file macros.asm contains a set of macros specific to amforth, used to simplify the coding of the amforth words and underlying assembly language routines.

Next, your template file should declare two key equates required by amforth. The equate cpu_frequency defines the oscillator frequency, in Hertz, of your target hardware and is expressed as a long integer. The example shown above shows the target system uses an 8 MHz clock; cpu_frequency is declared as 8000000. The equate rstacksize declares the size, in bytes, of amforth's return stack. This stack normally doesn't need to be very large; the declaration above allots 80 bytes for the return stack.

Your template file should also declare the baud rate of your system's terminal, even if you don't actually intend to connect a terminal to your target. This value should be an integer giving the baud rate, in bits per second, of the serial connection. The example shown above defines a terminal baud rate of 38.4K baud.

Next, your template file should set dict_appl to one of three possible values; the value you choose determines where (or even if) your custom dictionary file, dict_appl.inc, is added when the application is built. If you set dict_appl to zero, the build process will not include dict_appl.inc. If you set dict_appl to one, the build process will add your dict_appl.inc file to the low flash-memory portion of the amforth dictionary. If you set dict_appl to two, the build process will add your dict_appl.inc file to the high flash-memory (bootloader) section of the amforth dictionary. Generally, the bootloader section of flash has very little available space; in almost all cases, you will add your application-specific words to the low flash-memory section (dict_appl set to one).

Next up is the device-specific amforth hardware definition file, found in the /devices directory. As shown above, the .include statement must contain the folder (devices/) and the name of the definitions file (in this case, atmega168.def). As shown, this template file builds an amforth that will run on an ATmega168 target.

The next two .set statements are pretty much required and should not be changed unless you are very knowledgeable with the inner design of amforth and your target system. Use the values shown in all but special cases.

The .org statement that appears next defines the start of the actual amforth system. This statement should be left unchanged.

If you have a turnkey application or if you want to use the default turnkey application, you should add

the include statement for applturnkey.asm. If you understand how to define amforth words in assembly language, you can build your own turnkey application by modifying a copy of the included applturnkey.asm file. If you don't need a turnkey application, leave this statement as shown; the default turnkey application is suitable for most uses of amforth.

This completes the custom portion of the template file. All that remains is the final include statement, which basically adds all of the words that make up the standard amforth system. You should spend some time looking through this file so you get an appreciation for how the inner amforth machine is constructed. However, you shouldn't modify any part of this file unless you know what you're doing. Everything you would need to do for the majority of amforth systems can be done with the statements available to you in your template file.