# amforth User's Manual
## (for amforth v4.2)

**Document contributed to the amforth project on SourceForge.net.**

## Revision History

| Version | Author | Notes |
|---|---|---|
| 4.2.0 | Karl Lunt | Extensive changes to track amforth v4.2. |
| 4.2.1 | Karl Lunt | Minor changes to introduction |

# Table of Contents

# 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 v2.3 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, (optionally) 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 version 4.2 or higher; 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 3.2.0. 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

## A quick look inside amforth

amforth v4.2 is available as a single downloadable file (amforth-4.2.zip). Download this file into your working folder (I'll use c:\projects\amforth-4.2 throughout this document) and unzip them, preserving the subdirectory structure. You should end up with the following subdirectory layout:

```
c:\projects\amforth-4.2\              <-- main directory, holds your project files
                       \appl          <-- holds amforth systems for various MCUs
                       \core          <-- holds source for amforth primitives
                       \doc           <-- holds amforth documentation
                       \examples      <-- holds small projects written in amforth
                       \lib           <-- holds libraries of Forth source files
                       \tools         <-- holds amforth support tools (pd2amforth)
```

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 and .eep files 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.

### *core\words subdirectory*

The core\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:
    .dw $ff01
    .db "=",0
    .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:
    brne PFA_ZERO1
    rjmp PFA_TRUE1
```

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.

### *core\devices subdirectory*

The core\devices subdirectory holds several folders, each defining a target MCU. For each target MCU defined, the associated folder holds four files..

The device.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 <target>.frt file contains Forth source defining MCU-specific parameters, such as IO register names and addresses. You can use this file when working in Forth on your target system; it is not needed when building an amforth system in AVRStudio4.

The device.inc file contains assembler source and is intended to be included as part of any amforth applications you build for the target device. This file creates Forth words in your dictionary that provide access to the MCU registers on your target.

(Comments on device.py file are needed.)

If you need to develop support files for a different Atmel MCU, you can copy the existing files for a 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.

(Add details on how the device.py file is generated.)

### *core\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_appl.inc),
- adds the words to be assembled into high flash memory (dict_appl_core..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). 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.

### *core\dict_core.inc*

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

```
; this part of the dictionay 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/doliteral.asm"
.include "words/dovariable.asm"
.include "words/doconstant.asm"
.include "words/douser.asm"
.include "words/fetch.asm"
.include "words/store.asm"
.include "words/cstore.asm"
.include "words/cfetch.asm"
```

Each line in dict_core.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_core.inc provide flash read and write capability, though other primitives, such as branch operations and the inner interpreter, may also be 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_core.inc unchanged.

### *core\dict_minimum.inc*

This file was called dict_low.inc in previous versions of amforth. Like dict_core.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 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.

## Developing your own amforth system

The following sections describe the files you need to create for building your own amforth system. In general, these are the only files you will need to edit during development. You can often find suitable files already in the distribution that can serve as starting points for these files; just copy and edit them to make your own files.

### *dict_appl.inc*

The file dict_appl.inc is created by you and contains those amforth words that you want to add to your application above and beyond the words added by dict_minimum.inc and dict_core.inc.

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.

Here is the dict_appl.inc file I created for my sample amforth system.

```
; this dictionary contains optional words
; they may be moved to the core dictionary if needed

.include "dict_minimum.inc"                    ; <-- required!
.include "words/fill.asm"
.include "words/d-2star.asm"
.include "words/d-plus.asm"
.include "words/d-minus.asm"
.include "words/d-invert.asm"
.include "words/udot.asm"
.include "words/dot-s.asm"

.include "words/dotstring.asm"
.include "words/squote.asm"

.include "words/words.asm"

.include "words/edefer.asm"
.include "words/rdefer.asm"
.include "words/is.asm"

.include "applturnkey.asm"
.include "words/int-store.asm"
.include "words/1ms.asm"
.include "words/ms.asm"

.include "dict_compiler.inc"
.include "words/show-wordlist.asm"
.include "devices/atmega328p/device.inc"       ; <-- hard-coded path!
.include "dict_usart.inc"
```

Note that the first statement in my file includes the dict_minimum.inc file. Note also that I have hard-coded the path to the appropriate device.inc file. If you build your application for a different MCU, you will need to adjust this path.

You are free to include whatever words from the words/ folder you want in your application. 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.

### *dict_appl_core.inc*

The file dict_appl.inc is created by you and contains those amforth words that you want to add to your application above and beyond the words added by dict_minimum.inc and dict_core.inc. The difference between this file and dict_appl.inc above is the words in this file will be written to high flash (the NRWW or bootloader section).

Because of the limited amount of space in the NRWW sections of most ATmega devices, you will want to limit the words in this file to those absolutely required to support amforth. In general, these are words that modify flash, such as istore.asm. Below is the dict_appl_core.inc file for my sample amforth system.

```
.include "dict_core.inc"                        ; <-- required!
.include "words/estore.asm"
.include "words/efetch.asm"

.include "words/istore.asm"
.include "words/istore_nrww.asm"
.include "words/ifetch.asm"
```

Note that my file starts by including the dict_core.inc file. You can add additional words in your dict_appl_core.inc, subject to space limitations in your MCU. If you accidentally include words already included by other segments of the amforth system, you will get assembler errors for the duplicates; just remove the duplicates from your dict_appl_core.inc file.

### *applturnkey.asm*

This file is created by you and 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. Here is the applturnkey.asm file for my sample amforth system.

```
; ( -- ) System
; R( -- )
; application specific turnkey action
VE_APPLTURNKEY:
    .dw $ff0b
    .db "applturnkey",0
    .dw VE_HEAD
    .set VE_HEAD = VE_APPLTURNKEY
XT_APPLTURNKEY:
    .dw DO_COLON
PFA_APPLTURNKEY:
    .dw XT_INITUSER
    .dw XT_USART
    .dw XT_INTON
    .dw XT_CR
    .dw XT_CR
    .dw XT_VER
    .dw XT_CR
    .dw XT_SLITERAL
    .dw 22
    .db "Karl's amForth system "
    .dw XT_ITYPE

    .dw XT_EXIT
```

The above simple application initializes the user area and USART0, displays amforth's version information on the serial terminal, displays an extra message proclaiming this as my amforth system, then exits to the main amforth shell.

Note that my custom message ends with a space. I did this so the message had an even number of letters. If the string length is odd, the assembler reports a warning that it had to add an zero-byte to pad out an even number of bytes. (I could have also used the ,0 technique shown in the declaration of the applturnkey string at the top of the file.)

### *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 the template file, myproj.asm, for defining my sample amforth system running on an Atmega328P MCU with a clock frequency of 16 MHz.

```
;
;  myproj.asm      a simple AVRStudio4 assembly-language project for amforth
;
;
; The order of the entries (esp the include order) 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"

; include the amforth device definition file. These
; files include the *def.inc from atmel internally.
.include "devices/atmega328p/device.asm"            ; <-- hard-coded path!

; amforth needs two essential parameters
; cpu clock in hertz, 1MHz is factory default
.equ F_CPU = 16000000

; terminal settings
.set WANT_ISR_RX = 1 ; interrupt driven receive
.set WANT_ISR_TX = 0 ; send slowly but with less code space

; initial baud rate of terminal
.include "drivers/usart_0.asm"
.equ BAUD = 38400

.if WANT_ISR_RX == 1
  .set USART_B_VALUE = (1<<TXEN0) | (1<<RXEN0)| (1<<RXCIE0)
.else
  .set USART_B_VALUE = (1<<TXEN0) | (1<<RXEN0)
.endif

.equ USART_C_VALUE = (3<<UCSZ00)

.equ TIBSIZE  = $64    ; ANS94 needs at least 80 characters per line
.equ APPUSERSIZE = 10  ; size of application specific user area in bytes

; addresses of various data segments
.set here = ramstart           ; start address of HERE, grows upward
.set rstackstart = RAMEND      ; start address of return stack, grows downward
.set stackstart  = RAMEND - 80 ; start address of data stack, grows downward
.equ amforth_interpreter = max_dict_addr ; the same value as NRWW_START_ADDR
; change only if you know what to you do
.equ NUMWORDLISTS = 8 ; number of word lists in the searh order, at least 8
```

```
.equ want_fun = 1 ; in case of an error out print an additional line with an caret
indicating the error position

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

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 include the device.asm file specific to your target hardware. This file is found in the core\devices\<target> folder that matches your target hardware. Note that in the above sample, the path to the device.asm file is hard-coded in the .include statement. If you are using a different MCU, you will need to edit this .include statement.

Next, your template file should declare a key equate required by amforth. The equate F_CPU 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 16 MHz clock; F_CPU is declared as 16000000.

Next, you need to set a couple of equates defining how your target hardware will use USART0. As shown, my amforth system will use receive interrupts for incoming characters ( `WANT_ISR_RX = 1)` but will poll when sending characters (`WANT_ISR_TX = 0)`. For your first system, leave these as defined. After you get more skilled in amforth, you can change these equates to tweak performance.

Next, your template file should include the assembly source file supporting the USART you intend to use as a console on your target hardware. The example above uses USART0 and includes the appropriate device include file. The equates BAUD, USART_B_VALUE, and USART_C_VALUE defines the characteristics of the USART that will ultimately serve as the console on your target hardware. You may need to alter these values in your own template file to match your target.

In general, the remaining equate values (.equ and .set) in the template file will be calculated automatically, based on previously defined values in the Atmel definition files for the various MCUs.

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. 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.

### *Setting up AVRStudio4*

Begin by creating a new assembly-language project in AVRStudio4. For my sample, I named this project myproj and created it in the c:\projects\amforth-4.2 folder.

I then created the four above files in my project folder (the folder holding myproj.aps).

Next, I added the path to the core\ folder to my project. To do this, click on **Project/Assembler Options** and locate the entry for **Additional include path**. Enter in this field the full path to the core\ folder; in my case, this path is C:\projects\amforth-4.2\core.

This completes setup in AVRStudio4. Pressing F7 assembles the source files and leaves behind two object files. The file myproj.hex contains the flash contents for my amforth system and the file myproj.eep holds the EEPROM values needed at startup.

I then hooked my programming pod (an AVRISP mkII) to the target board and applied power. I first made sure that the device's fuses were properly set. In particular, I checked that the bootloader area was set to the maximum (2048 words for the ATmega328P), the EESAVE fuse was checked, and the fuse for BOOTRST (jump to bootloader on reset) was NOT checked.

Finally, I downloaded the EEPROM file to the target, followed by downloading the flash file. When I reset the target, the Hyperterm window hooked to the target's serial port showed the expected amforth announcement. My amforth system was up and running!

### *What could go wrong?*

When you build a properly created amforth system, you should not get any errors or warnings. If the assembler complains about not finding included files, double-check the layout of your amforth folder; it should match what I've described at the beginning of this manual. Additionally, make sure you entered the path to your core\ folder correctly in the **Additional include path** box when you set up the project's properties.

If the project builds properly but does not provide characters to your serial port when you download your code to the target, double-check that your Hyperterm settings match those in your template file (myproj.asm above). Also confirm that you got the MCU frequency correct in your template file. Also double-check that you put the correct hard-coded paths in your template file and in your dict_appl.inc file. Also make sure you download BOTH the .hex and the .eep files for your project, and that you have the EESAVE fuse on your device checked.

### *An alternative AVRStudio4 setup*

The above method uses two hard-coded paths in your source files. This can cause problems if you later try to change processors; it's easy to forget to change one of the hard-coded paths and it can take time to track down the error.

As an alternative, you can go to **Project**/**Assembler Options** and remove the path you added in the **Additional include path** field. Edit the **Additional parameters** field to include the following text, all entered on a single line:

```
-I C:\projects\amforth-4.2\core -I C:\projects\amforth-4.2\core\devices\atmega328p
```

Note that the above entry is for my project; adjust as needed for your file paths and target MCU. Be sure to include the single space after each '-I' as shown!

With this change in place, you can now remove the hard-coded paths in your template file and your dict_appl.inc file. For example, in my myproj.asm template file above, I had the line:

```
.include "devices/atmega328p/device.asm"            ; <-- hard-coded path!
```

After adding the above change to the **Additional parameters** field, this line becomes:

```
.include "device.asm"
```

Make a similar change in dict_appl.inc and reassemble.