Introduction

If there is a list somewhere of obscure computer programming languages Forth is surely on it. Although Forth has had a place in the evolution of programming languages that continues to this day, many programmers never got to or never wanted to see what Forth was all about. Programmers who did experiment with Forth typically fell into one of two camps -- those that thought it was the best language ever and those who hated it completely. Rarely was any middle ground taken. Discussions of Forth create debates akin to discussions of what is the best programming editor.

I was an early convert to Forth after having worked at JPL on inter-planetary spacecraft where Forth was used for testing purposes, at Rolm Corporation on Forth-based PBX control and test applications, and at Sun Microsystems where Forth was used as the boot environment on many Sun workstations.

Then as now, Forth is an excellent choice for programming small computer systems where direct control of the hardware, of inputs and outputs (I/O) and/or a small memory footprint are required. In this article I hope to convenience you of this assertion and to pique your interest enough to try it out for yourself. And as luck would have it, the Arduino Uno is the perfect vehicle for trying Forth.

What Is Forth Anyway?

Forth is a stack-based procedural programming language invented by Charles H. Moore around 1968. Moore used Forth for controlling space telescopes at Kitt Peak Observatory in Arizona. At that time programming tools for small computer systems were rather crude and computer memory was very expensive. C compilers were not yet widely available, BASIC was still fully interpreted and therefore very slow, and Fortran compilers were very expensive. Programming during this period usually meant assembler language or worse yet machine language. Forth offered a higher level of programming that increased productivity while still offering low level access to the hardware if required by an application.

Forth source code is line-oriented and can be entered interactively or stored in text files and loaded in bulk. A line of Forth source consists of a series of tokens (words and numbers) separated by space/blank characters. The Forth interpreter reads in the tokens and either executes them interactively or, if in compile mode, compiles them for later execution.

The Forth *word* is a language element that can be used interactively or within a program. Source code for a particular word is called its "definition." Primitive Forth words have their definitions coded in the CPU's native assembler language, whereas non-primitive words have their definitions coded in Forth. A Forth program is run by executing the highest level word which defines the program.

The Forth word ":" (colon) puts the interpreter into compile mode and the word ";" (semicolon) ends compile mode. Immediate Forth words execute even in compile mode, whereas non-immediate Forth words have their addresses compiled into the *word* being defined.

Many things make Forth unique but two things need to be specifically pointed out.

1. Forth is a stack-oriented language, meaning that all arguments to and values returned from Forth words are unnamed, untyped, and generally located on a stack.
2. Forth uses Reverse Polish Notation (RPN) for expression evaluation like many HP calculators.

Let's use the Forth word DUP as an example. DUP duplicates the value currently on the top of the data stack. DUP takes its input value from the stack and pushes two copies of that value back onto the stack.

Using RPN for expression evaluation takes getting used to when you're used to normal algebraic or infix evaluation. The algebraic expression 2 + 5 * 3 evaluates as 2 + (5 * 3), whereas the RPN evaluation becomes 2 5 3 * +. Expression conversion to RPN is easy if you keep the following in mind:

1. Identifiers are used in the same order
2. Operators are used in the same order
3. Operators follow identifiers

Forth leaves all data typing issues to the programmer. Data typing is usually handled by having Forth words defined for each data type. For example, the standard addition operator "+" expects its two arguments to be integers and integers only. If floating-point addition is required in an application, for example, a special operator like "F+" might be defined.

Forth words use a stack diagram for documentation. DUP's stack diagram is a follows:

(n -- n n)

The value on the left after the paren is what is on the data stack before DUP is executed. In this case a number n and the two items before the closing paren are a picture of the stack after execution indicating two copies of n. It is important to understand that on both sides of the -- the top of the stack is always to the extreme right.

Versions of Forth are available for almost all microprocessors and micro-controllers. In the past Forth was generally one of the first programming language implemented for new microprocessors because it is easy to implement. Code up a handful of Forth primitives for I/O for the new microprocessor and the remainder of the language, written in Forth, just runs.

Arduino/Wiring Programming vs. Forth Programming

I thought it might be useful to compare and contrast these two development environments to gauge their strengths and weaknesses. For those that don't know, Wiring is the name of the computer language that runs on the Arduino hardware.

The biggest difference between Wiring and Forth is in how software is developed. With Wiring, code is written on an external computer, compiled and linked into an executable on the external computer and when thought error free downloaded to the Arduino hardware for execution and debugging. The serial

monitor built into the IDE along with print statements in the code are the means of debugging the code. When an error is detected, the offending code is found and fixed then the above process is repeated. In a large program this process is repeated over and over. Hosting the software development tools on the Arduino itself as a means of shortening the turn around time is impossible.

Forth is different in that all required software development tools are built directly into the language itself and are available all of the time on the target system. You still need an external computer to interact with Forth during program development but its only function is as a serial terminal and mass storage of the Forth source code. This difference may seem subtle but it is important.

In Wiring when you identify an error you correct it and then re-flash the complete software image back to the target hardware. With Forth you just recode the Forth word in error and re-flash it; a much faster alternative.

With Arduino code you write a program using the key words and functionality provided by the Wiring language and supporting libraries. A program in Forth is actually an extension of the language itself. Much of the functionality of the Wiring language is hidden from the user whereas in Forth nothing is inaccessible. Don't like how a language feature like a loop structure works in Wiring, you are out of luck. In Forth you can write the perfect loop structure and it becomes part of the language.

Don't get me wrong, the Arduino/Wiring combination is a winner and I use it regularly as frequent readers of Nuts and Volts will attest. It has introduced many people to programming that would otherwise have been kept away by the complexity of programming in C or C++ for example. Another Wiring benefit is the availability of third party libraries for controlling all types of devices. Properly written libraries shield the programmer from having to understand the complexity of the hardware they want to use in their programs. Unfortunately Forth doesn't have this kind of support.

AmForth

Arduino hardware is the perfect vehicle for trying out Forth in the embedded realm. AmForth is a version of Forth that can run on the Arduino Uno, Duemilanove, Leonardo, Sanguino, Mega and many similar variations. Code and documentation for AmForth can be found via the links specified in the Resources section. From this point forward when I say Arduino I mean the Arduino Uno.

AmForth must be flashed onto the Atmel Atmega328 micro-controller chip on the Arduino. Once flashed, all aspect of Wiring are erased. To make this an Arduino device again will require that the Arduino bootloader be re-flashed on to the device. Don't let this be a barrier to trying AmForth because the process to flash AmForth and/or to re-flash the Arduino bootloader is fast and easy. We will do this by using one Arduino Uno to program another as shown in Figure One.

Bring up the Arduino IDE, select the Arduino Uno board type and select the correct serial port connection for your computer. Load the *ArduinoISP* sketch and download it to the Arduino Uno which will serve as the programmer. Now make the connections shown in Figure One.

I use a makefile on my MacBook to drive the AmForth flashing process. This makefile assumes the existence of uno.hex and uno.eep, components of AmForth, which can be downloaded from the AmForth website. My makefile is shown below:

```
# Makefile for programming the Arduino Uno with AmForth
```

```
# Examples of usage:
# 1) Upload the new firmware hex and eep
#    make uno
# 2) Set the appropriate MCU fuses
#    make uno.fuse
SHELL=/bin/bash
##############################
# TARGET DEPENDANT VARIABLES #
##############################
# 1) MCU should be identical to the device
#    Look at the /core/devices/ folder
# 2) PART is the device model passed to avrdude.
# 3) LFUSE, HFUSE, EFUSE are the device-specific fuses
uno:              PART=m328p
uno.fuse:         PART=m328p
uno.fuse:         LFUSE=0xFF
uno.fuse:         HFUSE=0xD9
uno.fuse:         EFUSE=0x05
# ------------------------
# PROGRAMMER CONFIGURATION
# ------------------------
PORT=/dev/tty.usb*
PROGRAMMER=avrisp
BAUD=19200
AVRDUDE=avrdude
AVRDUDE_FLAGS= -P $(PORT) -c $(PROGRAMMER) -b $(BAUD)
# Flash the target
% : %.hex
        @echo "Uploading Hexfiles to Arduino $*"
        $(AVRDUDE) $(AVRDUDE_FLAGS) -p $(PART) -e -U flash:w:$*.hex:i -U eeprom:w:$*.eep:i


# Set the fuse bits
%.fuse :
        @echo "Setting fuses to Arduino $*"
        $(AVRDUDE) $(AVRDUDE_FLAGS) -p $(PART) -U efuse:w:$(EFUSE):m -U hfuse:w:$(HFUSE):m -U lfuse:w:$
(LFUSE):m
```

Programming instructions for OSX, Windows and Linux are available in the AmForth documentation available online. If you use this makefile, make sure the PORT and BAUD entries are set appropriately.

If everything is setup correctly you would type:

make uno

to flash AmForth to the Arduino hardware and then

make uno.fuse

to program the fuses in the Atmega328 as required for AmForth's operation. If no errors are reported, your Arduino Uno has now become a *Forthduino*.

You need a serial terminal program to talk to AmForth. *Screen* can be used on the MAC. Many serial terminal programs are available for Windows and Linux. For serious AmForth development a custom terminal program written in Python called amforth-shell.py should be used. Discussion of this tool is beyond the scope of this article but trust me, you will want to use this tool if you do any serious work with AmForth. See the AmForth documentation.

Assuming you have your serial terminal program connected to your Forthduino and you cycle the power to the board or press the reset button you should see something similar to the following:

```
amforth 5.1 ATmega328P Forthduino
Apr 05 2013 20:11:48 ## master...uni/master [ahead 2]
>
```

The “>” is a prompt letting you know that AmForth is awaiting your command. Let's verify things are working by running a few examples. The “>” prompt and “ok” are provided by AmForth. All tokens must be separated by space characters.

```
> 1 . <enter>   NOTE: <enter> refers to the enter or return key
1 ok
```

Since we are not in compile mode, the first 1 is identified as a number and is pushed onto the data stack. The "." (called "dot") causes the item at the top of the stack to be popped off and printed to the console; hence the second 1.

```
> 10 hex . <enter>
A ok
```

This example performs number base conversion. Unless changed, Forth uses base 10 for numeric operations. Here we push a decimal 10 value onto the stack, change the numeric base to hex (base 16) and then use dot to print the top of the stack value. An "A" is printed as A is 10 in hex.

```
> : message ." *** Hello World ***" cr cr  ; <enter>
ok
```

This is an example of Forth compilation. Here we define a new *word* called message which when executed writes the string *** Hello World *** to the console followed by a couple of carriage returns. The ok signifies successful compilation. Later when message is typed on the command line, the string

will be output as follows:

```
> message <enter>
*** Hello World ***


ok
```

As a final example, let's compile a do loop Forth word and see how it works. We will name our do loop example *a_do_loop* and define it as follows

```
: a_do_loop 10 0 do i . loop ; <enter>
 ok
```

When subsequently executed by typing *a_do_loop* at the command prompt, the output is as follows:

```
> a_do_loop <enter>
0 1 2 3 4 5 6 7 8 9 ok
```

Of course these are trivial examples one cannot get too excited about. For a more complete example I rewrote the Arduino Blink example in AmForth. The original .ino file is shown first:

```
/*
  Blink
  Turns on an LED on for one second, then off for one second, repeatedly.

  This example code is in the public domain.
 */

// Pin 13 has an LED connected on most Arduino boards.
// Pin 11 has the LED on Teensy 2.0
// Pin 6  has the LED on Teensy++ 2.0
// Pin 13 has the LED on Teensy 3.0
// give it a name:
int led = 13;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);               // wait for a second
}
```

The port to AmForth results in:

```
\ Some of this example code is extracted from the file bitnames.frt written
\ by Matthias Trute and M. Kalus

\ Create a named bitmask in the dictionary which when executed
```

```
\ places the port address and pin bitmask on the stack
: bitmask: create ( C: "ccc" portadr n -- ) ( R: -- pinmask portadr )
    , ,
  does>
    dup @i swap 1+ @i
;

\ Define a portpin with specified name
: portpin: ( C: "ccc" portadr n -- ) ( R: -- pinmask portadr )
    1 swap lshift \ make it a bitmask
    bitmask:
;

\ Set DDRx so its corresponding pin is output.
: pin_output ( pinmask portadr -- )
    1- high
;

\ Turn a port pin on, dont change the others.
: high ( pinmask portadr -- )
    dup  ( -- pinmask portadr portadr )
    c@   ( -- pinmask portadr value )
    rot  ( -- portadr value pinmask )
    or   ( -- portadr new-value)
    swap ( -- new-value portadr)
    c!
;

\ Turn a port pin off, dont change the others.
: low ( pinmask portadr -- )
    dup  ( -- pinmask portadr portadr )
    c@   ( -- pinmask portadr value )
    rot  ( -- portadr value pinmask )
    invert and ( -- portadr new-value)
    swap ( -- new-value port)
    c!
;

\ Define the PORTB data port where the Arduino's LED is connected to bit 5
&37 constant PORTB

\ Create a named definition for this port/pin called LED_PIN
PORTB 5 portpin: LED_PIN

\ Run the blink method at the command prompt by typing blink

: blink

    \ Declare the LED_PIN a digital output
    LED_PIN pin_output

    begin
        LED_PIN high      \ Turn the pin on
        1000 ms           \ Wait 1000 ms or 1 second
        LED_PIN low       \ Turn the pin off
        1000 ms           \ Wait 1000 ms or 1 second
        0                 \ Zero here indicates false so the loop repeats indefinitely
    until
;
```

While the AmForth code may seem much more complex it really isn't. We had to instruct AmForth how to deal with the Atmega328 port hardware. If applications you write requires this functionality, you would include the library file *bitnames.frt* once and from then on AmForth would know how to manipulate ports and bits. NOTE: AmForth source files all use the ".frt" file extension.

If you want to get your Arduino back for use with Wiring, again connect the two Arduinos as shown in Figure One, bring up the Arduino IDE and select *Burn Bootloader* from the *Tools* menu. After a couple of minutes you Arduino Uno will be as good as new.

Conclusions

Forth is a big topic for a small article but I hope you got a feel for what Forth is. Forth looks strange to the untrained eye but after working with it, things become very natural and you can achieve high productivity in your programming efforts. Having all of the development tools built into the language can make turn around times on changes very fast. If you would like to see a full application coded in AmForth you should check out the DigitalFMReceiver code for the article I wrote previously for Nuts and Volts. Details below.

Resources

The source of all things AmForth is here: http://amforth.sourceforge.net/. Check out the user's guide and the technical guide.

AmForth source code for the Arduino FM Radio article I published in the XXXX issue of Nuts and Volts can be found in a zip file on the Nuts and Volts website for this article.

Figure One
Using an Arduino Uno to Program Another with AmForth