Back to Forth: Faster Testing and Development for Embedded Systems

Andrew Sterian and Aron J. VanderMeyden
Padnos School of Engineering
Grand Valley State University

Introduction

Embedded system development requires at minimum hardware design, firmware design, and perhaps the design of external software for testing or automation. This development process is often frustrated by trying to debug three unknowns simultaneously: the hardware, the firmware, and the external software. Debugging these three unknowns is much more complex than debugging either hardware or software in isolation. Powerful but expensive support tools, such as in-circuit debuggers, can be used to aid in the process, but these are often impractical or unavailable in undergraduate engineering education laboratories. These tools also have a steep learning curve and require experience and patience to be used effectively.

For embedded systems with limited memory resources, the C programming language is commonly used to develop firmware as this language is a good compromise between expressive power and low-level control. While trying to debug the initial system prototypes, however, writing firmware in C requires an investment in the edit-compile-link-download cycle (Figure 1), since C is a compiled language. For each problem encountered during testing, the C program must be edited and compiled. Compilation errors are common and must be addressed. Link errors may also occur due to missing library functions, misspelled function names, and so on. Finally, the compiled program must be downloaded to the target system, usually over a serial link, or programmed into the FLASH or other nonvolatile memory of the target system.

While this process may only take a few minutes, even this small amount of delay interrupts the flow of the testing and debugging process, leading to reduced productivity. The situation is made worse when cables must be swapped for downloading rather than interacting with the target system, or IC's must be removed from sockets for programming, and so on.

An alternative to working with a compiled language at this phase of development is to write a *monitor* program. The monitor program acts as a conduit between the user and the underlying hardware, providing low-level access to registers and I/O ports. The user interacts directly with the monitor (Figure 2) running on the target system, commonly using a serial link and terminal emulator program. However, the monitor program, usually written in C, has fixed functionality. When problems are discovered that cannot be easily debugged with the monitor (perhaps involving tight timing interactions), the monitor must grow in complexity. Then, the edit-compile-link-download cycle is applied to the monitor program and once again slows down the testing process. More and more time is spent writing and debugging the monitor itself.

Copyright © 2004, American Society for Engineering Education

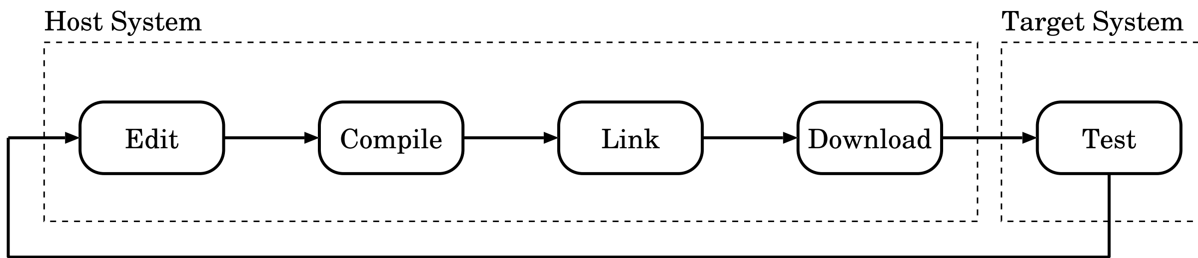

Figure 1: The edit-compile-link-download cycle for embedded system development in C, or other compiled languages. Development is slowed down by potential errors at each stage and by the time it takes to download and/or program the target hardware.

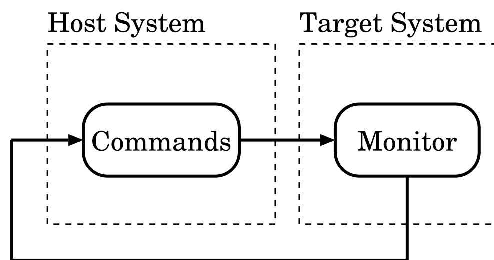

Figure 2: A monitor program running on the target system allows the user to interact directly with the hardware. However, the monitor program has fixed functionality and must be updated (with the edit-compile-link-download cycle) to implement new features.

In our undergraduate courses, we have used both modes of development described above. The C-based approach has high overhead, thus low productivity. The monitor-based approach allows for interactive experimentation and testing, thus higher productivity, until the limits of the monitor are reached. For example, in our Introduction to Digital Systems course[1], a sophomore-level course for all engineering students, we use 68HC11-based embedded systems running the BUFFALO monitor in ROM. Our development process uses the GCC C compiler targeted for the 68HC11. Students are able to easily modify memory and registers using BUFFALO, but as soon as they begin writing programs in C to perform complex tasks, they suffer an immediate drop in productivity.

We have used a similar 68HC11-based system for our development of a firefighting robot. This project requires a fair amount of experimentation and “tweaking”, processes not well suited to development in C. For example, the latest version of the software for the 68HC11-based robot has over 200 configuration options declared in a C array, and keystroke-menu commands for modifying each one during interactive execution. But there are always new parameters to adjust, and new algorithms to try. These additions require a return to the edit-compile-link-download cycle. Moreover, the data and code required to support the interactive modification of these options form a large part of the program itself, increasing the time it takes to download each new iteration of the code to the robot.

Copyright © 2004, American Society for Engineering Education

A better development environment would retain the programmability and expressive power of C with the interactive nature of a monitor. Ideally, an “interpreted C” monitor would allow for both off-line program development and interactive testing and tweaking. With limited memory resources, however (our 68HC11-based systems have only 32K of RAM), a C interpreter is too large for this task.

In this paper, we propose that the Forth programming language be considered as a solution to this problem. Forth can act as a replacement for both the monitor program and for large parts of the final system software. Thus, Forth can be seen as a “programmable monitor” or as an “interactive language.” This well-established language can act as a monitor due to its interactive execution and support for direct hardware manipulation. The language allows for more complex monitor tasks to be written incrementally, as the testing process unfolds, and without the edit-compile-link-download cycle of C programs.

We present a brief history and introduction to Forth then describe two new implementations of Forth for the Motorola 68HC11 and Atmel AVR processors. Finally, we provide specific examples of how the AVR Forth implementation was used in developing the embedded control system for our latest firefighting robot. We compare the process of embedded software testing and development in Forth with the same process, using C, for an earlier version of the robot.

The Forth Language

Forth was conceived in the 1970's by Chuck Moore as a small, efficient programming language to compete with the complicated yet primitive languages available on minicomputers at the time[2]. The language rose quickly in popularity but has since been overshadowed by more mainstream languages. Forth continues to play an important role in embedded systems where programmability and small code size are important. For example, the Forth-based OpenFirmware specification (IEEE Standard 1275) provides a common interface to the firmware of many hardware systems, such as Sun SPARCStations and Apple PowerMac systems. In industrial applications, Forth has been used to control a NASA simulator for the space shuttle arm, and to control a hand-held package tracker for Federal Express, among many other applications[3].

Over the years, Forth has been customized for many different environments, leading to a fragmented language with incompatibilities from one implementation to another. In 1994, an ANSI standard for Forth was accepted[4], informally known as ANS94. Among other things, this standard defined various *word sets*, which are Forth words grouped together by functionality. The *core* word set must be implemented by all Forth systems, while all other word sets were deemed optional.

There are five main components in a Forth system:

- The kernel is the core code of the Forth interpreter, usually written in C or assembly language.
- The stack is the main data structure used by Forth. This is a software data structure, not to be confused with the hardware stack of the underlying processor.
- The return stack is a companion to the regular stack.
- The dictionary stores just the names of all functions (called *words*), variables, and other named objects.
- The code space stores the definitions of all functions and values of all variables.

The kernel of a Forth system can usually be very small. When written in assembly, it often

Copyright © 2004, American Society for Engineering Education

occupies just a few kilobytes of memory. The two stacks are usually just a few hundred bytes of memory. The sizes of the dictionary and code space define how many words and variables can be added to the core system. These two areas will generally occupy a few kilobytes for a minimal system, more for a system with greater functionality. Thus, an entire Forth system can operate with very limited memory.

The Forth code below adds the numbers 2 and 3 then displays them on the screen:

2 3 + .

The Forth kernel performs the following operations when interpreting this code:
- The number 2 is pushed on the stack
- The number 3 is pushed on the stack
- The word '**+**' is looked up in the dictionary. It is found, and the word instructs the kernel to perform the addition operation on the top two items on the stack. These items are popped from the stack, and the sum is left as the top item on the stack.
- The word '**.**' is looked up in the dictionary. It is found, and the word points to more Forth code in the code space. The interpreter then begins executing this code, which has the effect of popping the top element from the stack and displaying it to the screen.

This simple example highlights several important aspects of Forth:

- Forth is a stack-based language. Most words in Forth will perform some transformation on the stack. Specifically, the Forth programmer must become accustomed to "reverse Polish notation" for arithmetic.
- Everything that is not recognized as a number is a word that must exist in the dictionary.
- Words and numbers are separated by whitespace.
- Some Forth words (called *primitives*) trigger actions in the kernel (like '**+**') while others are like subroutine calls in that they invoke further Forth code in the code space (like '**.**').

The size of the kernel can be reduced by implementing as much functionality as possible in Forth. As kernel code will be fastest, however, there is a tradeoff between kernel size and system performance.

New words in Forth can be defined as in the following example:

: negate 0 SWAP - ;

The colon word '**:**' begins a new word definition. The new word '**negate**' is added to the dictionary. All of the words that follow are added to the definition of '**negate**' (in the code space) until the semicolon word terminates the definition.

This new word has the effect of computing the negative of the topmost number on the stack (known as the *top-of-stack*, or *TOS*) and replacing the TOS with this negative. The definition of '**negate**' pushes 0 on the stack, swaps the two top elements on the stack, then subtracts them (thus computing 0-TOS, or -TOS) and leaves the result on the stack.

The fact that the overhead for making a function is so low makes it much easier to create a small set of simple tools that can be then be tested easily and then reused to make other

Copyright © 2004, American Society for Engineering Education

higher complexity tools.

For example, this new **NEGATE** word can then be used to define other words, such as an absolute value function:

```
: abs ( X -- |X| )
   DUP            \ -- X X
   0<             \ -- X flagIfX<0
   IF             \ -- X
      NEGATE      \ -- -X
   THEN           \ -- X | -X
;
```

Forth is non-case-sensitive, hence '**negate**' is equivalent to '**NEGATE**'. Comments are set off by parentheses, or a \ character which then comments out the rest of the line. It is customary to document the overall effect of a new Forth word by describing its effects on the stack. In the comment "**(X -- |X|)**" we indicate that prior to executing this word, a number X is on the top of the stack. After this word executes, the value |X| is left on the top of the stack and X is consumed. The documentation on each line is added to help understand the stack effects of each word (although a Forth expert would consider our documentation for **ABS** to be overly verbose and unnecessary).

The definition of the **ABS** word illustrates the **DUP** Forth word, which duplicates the TOS, and the **0<** word, which replaces the TOS with a flag that is true if the TOS was less than 0, and false otherwise. The **IF...THEN** construct (which should be interpreted as IF...ENDIF) then examines this flag and executes the **NEGATE** word if the flag was true (the flag on the stack is consumed by **IF**).

Constants can be defined to represent special memory locations or registers:

```
hex
1000 constant PORTA
```

The word '**hex**' indicates that all subsequent numbers are to be interpreted in hexadecimal. The '**constant**' word parses the next word encountered (in this case, '**PORTA**') and adds it to the dictionary with the TOS as its value. Thus, the above code defines '**PORTA**' to have the value 0x1000, which is the location of the Port A register on a 68HC11 microcontroller.

The following code stores hexadecimal 80 to Port A (using the word '**C!**' which stores an 8-bit number at a memory location):

```
80 PORTA C!
```

On a 68HC11, the above code would have the effect of setting Port A bit 7 to 1. We could define a new word that sets this bit to 1 while leaving other bits unchanged:

```
: PA7ON PORTA C@ 80 OR PORTA C! ;
```

The above code defines a new word '**PA7ON**' which reads the 8-bit contents of Port A (using the '**C@**' word), bitwise-OR's this value with hexadecimal 80, then writes this result back to Port A. This code can be immediately tested simply by typing **PA7ON** at the Forth prompt.

Copyright © 2004, American Society for Engineering Education

The code to turn the bit off is equally simple:

```
: PA7OFF PORTA C@ 7F AND PORTA C! ;
```

Here is a new word, using the words defined above, to pulse Port A bit 7 on and off 10 times:

```
: PA7PULSE
        10 0 DO
                PA7ON
                PA7OFF
        LOOP
;
```

From this brief introduction to Forth, it is clear that Forth can act as a monitor by providing interactive low-level access to system resources. But Forth is also easily programmable, even in an interactive session, to allow for more complex tasks to be defined. These tasks can be combined to form even more complex tasks, as with any structured programming language.

Forth Implementations

There are three main approaches to implementing a Forth development system, illustrated in Figure 3. In a native code Forth, a compiler running on the host system translates Forth code into the native machine code of the target system, which must then be downloaded. This development approach is identical to the edit-compile-link-download cycle of C, simply in a different language. This approach is not desirable from the point of view of testing and debugging an embedded system as it suffers from the same drawbacks as C-based development. Native machine code, however, will provide higher performance than interpreted Forth code.

In the second approach, Forth code is interpreted on the host system and interacts with a small monitor on the target system. This *umbilical Forth* implementation requires very few resources on the target system, relying on the host system for the majority of the work. The drawbacks of the umbilical Forth approach are the inability to run untethered and lower performance than a self-contained system, due to the need for frequent communication to the host system.

In the third approach, Forth is fully contained on the target system. This approach requires the most memory on the target system, but has no dependencies on the host and can run untethered. The self-contained Forth approach is the one that we will focus on in this paper.

In a self-contained Forth, testing and debugging can proceed very quickly because of Forth's dual nature as an interactive language and also as a structured programming language. Once a set of Forth words are defined and deemed useful, they can be placed in a file and uploaded to the target system before each debugging session. The debugging session can then proceed interactively. If desired, the new Forth words can even be added to the Forth system by recompiling the Forth implementation then reprogramming the target system with the new code.

Thus, there are three levels of code development available:

- Interactive exploration, in which new ideas and code are developed on-the-fly. New

Copyright © 2004, American Society for Engineering Education

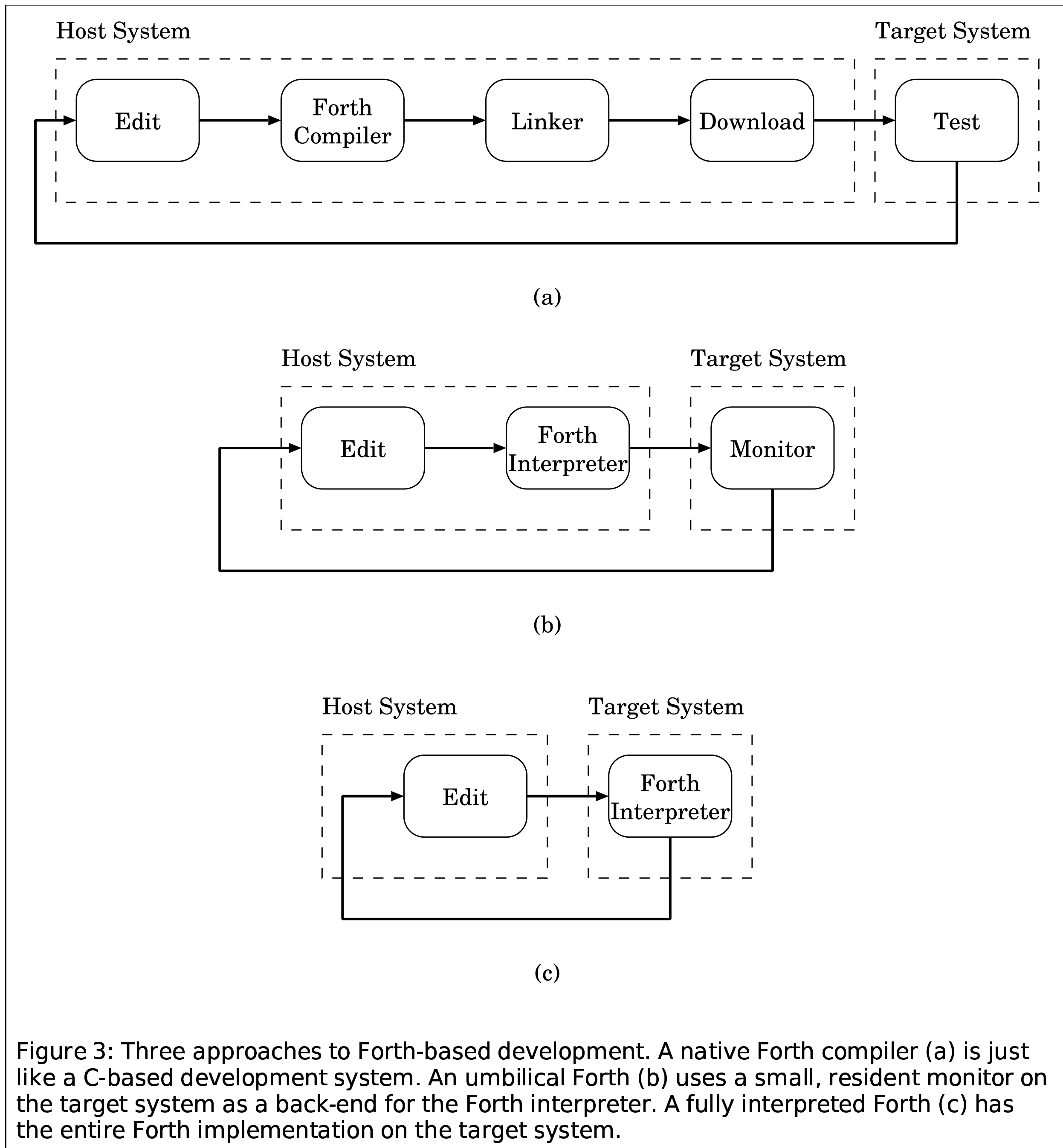

Figure 3: Three approaches to Forth-based development. A native Forth compiler (a) is just like a C-based development system. An umbilical Forth (b) uses a small, resident monitor on the target system as a back-end for the Forth interpreter. A fully interpreted Forth (c) has the entire Forth implementation on the target system.

Forth words can be immediately defined and tested quickly.

- New Forth words that have proven useful can be saved in a file and uploaded at the beginning of a new interactive debugging session, and after every target system reset.
- Forth words that have withstood the test of time can be incorporated into the Forth kernel or word sets and programmed into the target system, so that they are always available.

Copyright © 2004, American Society for Engineering Education

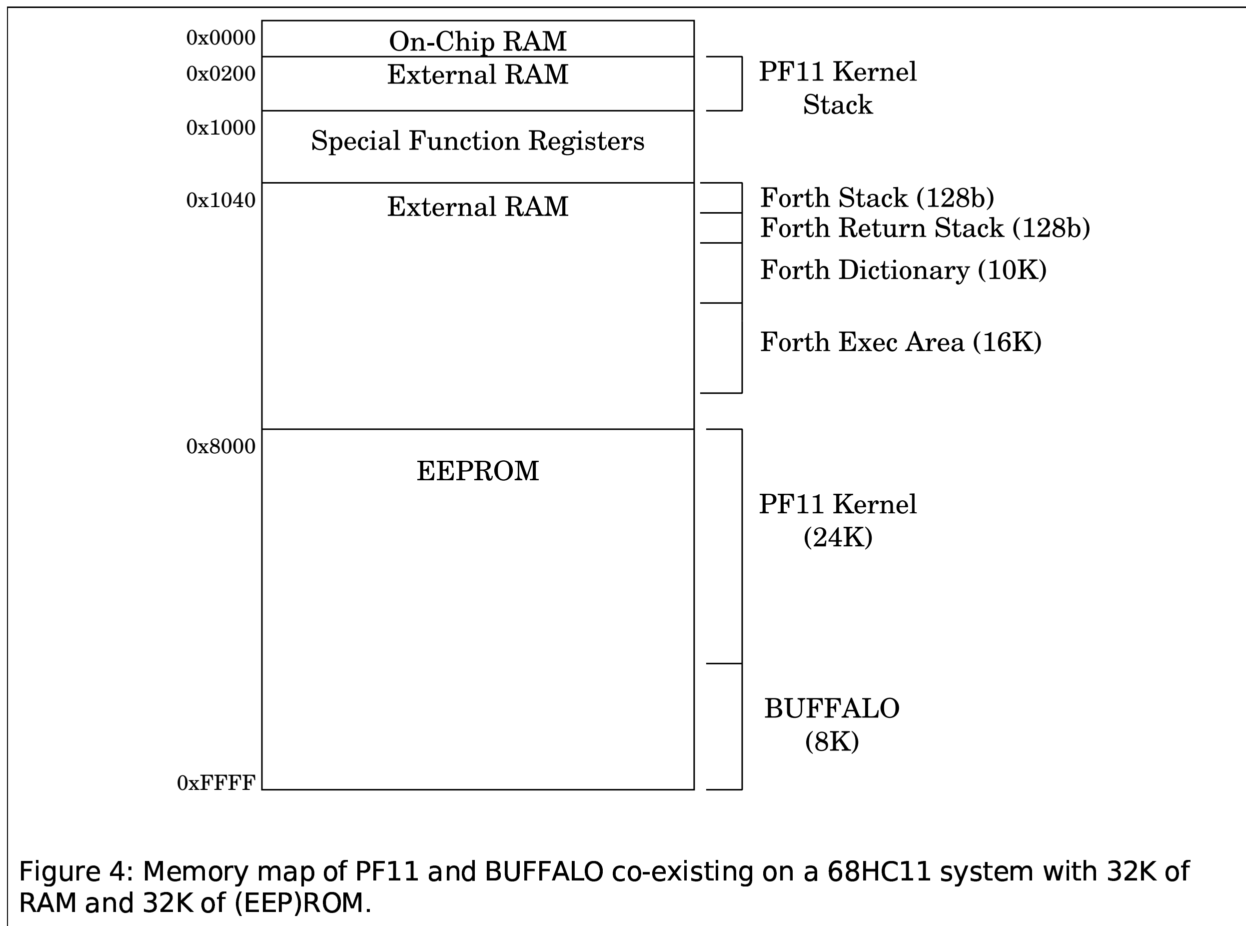

Figure 4: Memory map of PF11 and BUFFALO co-existing on a 68HC11 system with 32K of RAM and 32K of (EEP)ROM.

The engineer is free to work at whatever level of development is most appropriate for the task at hand.

Forth for the 68HC11

There are some existing, free, self-contained Forth implementations for the 68HC11[5]. These are written in assembly language, thus are difficult to modify or extend, and do not implement the ANS94 Forth standard. The goal of implementing another Forth for the 68HC11 was to remedy these drawbacks. The result, named PF11 (available from the first author's web site), has the following attributes:

- It implements the ANS94 core, extended core, double-number, exception, programming tools, and string word sets.
- It is written entirely in C for ease of modification and extension
- It fits into less than 24K of ROM and can operate with less than 28K of RAM
- It is free software licensed under the GNU General Public License (GPL)

The memory requirements of PF11 are higher than the assembly-language-based Forths for the 68HC11, but there is still enough room left on a typical, modern 68HC11 system for additional functionality.

PF11 can also be combined with BUFFALO on a target system, thus easing the transition from

Copyright © 2004, American Society for Engineering Education

existing development environments. Figure 4 shows the memory map of a 68HC11 system with 32K of (EEP)ROM and 32K of RAM in which both BUFFALO and PF11 are available (though not simultaneously, of course).

Forth for the AVR

Since PF11 was written in C, it was easy to modify for the Atmel AVR architecture. This implementation is known as PFAVR (also available from the first author's web site). It has the same attributes as PF11 and approximately the same memory requirements. Due to the need for more RAM than is contained on AVR devices, only AVR devices with an external RAM interface (currently the ATmega64 and ATmega128) are supported. Unlike the 68HC11, there are currently no free Forth alternatives for the AVR.

With the separate program and data spaces of the AVR processors, there is plenty of space available in the FLASH program space of an ATmega64 or ATmega128 for extending PFAVR with custom C code. The amount of required external RAM depends upon how much space is allotted to the Forth dictionary and code space. A good starting point is to use 32K of RAM, which is nearly completely filled by an ample Forth dictionary and code space. These contain the entire Forth system and have lots of room for user code.

Forth vs. C

For our latest firefighting robot design, currently in development, we switched from a 68HC11-based controller board to one based on the ATmega128 AVR. The controller board is connected to several daughterboards that perform sensing and actuator functions. The communication between the controller board and the daughterboards is implemented with an I^2C bus. The controller board always acts as the I^2C master. All of the daughterboards use a PIC16F818 microcontroller.

Our development efforts proceeded as follows:

- Verification of the controller hardware
- Development of I^2C driver functions for the controller
- Verification of each daughterboard through I^2C commands

This incremental process was well-suited to using Forth. To verify the controller hardware, we programmed PFAVR into the ATmega128. We were then able to interactively manipulate all of the on-chip registers of the ATmega128, perform memory tests, and so on.

As an example of how the interactivity of Forth allows for rapid integration, we describe the development of an interface with the daughterboards over the I^2C bus. We were able to implement the functions necessary to initialize the bus and then immediately test these functions. When we knew that the bus was properly initialized, we moved to testing the bus. This was difficult because we were also testing the code on the answering PIC16F818, which was written in Microchip assembly language. This was a classic case of debugging three things at once: our I^2C drivers in Forth, the hardware, and the I^2C drivers on the PIC16F818. This task would have been nearly impossible without the power of Forth.

Forth allowed us to make words for each step of the communication process. For example, the first word that was implemented was a start function. This begins the communication sequence on the I^2C bus and was implemented as follows:

Copyright © 2004, American Society for Engineering Education

```
: TWI_START  ( -- )
      E4 TWCR C!
;
```

(On the AVR platform the I^2C bus is referred to as the Two Wire Interface, which is why it is called TWI_START.) This word stores the hexadecimal value E4 into the TWCR register. This register is a native register on the AVR that controls the I^2C bus. The value 0xE4 clears the interrupt flag, turns off interrupts, and tells the hardware to try to start the communication sequence. Another word was then implemented that polls for the interrupt flag from the hardware that tells us the command has been completed:

```
: TWI_WAIT ( -- )
        BEGIN          \ Loop until bit 7 of TWCR is set
                TWCR C@ 80 AND
        UNTIL
;
```

During testing it was useful to be able to check the status of the I^2C hardware, so a word was implemented as follows:

```
: STA ( -- )
      ( print out twi status and mask off TWPS bits )
      TWSR C@ FC AND .
;
```

Unit testing each word was very simple since we didn't have to go back to a C compiler to write code to test each function. The ability to casually write and then forget functions at will encourages modularity.

When each step in the communication sequence was implemented with a word and unit tested, the next abstraction level up, sending a value to the PIC16F818, could be implemented as follows:

```
: SendToSlave  ( data address -- )
      TWI_START
      TWDR C! \ consumes address value

      TWI_WAIT
      TWI_SEND_ACK ( sending address of slave )
      TWI_WAIT
      TWDR C! \ consumes data value

      TWI_WAIT
      TWI_SEND_ACK ( sending data )
      TWI_WAIT
      TWI_STOP
      STA
;
```

This word starts the communication process, stores the address that was passed to it on the stack, and then waits for the hardware to signal it is ready for the next step. It then sends the address and waits for that to be sent, stores the data value, sends it, and then stops the

Copyright © 2004, American Society for Engineering Education

sending process. The word ends by showing the user the status of the hardware.

We were then able to work with the I^2C bus on several levels: using the full **SendToSlave** function to test the overall process or with each individual word (**TWI_START**, **TWI_WAIT**, etc.) to watch the communication process unfold one step at a time. By having the power to work at both the low and high levels, we were able to quickly debug the Forth code, hardware, and PIC16F818 drivers. By far, the most difficult task was debugging the PIC16F818 code as it was difficult to instrument this code and difficult to modify, since recompilation and FLASH programming was required for any change to the code. In contrast, the debugging of the Forth TWI drivers was extremely simple.

Conclusion

The Forth programming language's greatest strength lies in the ease with which words are implemented. All that is necessary to make a Forth word is to put together a list of commands and then put a colon and a name before and a semicolon afterward. Once stack manipulation is mastered, the language itself is very simple. The hardest part is learning the language of the problem to be solved. Once this language is learned, the code is practically written. Since the code can be written in the language of the problem, re-reading the code actually reminds the developer of the problem language, and if the code is carefully written, it can be self-documenting.

The interactivity, extensibility, and modularity of Forth allows for ease of unit testing and incremental development which shortens development cycles. The strengths of Forth allow the developer to put aside debugging their process and allows them to concentrate on debugging the problem. We were able to apply these benefits to the development of an embedded system, in which we observed significant improvements in the speed of development, testing, and debugging when compared to a C-based approach.

The availability of new, free Forth implementations for the popular Motorola 68HC11 and Atmel AVR microcontrollers, as described in this paper, allows students to quickly become familiar with and develop software for embedded systems.

Bibliography

1. A. J. Blauch and A. Sterian (2002). "A Practical Application Digital Systems Course for All Engineering Majors," *Proc. 2002 ASEE Annual Conf.*, Montreal, Canada.
2. E.D. Rather, D.R. Colburn, C.H. Moore (1993). "The Evolution of Forth," *ACM SIGPLAN Conf. on History of Programming Languages*, Cambridge, MA.
3. Forth, Inc. <http://www.forth.com/Content/Products/CaseHist.htm> Case histories of Forth applications.
4. NCITS Technical Committee J14. <ftp://ftp.uu.net/vendor/minerva/x3j14/j14-94.htm> Last ANS Forth 94 draft document.
5. A. Sterian. <http://claymore.engineer.gvsu.edu/~steriana/Software/pf11/rationale.html> List of free Forth implementations for the Motorola 68HC11.

Copyright © 2004, American Society for Engineering Education

Biographical Information

ANDREW STERIAN is an Assistant Professor in the Padnos School of Engineering at Grand Valley State University. He obtained the B.A.Sc. degree from the University of Waterloo and the M.S.E. and Ph.D. degrees from the University of Michigan. He teaches courses in digital systems, microcontrollers, and signal processing. His interests include embedded hardware and software design and robotics.
<http://claymore.engineer.gvsu.edu/~steriana>

ARON J. VANDERMEYDEN is a senior in the Padnos School of Engineering at Grand Valley State University. He is a member of Dr. Sterian's ground-breaking mechatronics course and has been the primary developer of the embedded software development for the robot.

Copyright © 2004, American Society for Engineering Education