Tom Napier's Forth Article

Circuit Cellar May 1998

Commented by Valter Foresto – 20060712 – Comm.SEC

Used as a Forth Primer to introduce the SX-Forth Interpreter

BLUE: I agree and I think it's true and important
GREEN: I added a comment to describe 'SX-Forth Interpreter'
RED: I disagree (... not used yet)

Just as at one time, no one got fired for buying from IBM, now no one gets fired for programming embedded applications in C. If an alternative is considered, it's usually assembly, although fashion is now swinging towards Java. **Very few programmers use Forth, a language that combines the speed, versatility, and compactness of assembly language with the structure and the legibility of C. These few have found Forth to increase programming productivity.**

In this article, we'd like to (re)introduce you to Forth. You'll be surprised how quickly and interactively you can write and test embedded programs without using complex tools.

The first step in writing any program is to work out in detail what it has to do. Some people draw flowcharts, while others use a Program Design Language (PDL) that describes the sequence of operations and test conditions in an English-like manner. Once this is done, the design is broken up into modules, and each is converted into executable code. The whole thing is compiled, linked, and tested, an iterative process that can take months.

However, if the PDL could be executed and you didn't need to translate it into another language, think how much time you could save. Wouldn't it be more convenient if you could test each program module interactively to verify its operation? Suppose too that there's a language that executes as quickly as any other language, has a run-time overhead of a thousand bytes, conforms to an ANSI standard, and can be extended to cope with any special requirements your application has. What if, after a week or two of familiarization, you could turn out at least three times as much finished code in a day as your fellow programmers? Would you be interested? If so, listen up to hear how you can do all this with Forth.

What is Forth?

In a sense, **Forth is not a language but rather a programming methodology for writing an application language for the task in hand**. You write the bulk of your program in terms of the job's requirements, not the compiler's edicts. Forth supports whatever operations and syntax you need.

Forth understands a range of primitive words that handle all the normal arithmetical, logical, and program flow operations. However, it also has a defined way of adding words to the language. You can decide what words best describe your application. You then define these words in terms of existing words. Once you've defined a word, it becomes part of the language and can be used to define other words. The highest level word is the program itself.

In Forth everything is either a word or a number or a string. Both are separated by spaces. There is no parsing in Forth and very little syntax. There are no operators, no functions, no procedures, no subroutines, not even programs—just words and numbers.

Every word tells the computer to execute a clearly defined finished operation. After you define a word, you can test it as a stand-alone element. You don't need to complete the program before you start testing. You can type any word at the keyboard, have it execute, and see if the result is what you expected.

Forth is its own symbolic debugger, so testing a Forth program is much faster than testing a program in another language. You write Forth incrementally, defining and testing words in turn. Once you're sure a word works, you can incorporate it into your program. Once you've defined the highest level word, you have a finished program that should need no further debugging.

Although a Forth program is normally designed from the top level down, you write it from the bottom level up—it expects you to define words before you use them. In practice, however, Forth programs are often written from both ends towards the middle. At the outset, you know what top-level program behavior you need, and you know what the low-level hardware-interactive words have to do. It's the stuff in the middle that needs working out.

You can assign names to functions and use these names before you define them. (Give them null definitions if you want to test compile.) The top-level word of a program might be an endless loop which starts with the word GET.FRONT.PANEL.INPUT followed by CHECK.USER.INPUT.LIMITS, thereby using Forth as its own PDL. Of course, having assumed the existence of CHECK.USER.INPUT.LIMITS, you eventually have to define exactly what this word should accomplish.

Breaking down a program into manageable and self-descriptive chunks is exactly what all good programmers do. The difference: in Forth, the end result is an executable program, not just the first step of a long process.

Is Forth a Compiler?

Forth is compiled, but its user interface behaves like an interpreter. **SX-Forth don't have Compiler, compilation is made using C compiler only**. It maintains a dictionary of all the words it knows. Each definition consists of a list of the addresses of the words that form the definition. (For code brevity, Forth on machines with 32-bit or longer addresses may use 16-bit tokens rather than addresses.) Compilation adds the new words and their definitions to the dictionary. SX-Forth use new words written in C or in Forth source. Both types of new words are added to dictionary only at compile-time.

Because Forth compiles each word in the source one-for-one to an execution address, a Forth compiler resembles an assembler. Figure 1 shows the complete flowchart of a Forth compiler and interpreter. Similar charts for C are 4' x 6' wall posters.

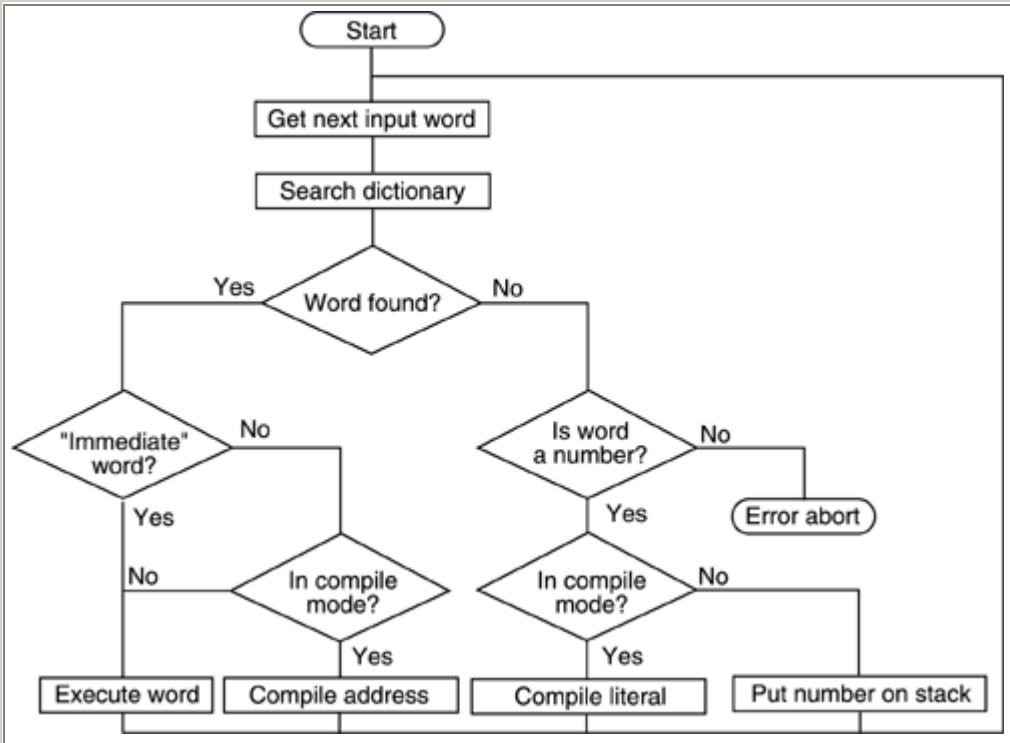

Figure 1—Here's the complete flowchart of a Forth compiler. The loop is executed once for each word in the source.

It may be helpful to think of a Forth program as consisting entirely of subroutines. Since every word calls a subroutine, there is no need for a CALL instruction, only an address. At run time, a machine code fragment fetches the next instruction address, saves the current program counter on the return stack, and executes the call. This tiny overhead is executed once for each word, making Forth marginally slower than an optimized assembly-language program.

I made SX-Forth from the ground-up using Figure 1, without compilation.

This model was defined by the Forth inventor Charles MOORE in the 1970 http://colorforth.com/bio.html .

How Forth Works

Executing an endless series of subroutine calls is not a very productive enterprise. Luckily, some 60 Forth words are defined in machine language or C language. Every definition eventually calls a combination of these "primitives CORE" and does some real work.

The primitives define a **Virtual Forth Machine**. To port Forth to a new system, only the primitives are rewritten. While some Forths run under DOS or Windows, in an embedded application the machine-code definitions of the primitives are the operating system.

Forth passes parameters onto a stack. Before a word is executed, the necessary parameters must be present on the stack. After execution, the results, if any, are left on the stack.

This is precisely what happens in most modern computer languages, but the stack is usually hidden. In Forth, the programmer is aware of what is on the stack and can manipulate it directly. For example, the primitive Forth word SWAP exchanges the top two elements of the stack. Most languages save pending operations. When you write C = A + B, the compiler puts

the "equals" and "plus" operations on its pending list until it gets to the end of the expression. Then, it rewrites it as "Fetch A, Fetch B, Add, Store C."

Forth cuts out the middle step. In Forth, you write the same operation as A @ B @ + C !. The @ and ! are Forth's shorthand for the "fetch" and "store" operations. The + , oddly enough, represents addition.

Luckily, only a handful of Forth words are this cryptic. Most Forths accept names up to 31 characters long, and most standard words describe their function. Good Forth is self-commenting, so make your words as self-descriptive as possible. At run time, any numbers you type are placed on top of the parameter stack. You debug a word by typing its input parameters, then the word. It executes immediately as if Forth were an interpreter, allowing you to check the results on the stack.

A stack element typically has 32 bits (some Forths use 16, SX-Forth can be 16 or 32 bits based on the microcontroller architecture) and is untyped. It might represent a signed or unsigned integer, an address, a single-precision floating-point number, or a Boolean flag, or an object address. You need to keep track.

Forth's philosophy is to permit, not to impede. If you have a good reason to add a Boolean to an address, Forth won't stop you. For that matter, there's nothing in Forth that prevents you from putting the wrong number of items on the stack. Forth is fast and efficient, but you have to keep your eyes open.

Creating a New Definition

Perhaps the most important word in Forth is the colon, which switches the compiler from run mode to compile mode and creates a new dictionary definition. The first word in the source after a colon is the name of the word to be defined. The definition follows the name. Logically enough, the definition is terminated by a semi-colon. This compiles a return instruction and switches the compiler back to run mode.

Thus, a complete Forth definition might look like this:

: MAGNITUDE (X Y—vector magnitude) DUP * SWAP DUP * + SQRT ;

The expression in parentheses is the stack picture. It reminds the programmer what the word's input and output parameters are. DUP (duplicate) generates a second copy of the top element on the stack, * is a single-precision multiplication, and SQRT takes the square root of a number.

As an example of Forth's flexibility, suppose you had a hankering for C's ++ operation. Forth's nearest equivalent is +! which adds a specified number to a variable. If you define : ++ 1 SWAP +! ; then ALPHA ++ adds one to the variable ALPHA. What Forth does not allow, but C does, is writing this as ALPHA++. Since Forth does not parse expressions, it would read ALPHA++ as an undefined word.

SX-Forth don't have Compiler, compilation is made using C compiler only. New WORDs can be entirely written in C or embedded in a string that hold the Forth source code.

Program Structure

Forth is highly structured. There is a way to compile a GOTO if you really want to, but normally you use the words IF, ELSE, THEN, BEGIN, UNTIL, WHILE, REPEAT, DO, and LOOP to control the flow of the program. These words compile conditional jumps into a definition.

Forth's IF checks the top of the stack for a flag left by one of Forth's many comparison words. To execute option 1 if the top two numbers on the stack are equal and option 2 if they are not, Forth's syntax is:

= IF do-option-1 ELSE do-option-2 THEN.

(I use ENDIF in my programs because I feel that THEN is an illogical throw-back to BASIC. Forth condones such personal idiosyncrasies, even if your bosses and co-workers may not.)

SX-Forth define only the IF … ELSE … ENDIF construct for making decisions and DO … LOOP construct for making iterations. D0 … LOOP can have a '==0_BREAK' for iteration breaking if top of stack is zero.

Constants, Variables, and Strings

A number in the source is compiled as a literal. A named constant stores a value at compile time and puts that value on the stack when it is invoked. Naming a variable compiles a storage space. Using a variable puts that address on the stack, ready for a fetch or store operation. A Forth string is just a variable whose first byte specifies its length.

Since a variable supplies its address, it can be manipulated before being used. For example, if you were using a Forth that had no ARRAY structure, you could define one. Forth can specify new types of defining words. Alternatively, you could fake it. BETA 7 + C@ fetches the eighth byte in the array that starts at the address of the variable BETA.

One deficiency of Forth source is that it may be unclear whether a word represents a variable or a function. Some follow the convention of hyphenating variable names but splitting function names with periods. Since good Forth code can be quite English-like, it is handy to distinguish code from comments visually without needing to parse a line. Thus, many commonly use upper case for code and lower case for comments.

SX-Forth use internal built-in variables I and J with many dedicated operators; I can be used for indirect addressing all kind of memories with automatic post-increment or post-decrement.

SX-Forth use the type of memory currently named (RAM, EEPROM, FLASH, RAM_USER area, …) for memories access that can be made on bytes (8 bits) or word (16 or 32 bits).

Hardware for Forth

Forth has been implemented on just about every microprocessor which has ever existed but some chips are more suitable than others. Obviously, the closer the chip architecture comes to the Forth virtual machine outlined in Figure 2, the better Forth runs. Forth needs two stacks so it runs faster on a chip that supports more than one. Since Forth needs few registers, a chip with many is just a lot of wasted silicon.

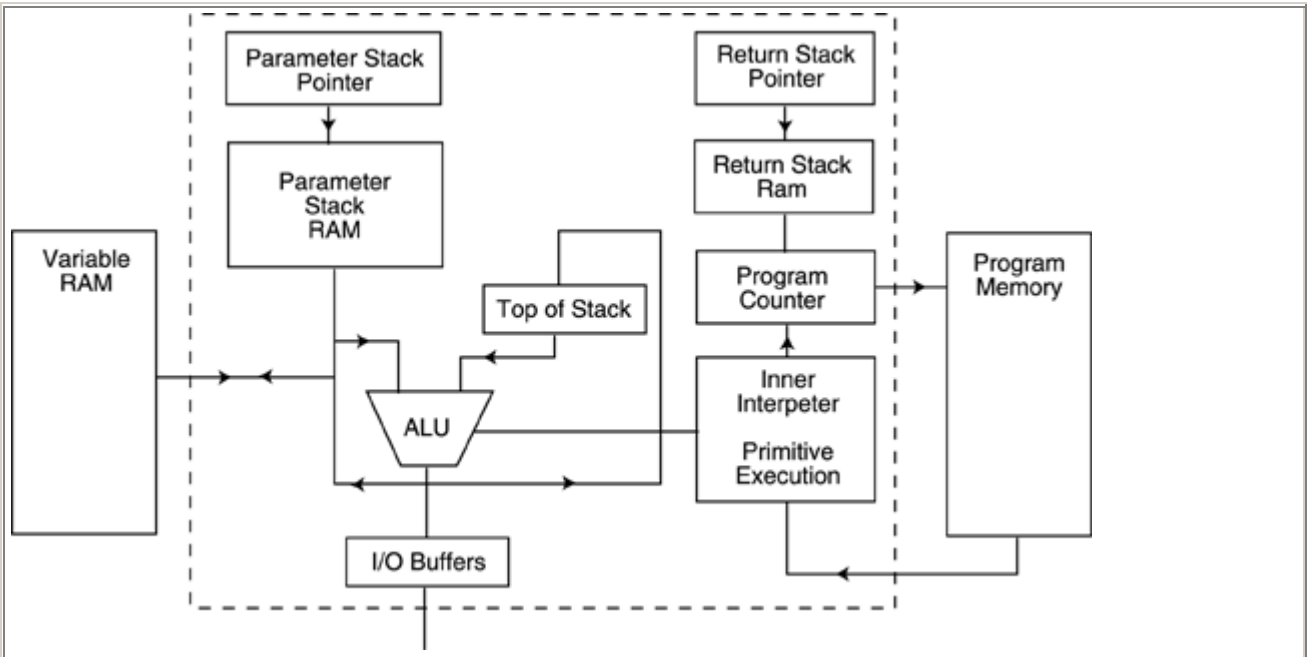

Figure 2—The Forth virtual machine has a Harvard architecture. Hardware implementations frequently keep
the top stack element in a separate register.

Minimal Forths do arithmetic on 16- or 32-bit integers, so Forth runs slowly on eight-bit chips. Historically, Motorola microprocessors have been more suited to Forth than Intel ones. The MC6809 and the MC680X0 series were ideal 8-bit chips for Forth.

SX-Forth use 6.8 KB of FLASH on an AVR 8 bits RISC microcontroller with Harvard Architecture with RAM used only for stacks. On 16 or 32 bits microcontrollers with Von Newman architecture the amount of required FLASH drastically decrease.

Since the Forth virtual machine is relatively simple, it can be implemented on a gate array. A pioneering effort by Charles Moore, the inventor of Forth, led Harris to introduce their RTX 2000 in 1989. This 10-MIPS single-chip Forth engine used a Harvard architecture with its parameter and return stacks on the chip. Unfortunately, it was not a commercial success and is now used only in niche markets such as processors on satellites.

Using Forth

Commercial and public-domain versions of Forth exist at all levels. For an embedded application on an 8- or 16-bit processor, it may be most convenient to write Forth programs on a PC and then transfer the finished code to the target system. While the development system may use the full facilities of DOS or Windows, there is no need for the finished product to contain more than a small run-time package and the program itself. Even the dictionary is only needed when compiling and debugging the program. It can be omitted from the final code.

Because Forth programs tend to compile to about 10 bytes per line, a 2000-line program plus a 4-KB run-time file easily fits in a 32-KB PROM. If the target system supports a serial port and executes from RAM, I prefer to compile and debug on the target. Even though this means finding memory space for the dictionary and the compiler, it helps immensely in testing the hardware. I've resolved many hardware bugs by programming a short Forth loop to wiggle a bit so I could follow a signal through the suspect area with an oscilloscope.

A commercial Forth comes with an initial dictionary that contains the Forth primitives and the words needed to implement the compiler. Many Forths have a built-in source editor, but you can use any editor you find convenient. You will probably be supplied with libraries of the operating system calls needed to develop Forth programs under OSs like Windows. For an embedded application using a single-card PC, a DOS function library is useful.

You can also get a library of Forth extensions, which you can load if your program requires them. For example, simple Forths process only integers, so floating-point arithmetic is optional. It's easy to write your own libraries. I've been using JForth from Delta Research to write programs which analyze filters. JForth supports windows, pull-down menus, input gadgets, and slider control of parameters. However, it can't handle complex numbers. I wrote my own floating-point complex arithmetic library in 20 minutes.

In a team writing programs to control a series of related instruments, one member should be assigned to write a library of hardware-interface functions to do things like access the front-panel controls and displays in a consistent manner. Because Forth lets programmers develop idiosyncratic ways of solving problems, you must have good documentation and close coordination between team members on a large project. Companies using Forth should maintain in-house standards for Forth extensions that relate to their products and teach these to all their programmers.

What Can You Do With Forth?

The simple answer: anything. Other computer languages limit you to the set of operations the compiler authors thought you'd need. Because Forth is naturally extensible, you can do what you need. If all else fails, you can easily drop into machine code and create any data structure you need.

JForth even implements C structs, which it uses to interact with the host operating system. I once needed a structure which was written to 30 named, one-dimensional arrays. It was read as a single, named, two-dimensional array. I'm told this is feasible in C, but I've never met anyone who wanted to try it.

Forth Standards

Since its invention by Charles Moore about 1970, many Forth standards and dialects have emerged. Forth encourages innovation so there has always been a tendency for Forth vendors to customize and improve it, even when ostensibly adhering to a standard. I do most of my professional Forth programming in 1979 vintage FIG-Forth on the grounds that it is already so obsolete that it won't change. Since then, there was Forth-79 and Forth-83 and now there's an ANSI standard for Forth (X3.215/1994).

SX-Forth use a subset of ANSI Forth CORE words with some modification and add specialized words for doing a scripting language on a C compile-time development environment.

SX-Forth, as many Forth, is implemented using my 'vision of firmware development'... see 'Thinking FORTH' of Leo Brodie at http://home.earthlink.net/~lbrodie/forth.html to understand why !

How does Forth Compare with C?

Both Forth and C let you think at a higher level and spare you the slower development process of assembler. Forth's logical compilation order eliminates the need for C prototypes within a file.

All the standard program controls of C (do, if, else, while, switch, etc.) are in Forth, often with the same names. All the important logical and mathematical operators are there, too. Conditional compilation, arrays, and unions are all supported with a Forth flair. Constants replace #defines, and Forth's direct stack manipulation eliminates most need for auto variables. Forth's use of vocabularies and its ability to FORGET definitions are more powerful than C's weak scope operators. You can support your own data types with even less pain than in C++.

Forth assumes you know what you are doing. It protects you from typographical errors and incomplete structures, but the manual has only one page of compiler error codes, not a whole chapter. As someone once said, Forth can't flag syntax errors since it doesn't know what syntax you decided to use.

In C, you're more protected. But then, you also have to do things like type casting to circumvent a patronizing compiler constantly checking up on you.

Forth has these advantages over C:

1. The development environment is much simpler. You don't need to install a whole development suite since Forth is its own development system and, in an embedded application, its own operating system. It offers an OS, source editor, and all the debug utilities you need, and they fit on one 360-KB floppy.
   As a result, you're working with a single tool set and a single user interface. Compare this with having a compiler, OS, debugger, and maybe a target monitor program, all coming from different vendors and not designed to work together.
2. When you buy Forth, you often get the source code for the whole development environment. Try telling Borland or Microsoft that you want to make a backward compatible variant of C to do stronger type checking, fuzzy logic, or different floating-point implementation.
3. It's often possible to develop a Forth program on the target system itself. In my present C contract, I use a Sun workstation to run Make and to compile and link a target executable. Then, on a target machine, I download the code before powering it up and testing it. If I want to make an adjustment, it takes an hour to go through the whole loop again.
   With Forth, I could type a new word right into the serial port of the target, push parameters onto the stack, then call it to see if it worked. I could easily "splice" the new word to intercept any calls to the old word.
4. The extensibility of the compiler lets you follow any coding fad that comes along without switching languages. Forth has been object oriented, "sandbox supporting," and platform independent from the get go. Added data structures or operator overloading that would choke C++ would not be a problem in Forth.
5. You can drop into assembly much easier than in C, and all data structures are accessible from assembler.
6. **Target testing is far easier. You can interactively examine and manipulate data using the same commands you used in the code. To do the same thing in C requires advanced knowledge. It takes lots of key pressing to dominate a debugger.**
7. **You don't need a target OS. Forth is at its best when it is the OS. Some Forths support multiple users and multitasking. Since each task has an independent parameter stack and return stack, task switching is essentially instantaneous.**

8. Forth allocates memory resources at compilation, which makes operation times determinate. It doesn't spend an unknown time defragmenting memory.
   In a real-time OS, I prefer not to dynamically allocate memory, but if you want something akin to alloc() and free(), that's up to you. A page of code would cover it.
   Forth can service interrupts with little latency since, being stack based, it doesn't need to save the context.

C functions can have only one return parameter. SX-Forth use always the data stack with a standardized 'Push(inParam)' and 'outParam=Pop()' C functions … so C can call Forth and viceversa and C can have as many return parameters as you want.

On the negative side, Forth can be a little slower. In a large program, it may use slightly more code than newer C compilers. However, although a "hello world" program in Forth might run to 2 KB, it has no huge run-time libraries to load. Forth encourages programmers to use fixed-point notation, which can greatly speed execution, but requires more analysis during coding.

The biggest drawback of Forth is the Catch 22 that attends any nonconformist idea. Not many people know Forth, and people won't usually learn something unless everyone else is already using it. That's how Mr. Gates makes a living.

If you can persuade your boss to let you use Forth, it can be your secret weapon. **Industry experience has shown that a good Forth programmer is up to ten times more productive than a C programmer.**

We'd like to show you some of the differences between Forth and C. For an embedded program using an onboard PIC to drive a jittering oscillator, we wrote an emulation program in Forth to show the PIC programmer how things should work. Listing 1, below, shows the PDL description of the outer loop of this program. Listing 2 offers the executable Forth. It took me about ten minutes. (In a real Forth program, this much code would be factored into several definitions.) Listing 3 presents the C version of the same program.

Listing 1—Here's the top level of a program for a jitter generator in PDL

```
Main Program:
HouseKeep                       (set ports, clear flags, set defaults)
Read upload bit          (has user saved previous settings?)
If low
        CopyPROM                (load defaults from EEPROM)
Endif
ReadConfiguration        (get former settings from EEPROM)
SetConfiguration  (set board registers)
Beginloop:                      (Start of endless loop)
        Read self-test bit
        Read self-test number
        If bit=0 and number <>0  (self test operation)
                Case:    (test number)
                         On 1 do test 1
                         On 2 do test 2
                         On 3 do test 3
                         On 4 do test 4
                Endcase;
        Else                    (normal operation)
                Read interface flag        (Check for faults or user input)
                If set
                         Read status word (Identify faults or user input)
If fault flag, do soft reset, endif
```

```
If jitter flag <> jitter state, toggle state, endif
If calibration request, Calibrate, endif
If Bit 0, SetAmplitude, Endif
If Bit 1, SetBitRate, SetAmplitude, Endif
If Bit 2, SetBitRate, SetAmplitude, Endif
If Bit 3, SetFrequency, Endif
If parameters have changed
                                        Update EEPROM
Endif
Clear interface flag
                    Endif
          Endif
Endloop;
```

Going Further

The best way to learn more about Forth is to join the nonprofit Forth Interest Group (FIG). They publish a journal, *Forth Dimensions*, and sell books and public domain versions of Forth.

The classical, but now dated, book on Forth is Leo Brodie's *'Starting Forth'*. If you can't find it elsewhere, you can buy it from FIG. **Brodie's *'Thinking Forth'* won't teach you how to use Forth from scratch but is a fine examination of the philosophy and structure of Forth and other languages**. Another good beginner's book is volume 1 of C. Kevin McCabe's *Forth Fundamentals*.

To implement an embedded system in Forth, you can adapt a public-domain version. Alternatively, you can buy a ready-made system designed around your target processor. Forth Inc. advertises versions of their DOS-based chipForth for the 80196, 80186, 68HC16, and 320C31. They also have Forths running under Windows for the 68HC11 and 8051 processors.

A number of smaller Forth vendors advertise in *Forth Dimensions*.

Why Not Use Forth?

We've often been told that it's easy to find C programmers and that hardly anyone uses Forth. This is true. Few "programmers" know Forth, but we've found that hardware engineers are often familiar with it. Engineers with programming experience generally write better embedded code than career programmers who are unfamiliar with hardware.

You need to ask what your company's aim is. If you really want to get product out the door, then check Forth out. It's the way to go.

Tom Napier has worked as a rocket scientist, health physicist, and engineering manager. He spent the last nine years developing space-craft communications equipment but is now a consultant and writer.

You may reach Eric via http://www.voicenet.com/~eric/forth.htm*.*

References and Sources

Forth Dimensions (journal), books on Forth, public-domain versions of Forth
Forth Interest Group (FIG)
100 Dolores St., Ste. 183

Carmel, CA 93923
http://www.forth.org/

Forth, Inc.
111 N. Sepulveda Blvd., Ste. 300
Manhattan Beach, CA 90266
http://www.forth.com/

forthinc@forth.com