Tiny Open Firmware: Extensibility for Small Embedded Systems.

Brad Eckert brad@tinyboot.com

Abstract

Tiny Open Firmware (TOF) is a flexible firmware architecture for embedded systems. TOF features an efficient late-binding mechanism that renders all ROM-based routines patchable and enables rapid compilation of processor-independent add-on code. A distant cousin of IEEE1275, TOF contains an evaluator that quickly converts tokenized Forth source code to native machine code and links the new code into the application. A TOF implementation is small, typically under 20K for 8051 and 68K processors.

Run-time compilation of add-on code simplifies systems whose configurations may change. At startup, a TOF-based system reads and evaluates tokenized boot code from add-on devices. Add-on hardware modules contain their own boot code that links added features into the application. TOF brings self-installing plug-and-play hardware to small embedded systems.

Introduction

Today's microprocessor-based electronic and electromechanical equipment can often be designed to accommodate add-on accessories. Add-on hardware must either live with the design constraints of the original firmware or provide a means of upgrading firmware in the field. One solution is to have the add-on hardware patch the application at run time so as to take advantage of the new hardware.

Letting add-on hardware reconfigure the application at startup greatly simplifies plug-and-play operation. Unlike the PC world's approach to plug-and-play, this one is pretty painless. The user physically plugs in a hardware widget to install and unplugs it to uninstall. Add-on firmware functionality is contained in the widget.

A common way to patch code at run time is to put hooks (function pointers in C) at strategic points in the application. These hooks point to function pointers in RAM, which can be patched. For example, you might decide that `putch` may change, so you define a default ROM version `DefPutch` and then declare `void (*putch)(byte) = DefPutch;`. To define a patch, you can compile a new version of `putch` for an absolute location in memory. To patch the ROM code, you'd load the patch code into the RAM location you compiled for and then change the function pointer for `putch` to point to the new code.

This method can be used with assembly too. Either way, there are serious drawbacks:

1. The patch code is native code, and on many platforms, absolute code. This greatly complicates the use of multiple add-ons. Plus, you have to freeze the ROM if the add-on code is to re-use sections of ROM code.

2. You need to provide enough hooks to address every anticipated need. Invariably, a need comes up that you didn't think of so the add-on code has to replicate a big chunk of the application in order to work. This bloated code eats up system resources that may already be scarce.

3. Changing the hardware design or moving to a new processor will break existing add-on code. Accessories in the field that can't practically be updated will be rendered obsolete.

Besides flexibility in the interface between add-on code and application, you need a way to handle run-time variations and provide extensibility. If you want to enable the end user to write add-on code, you can't just give them your source code and tell them to buy a compiler toolchain. An interpreter such as Lua, TCL or Java could provide a virtual machine to isolate add-on code from the implementation details of your application. But, interpreters are slow and usually bulky and don't really address the interface issue. Better to compile add-on code at startup.

One solution is to implement a computer language (preferably not a new one) in a way that solves these problems. A late binding mechanism that renders all subroutines patchable at run time would solve part of the extensibility problem. Any subroutine could be patched with a relatively small amount of machine code. And, although a token interpreter offers some extensibility, real extensibility requires removing the wall between application and add-on code.

Forth to the Rescue

Much flexibility can be attained if add-on code is compiled at boot time. The compiler needs to quickly translate source code in the add-on module to machine code at startup. It also needs to be able to execute commands that bind the new firmware features into the application. The combination of run-time compilation and immediate execution renders the application extensible.

Extensibility is a key feature of the Forth programming language. Forth is an industry-proven computer language originally developed for real-time control. Forth is used by the IEEE1275 "Open Firmware" standard to boot up millions of Sparc and PowerPC workstations. On a workstation motherboard, Open Firmware probes the busses for add-on cards and loads driver code from a ROM resident on the card. The driver code is in tokenized form, meaning that Forth keywords are represented by numbers instead of ASCII strings. The most often used Forth keywords are represented by one-byte tokens, leading to very compact code. Semantically, it's still processor-independent Forth source code.

A dialect of Forth called Tiny Open Firmware (TOF) implements the features of IEEE1275 useful to small embedded systems. TOF uses subroutine/native threading in which Forth keywords (tokens) are converted to subroutine calls or inline machine code. TOF adds an extra level of indirection to each subroutine call in the form of a RAM-based jump table. Instead of calling a subroutine directly, compiled code calls into an array of jump instructions. For each subroutine call the extra overhead is one jump instruction. On some processors there is a pipeline stall penalty, but it's still an efficient way to achieve late binding.

The RAM-based jump table, called the binding table, is initialized from ROM at startup. You can think of a TOF-based system as an object with hundreds or thousands of late-bound methods. You can patch any ROM-based subroutine by putting new code in RAM and changing the appropriate binding table entry to jump to it. All code that uses it will be redirected to the new version. Besides giving the application fine-grained patchability, the binding table enables rapid compilation.

A freeware Windows program called Firmware Studio implements TOF for several processors. It's available at http://www.tinyboot.com. Full source is included.

A Different Kind of Language

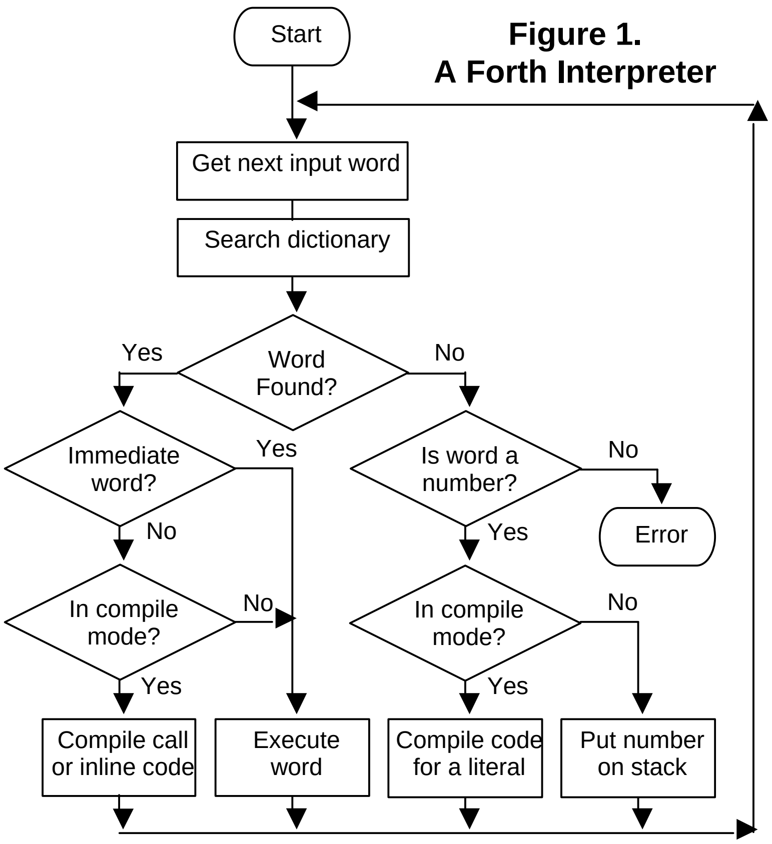
Most programmers today haven’t had much exposure to Forth, although it’s not a hard language. TOF needs to be described within the context of Forth. Forth has been in embedded systems for about 30 years. Over the last decade, the Forth community has rallied around an ANS standard based on specification rather than implementation, and this shift in thinking has allowed Forth to move forward on many fronts. Let’s have a look at Forth.

Forth is a computer language that has a simple syntax and many keywords. This is in contrast to Algol-style languages that have a complex syntax and few keywords. If you are familiar with C or Pascal, try to forget about syntax as you learn about Forth.

Forth is compiled, yet it has no compiler in the traditional sense. Essentially, it consists of a population of subroutines and an interpreter. Subroutines are called *words* in Forth. The interpreter can invoke words that perform compilation actions.

There are two things that characterize Forth: Its interpreter mechanism and its collection of words. Forth is a definition tool whose lexicon conforms to a language specification.

Figure 1.
A Forth Interpreter



Forth is a programming language, a compiler, a debugging tool, an assembler, and an operating system. It is many things to many people.

There are many implementations and many threading mechanisms. Despite their differences, the thing that all Forths have in common is a syntax based on blank-delimited strings.

Just as Newtonian Physics has F=ma at its core, Forth has a simple interpreter at its core.

Figure 1 shows a typical Forth interpreter. A Forth interpreter evaluates blank-delimited strings taken from an input stream such as a console or file, usually in one pass. This implementation uses subroutine/native threading, meaning that it uses CALL instructions to invoke Forth words. Simple Forth words that can be done with one or two machine instructions are inlined instead of called.

This simple interpreter gives Forth its characteristic RPN syntax. For example, typing **`1 2 +`** at the console leaves **`3`** on the stack. Operators perform simple transforms on the top of a data stack. Parameters are implicitly passed between Forth words via the data stack, so less code is wasted shuffling parameters around. Implicit parameter passing also allows a natural language syntax, since the source needn't be cluttered with parameter declarations.

The data stack is a LIFO list used to hold intermediate values such as numbers and addresses of words and data. Forth also makes use of the return stack for temporary storage. For stack pointers, usually one stack uses the CPU's stack pointer and the other uses a register.

Forth is made up of an interpreter and a large number of keywords. These keywords are small subroutines held in a data structure called the dictionary. The dictionary consists of a linked list of headers and accompanying code. Each header contains the name of the word, a flag byte, a link to the next header in the dictionary, and possibly a link to the executable code. Each word has an *immediate* flag to denote whether or not it's an immediate word.

Essentially, the language is the compiler. Some keywords perform compilation actions, and you can always define new ones. The word `:` (colon) begins a new Forth definition. It creates a new header and sets a flag called *state*. When *state* is set, the interpreter is in compile mode. The word `;` (semicolon) is an immediate word (its *immediate* flag is set) that ends a definition and clears *state*. Immediate words are typically used to form loops and control structures.

Firmware Studio is a Windows-based Forth implementation that uses the algorithm shown in Figure 1 to compile ROM code for a target processor. The dictionary consists of a linked list of headers and a ROM image. Firmware Studio assigns each new word an execution token (*xt*) and stores it in the word's header structure. Each word in the system has a different *xt*. This is a key feature of TOF.

TOF uses two compilation modes: *static* and *dynamic*. Both compile CALL instructions. In static mode, the destination is the address of the word. This is typical of subroutine threaded Forths. In dynamic mode, the destination is computed from the word's *xt* so as to compile calls into the binding table. The binding table is an array of jump instructions in RAM, initialized at startup. Most words are compiled in dynamic mode.

Let's look at a typical definition:

```
: #S     ( ud -- 0 ) begin # 2dup d0<> while repeat ;
```

Here is what the interpreter does for each of the blank-delimited strings. Immediate words are underlined.

:	Creates a header for #S, enters compile mode.
<u>(</u>	Comment: discards input up to).
<u>begin</u>	Marks a backward reference: Pushes 0x489A onto the stack.
#	Compiles a call to #. Binding table grows downward from 0xFFFF.
2dup	Copies inline code for 2dup.
d0<>	Compiles a call to D0<>.
<u>while</u>	Compiles code to do a conditional branch and marks a forward reference: Pushes 0x48B2 onto the stack.
<u>repeat</u>	Lays down a backward branch to the address marked by <u>begin</u>, resolves the branch offset at the address left by <u>while</u>.
<u>;</u>	Lays down a return instruction, exits compile mode.

The 68K/Coldfire implementation compiles the following ROM code for #S:

```
0000489A 4EB90000FB50 #S:      JSR 0xFB50(xt:200) #
000048A0 2015                  MOVE.L (A5),D0
000048A2 2B07                  MOVE.L D7,-(A5)
000048A4 2B00                  MOVE.L D0,-(A5)
000048A6 4EB90000FCB2          JSR 0xFCB2(xt:141) D0<>
000048AC 2007                  MOVE.L D7,D0
000048AE 2E1D                  MOVE.L (A5)+,D7
000048B0 4A80                  TST.L D0
000048B2 67000006              BEQ 0x48BA
000048B6 6000FFE2              BRA 0x489A:#S
000048BA 4E75                  RTS
```

Tokenized Code

Firmware Studio has a specialized interpreter called a tokenizer. Instead of compiling machine code, it converts Forth keywords into tokens. The resulting tokenized code is semantically equivalent to its textual Forth source, but is stripped of comments and stored in a compact form.

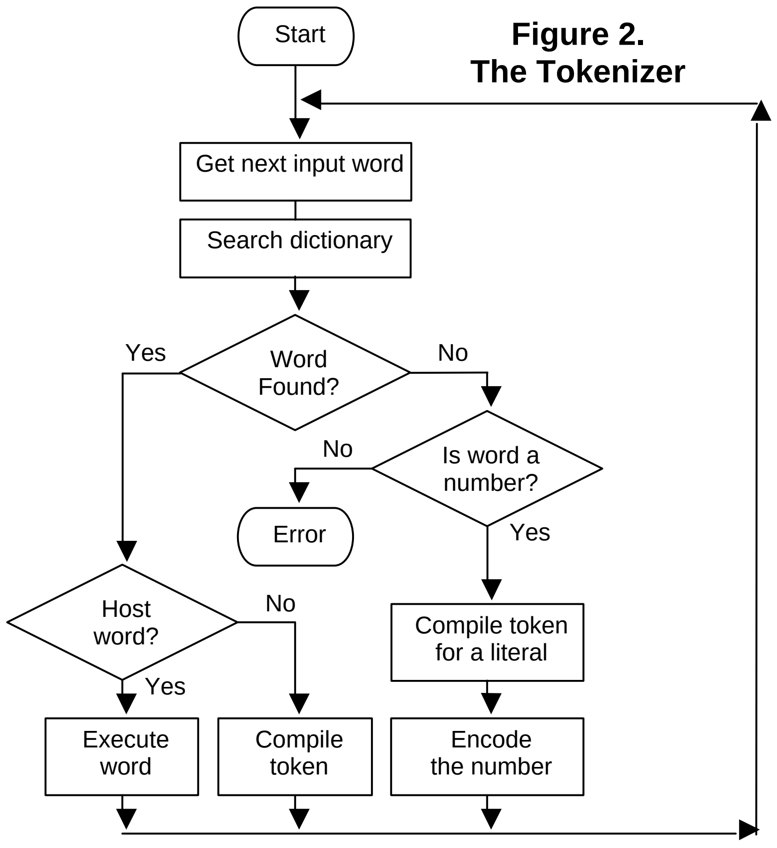


Figure 2. The Tokenizer

The tokenizer is the part of TOF that runs on the host PC. The host knows the token assignments of Forth keywords and can compile tokenized code. Tokenized code may be used as boot code for add-on hardware.

For interaction with the target hardware, tokenized code is sent to the target over a communication link for immediate evaluation. Typically, console input is tokenized and sent to a free part of RAM in the target. Then a program resident in the target evaluates the tokenized code.

Tokenized code may also be stored in the program ROM to archive temporary features so as to pack more features into a space-limited application.

Figure 2 shows a tokenizer, which is similar to a typical Forth interpreter. Token values between 0x20 and 0xFF are encoded using one byte, others are encoded using two bytes. Two-byte values concatenate the lower five bits of the first byte with all of the second byte for a possible range of 0x0000 to 0x1FFF.

Host words are parts of the tokenizer resident in a special wordlist. They're mostly defining words. The tokenizer has a *state* flag, which defining words use to keep semantic consistency with the original Forth source.

When a ROM image is built, token numbers are assigned to word names. These token assignments can be saved to a file. A token file really acts as an interface specification that connects add-on code to ROM routines. This file can be used instead of the original source code to set up token assignments. It can be given away without revealing proprietary source code, enabling third parties to write add-on code.

Tokenized code serves as input for the evaluator.

The Evaluator

The evaluator is a Forth program resident on the target hardware. It converts tokenized Forth source to machine code. The evaluator is like a traditional Forth interpreter except that it computes the location of Forth words instead of traversing a header list looking for them.

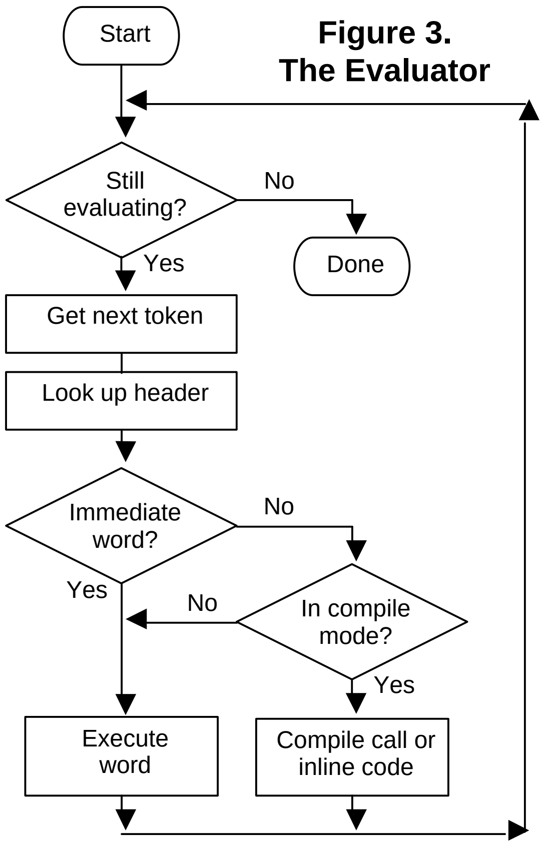


Figure 3.
The Evaluator

The evaluator is the part of TOF that runs on the target hardware. It uses the token value as an index into the binding table. This index is used as the call destination for compiled words. Every word is preceded by a header byte containing an *immediate* flag. To get this flag, the index is used to extract the address of the code from its binding table entry and the byte immediately before the code is fetched.

Numbers are represented by an immediate word like (LIT8) followed by one or more data bytes. (LIT8) and similar words fetch data from the input stream and sign or zero extend it if necessary.

The evaluator fetches bytes sequentially from the input stream until the END token is executed. END causes evaluation to end.

```
E1 02 06 : STARS
4B         0
F0         DO
02 05      STAR
F2         LOOP
E2         ;
```

Here is a simple tokenized Forth word. Words associated with tokens E2, F0 and F2 are immediate words.

Token values between 0x1000 and 0x1FFF are regarded as relative tokens. The evaluator subtracts 0x1000 and adds the highest unused token value of the last evaluation session. This has the effect of mapping the relative tokens of each add-on peripheral onto a different set of unused absolute token values. Consider the case where your application's ROM has *xt* values up to 0x480, peripheral A has *xt* values ranging from 0x1000 to 0x1021, and peripheral B has *xt* values ranging from 0x1000 to 0x1014. Peripheral A's words will be mapped to the *xt* values 0x481..0x4A2 and peripheral B's words will be mapped to 0x4A3..0x4B7.

The evaluator is TOF's version of the classical Forth interpreter. Instead of feeding it textual source, you feed it tokenized source. This source can come from a host PC, non-volatile memory in add-on peripherals, program ROM, or any other data sources available at run time.

Putting it all together

Tiny Open Firmware removes the wall between application and add-on code. Add-on boot firmware is free to invade the application and do anything it wants, to any hardware or code that it wants. You control the add-on code, so you know your guests are reasonably well behaved. The software equivalent of a bouncer can keep out unknown code.

A typical system has some kind of expansion bus. At startup, TOF probes each module on the bus looking for boot code. If it finds it, the evaluator verifies its boilerplate and checksum and then evaluates the boot code. TOF continues probing the bus until all boot code has been evaluated.

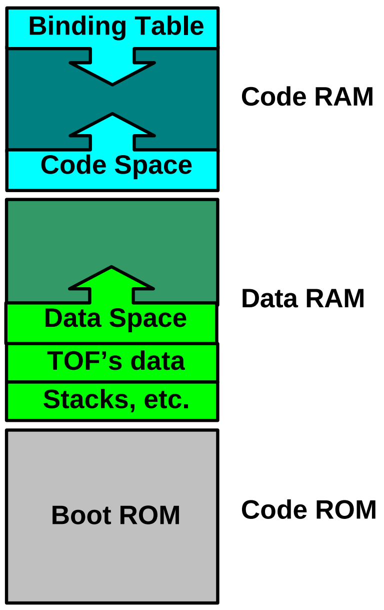
Add-on modules usually aren't just generic hardware. They are designed to supplement the application. As such, their boot firmware patches part of the application to make use of the new hardware. A typical boot program defines driver code and an application extension for the new hardware, initializes it, and links the code into the application.

The evaluator can handle boot code from multiple modules, each device containing tokenized source that's compiled to native machine code at startup. Installation instructions for the end user reduce to “Plug it in”.

For debugging, the tokenizer and evaluator together act as a normal Forth interpreter. Keyboard input is tokenized by the host PC, sent to the target and evaluated. The resulting output is read and displayed by the host PC. Since most Forth code is inherently reentrant, the debugger has its own execution thread that lets the application run while debugging is underway. TOF supports live debugging with which you can probe and patch a live, running system.

Target hardware

The target hardware requires program RAM and data RAM. These may be the same memory device, or different devices. Whatever the case, there are two separate memory areas: code space and data space. They both grow upward in memory. The system is assumed to boot from ROM.



When a ROM is built using Firmware Studio, RAM is allocated in data space. At startup, this RAM is initialized from ROM. The initialization table uses run length encoding to limit the impact of large, blank areas of pre-initialized RAM. The binding table is also initialized from ROM.

In a Coldfire/68K system, code and data space overlap. The memory map is close to the diagram shown on the left. The boot ROM can be copied to and run out of RAM to get more speed, especially if it’s an 8-bit boot device.

In a SOC implementation where you’re free to use a Harvard architecture, code and data spaces can be separate physical memories. Also, the binding table can be implemented in hardware so as to incur no run-time overhead, and actually result in quicker calls because fewer bits are needed to represent a token than a physical address. The falling cost of programmable logic and design tools is making this approach ever more attractive. TOF can be ported to a custom stack processor in a matter of weeks.

Porting to other processors

So far, TOF has been ported to 8051 and 68K/Coldfire processors. This covers a wide performance spectrum. Porting to a new processor takes some work, but it can be done in a few weeks. The effort isn't unreasonable since the source code for the assemblers, disassemblers and builders of other processors is present.

Firmware Studio also includes builders and disassemblers for 386, M-core and AVR. The AVR builder has been used heavily so it's pretty stable. The AVR is good at Forth but won't run subroutine threaded TOF because of a lack of program RAM. The 386 and M-core builders haven't been tested much since I haven't had occasion to finish porting TOF to these processors.

Experience shows that any processor that's designed to run C code efficiently will also run Forth code efficiently. The only problem with running TOF is the need for RAM. Subroutine threading requires availability of program RAM. If some speed can be sacrificed, a different threading method can be used to reduce RAM requirements.

Token threading can be used to run TOF on processors with small RAM. Each Forth call is encoded as two bytes. Table1 lists estimated expansion capacity using token threading.

Table 1: Some single-chip microcontrollers I've looked at for TOF implementation.

uC	Speed	Headroom	SRAM	Flash	Threading type	Notes
Atmel ATmega64	Fair	80	4K	64K	Token	AVR
Atmel AT91F40816	Excellent	80 [160]	8K	2M	Subroutine [Token]	ARM7
Hitachi HD64F3684	Good	50 [80]	4K	32K	Subroutine [Token]	H8tiny
Hitachi H8/3068	Good	300 [500]	16K	384K	Subroutine [Token]	H8/300H
Hitachi SH7046	Excellent	100 [300]	12K	256K	Subroutine [Token]	SuperH
Microchip PIC18F680	Fair	40	3.75K	64K	Token	[1]
Mitsubishi M30625FGAGP	Good	300 [600]	20K	256K	Subroutine [Token]	M16C
Motorola 68HC912DG128	Fair	200	8K	128K	Token	HC12
Motorola MMC2107	Excellent	50 [160]	8K	128K	Subroutine [Token]	Mcore
Motorola MPC555	Excellent	200 [700]	24K	448K	Subroutine [Token]	PPC
Samsung S3FB42F	Good	1K [2K]	48K	213K	Subroutine [Token]	[1][2]
ST ST72651	Fair	70 [100]	5K	32K	Subroutine [Token]	ST7
ST ST92F150JD	Fair	100 [150]	6K	128K	Subroutine [Token]	ST9
ST ST10F168	Good	100 [150]	6K	256K	Subroutine [Token]	ST10
Toshiba TMP93PW44DF	Good	50 [80]	4K	128K	Subroutine [Token]	[3]
Ubicom IP2022	Excellent	300 [600]	20K	64K	Subroutine [Token]	Ubicom

Notes: [1]=future device as of 1Q02 [2]=masked ROM [3]=OTP

The headroom rating is the estimated number of definitions that the SRAM will hold assuming TOF and the application use 800 tokens and each add-on definition averages 10 words. It also sets aside 25% of the remaining memory for data space and miscellaneous stuff.

The speed rating is a very rough estimate based on the fastest threading method. Token threading is roughly ½ to ¼ the speed of subroutine threading, depending on the quality of optimization in the ROM builder. On 32-bit chips, there is a big headroom difference between them because subroutine calls take 4 or 6 bytes versus 2 bytes for a token.

For some processors, internal RAM is very small. A reduced version of TOF with most of the binding table in ROM would free up some space. Such an approach would allow TOF and an application to run with as little as 2 KB of SRAM, leaving 1K to 1.5K free for add-on code and data. Doing so would save about 900 bytes of RAM and result in some added complexity in handling the binding table.

In many cases, the binding table can be built from addresses instead of jump instructions, saving space on 16-bit processors while incurring little run time penalty. The headroom rating assumes that this has been done. For example, the 68K/Coldfire would use

```
MOVEA.L 0x4C814,A0
JSR (A0)
```

Instead of

```
JSR 0x4C814.
```

The headroom rating is just a ballpark figure and is most meaningful when comparing token threaded implementations. Subroutine threading adds processor-dependent variables.

The speed rating is an indication of threading efficiency, not raw processor speed. Although threading efficiency is lower for token threading, it doesn't necessarily have a large impact on overall performance. What it does mean is that more time-critical words will have to be expressed in assembly instead of Forth.

Token Threading

Token threading relies on a section of code called a token interpreter. For portability, we assume that add-on code can't write to program memory. The tokens in this context are bytecodes: 8-bit or 16-bit values used as indices into the binding table. Elements in the binding table are addresses.

An instruction pointer IP points to the next token to be executed. The token interpreter fetches the address of the next token (BindingTable[codemem[IP++]]) and determines whether it is machine code or threaded code. The address space is mapped onto real hardware by treating different address ranges differently. Here is one possible mapping:

```
0000..5FFF =   machine code in ROM at code location 0000..5FFF
6000..DFFF = tokenized code in ROM at code location 0000..7FFF
E000..FFFF = tokenized code in RAM at data location 0000..1FFF
```

Token threading can be very compact, although it is slower than subroutine/native threading. If both threading methods are supported, you can switch the compiler back and forth as needed to get compact code at minimal execution expense.

Keep on rockin' in the C world

We live in a C world. Code generators create C, IP comes in C source and there are lots of programmers fluent in C. So, an application will often have to contain sections of C. Fortunately, it's not hard to use both Forth and C in the same application.

The exact interface depends on the C implementation. For example, the register usage of Diab's Coldfire C functions doesn't overlap Forth's register usage. Some compile options cause C functions to use A5, in which case TOF must save A5 before the call and restore it afterwards.

Language	Scratch	Saved	Reserved
TOF Coldfire Forth	D0-D3, A0-A2	D4-D6, A6 are not touched	A3-A5
Diab Coldfire C	D0-D1, A0-A1	D2-D7, A2-A4 are saved	A6-A7, sometimes A5

Calling a C function from Forth is straightforward. Transfer data from TOF's data stack to C's argument stack, call the function and push the result onto TOF's data stack.

The absolute address of the C function must be known. One way to fix their locations at known addresses is to use assembly code in the C source to manually build a small table of JMP instructions. This extra level of indirection would insulate the Forth code from address changes due to ROM re-builds.

The Diab compiler passes parameters on the return stack. The compiler can be told to pass some arguments via D0/D1/A0/A1. To preserve reentrancy, let's assume just the return stack is used.

Calling C from Forth

To call a C function like "int FOO (int ARG1, int ARG2)", the following could be used assuming that the code for FOO is in ROM location 0x9044C.

```
code C_FOO  ( Arg1 Arg2 -- result )
        move.l tos,-(r)         \ push Arg2
        move.l (s)+,-(r)        \ push Arg1
        move.l A5,D2            \ just in case FOO trashes A5
        jsr 9044C               \ FOO consumes args and returns D0
        movea.l D2,A5
        move.l D0,tos           \ Diab functions always return result in D0.
        rts c;
```

Calling Forth from C

Calling a TOF function from C is a little more involved. C functions save most registers, but since the compiler is free to use registers within a function you don't know which Forth registers are valid at the time you call the TOF function. So, the best policy is to save and restore all registers that are used by TOF.

In switching from C context to Forth context and vice versa, the register set D0..D3/D7/A0..A5 would be exchanged with a register image at a fixed RAM location.

Calling a TOF function from C generally involves assembly. Calling into the binding table insulates the code from any address changes due to ROM re-builds.

At system startup, the C application initializes itself and then jumps to the TOF initialization code. TOF does its startup thing, sets up the register image it will need later, and returns control to the C application or just runs an application itself.

TOF can be mixed with C to give a C application more flexibility. For example, it can act as a low-level interface to hardware that insulates the application from hardware configuration changes. Or, it can run at the application level to present a flexible user interface that can be modified by add-on hardware.

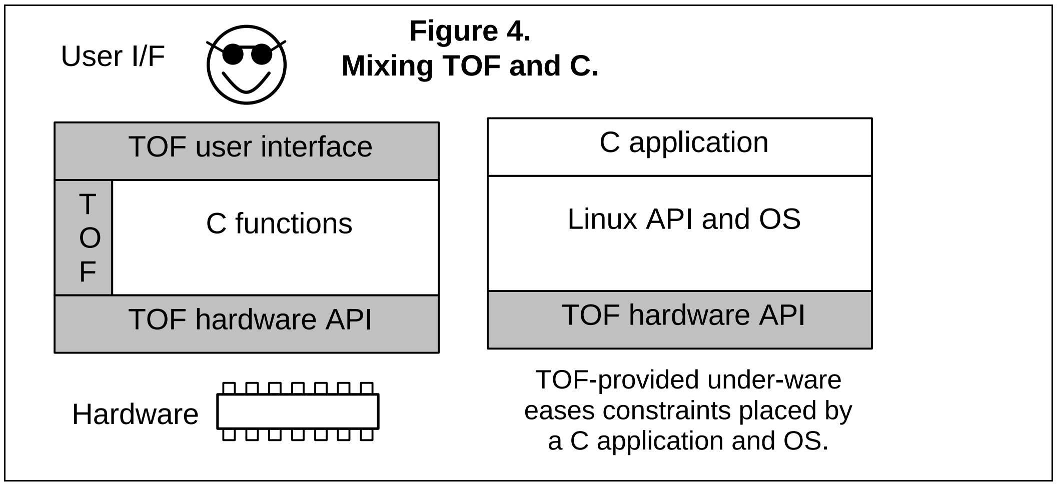
Communication stacks are a good example of IP that comes in C. The stack can sit atop an API provided by TOF, which allows add-on hardware to connect itself to the bottom of the communication stack.

Most TOF functions are inherently reentrant. Many of Diab's C functions are also reentrant. This makes it possible for Forth to call C, which calls Forth, which calls C, etc.

The same techniques apply to other C implementations and other target processors. Refer to the TOF documentation and the literature that comes with the C to determine register usage and how to pass parameters.

Sandwiching a C application between Forth-based abstraction layers as shown in Figure 4 allows plug&play add-on hardware to meld itself onto an application, although not as elegantly as is possible with a pure Forth application.

Figure 4.
Mixing TOF and C.



TOF is meant for embedded systems, not as a replacement for Open Firmware on the desktop. TOF could be used as the basis for a kind of BIOS for Linux to run on, but it would only be useful if add-on hardware contains TOF boot code. Off-the-shelf cards won’t, so it probably wouldn’t be worthwhile outside the embedded arena.

An example TOF system

Two evaluation boards have been built: one 8051-based and the other 68331-based.

One or more 20x4 LCD modules connect to the evaluation board in daisy chain fashion using modular telephone cables. These cables form a simple expansion bus that I call the SPIX bus. The SPIX bus is a superset of the SPI protocol and can be implemented in a small CPLD. The SPIX bus is used in the modules to demonstrate a low cost configuration and to avoid the impression that the added functionality could be coming from firmware on board the modules.

Modules on the SPIX bus are daisy chained. Each node has a physical address determined by its position in the chain. To select a particular node, the CPU shifts a node select signal to the desired node, leaving the other nodes not selected. Otherwise, the SPIX is like a classic 3-wire serial bus.

At startup, TOF probes the SPIX bus looking for boot code resident on a serial EEPROM in each module. If it finds some, it feeds it to the evaluator and continues probing until no more modules are found. A word called MY returns or compiles the current node address to let module address itself. For example, if code on node 3 wants to write to its own hardware (which is usually the case) it uses MY to compile the node selector. So, addresses are automatically resolved at boot time.

The CPU periodically polls the bus to allow hot plugging. Add or remove a module, and the system reboots and reconfigures itself in a couple of seconds.

The boot code on the LCD modules is processor independent. They can be plugged into either board, and the boot code from each module will be compiled to machine code and installed. At startup, the CPU board will:

- Evaluate primary boot code from the on-board 24C64 serial EEPROM.
- Evaluate boot code resident in each module on the SPIX bus.
- Evaluate secondary boot code from the on-board 24C64 serial EEPROM.

In a typical demo, the EVB gets its output redirected to the LCD and then the secondary boot program displays a message.

The following listing contains some of the boot code for one of the LCD modules. The boot code starts with a boilerplate, defines a simple terminal emulator, and redirects some text output words to the LCD module's driver words.

The evaluator verifies the boilerplate before accepting a block of tokenized code. It verifies the checksum and makes sure sufficient code and data memory are available. In this case, the code calls for 256 cells of code and 64 cells of data. If the code isn't accepted, the loader spits out a message to the terminal. In our case, the terminal is either a section of RAM used as a virtual console or the LCD module.

The device type byte in the boilerplate is used by the evaluator to allow generic modules to load onto specialized hardware. The evaluator will accept either 0 or a specific value chosen by the OEM. Specialized add-on modules will have a nonzero device type matching that of the equipment it's supposed to plug into.

```
\ Boot code for a character LCD module on the SPIX bus.     8/05/01 Brad Eckert

\ This code reroutes terminal output to the LCD display. At boot time, the
\ evaluator loads a driver for the LCD module and executes code that links it
\ into the system.

host decimal homeorder building         \ building a ROM image
also core also system definitions
{{ 0 org }}                             \ start at location 0
0xC0 c, 0xDE c,                         \ standard TOF key
0x00 c,                                 \ generic device type (see LOADER.FF)
1 c, 0 c, 0 c, 0x40 c,                  \ minimum memory requirements
tokenizing {{ 0x1000 >token# }}         \ relative TOKENIZED CODE
program autoexec                        c( tokenized boot program)
\ _end_of_boilerplate________________________________________________________

\ Character LCD interface -- tokenized code starts here.

{{ hex }}

01 constant lcd_rs      \ Wire positions (2^bit#)
02 constant lcd_e

create LCDinit  4 c,    \ LCD initialization string: length
                28 c,   \ 4-bit data
                06 c,   \ increment cursor
                0E c,   \ display on, cursor on
                01 c,   \ Clear display

: >lcd          ( -- )          my x_node 2 x_port ;
: lcd_port!     ( c -- )        80 or x_byte! ;
```

[=== a bunch of stuff snipped ===]

```
: myemit        ( c -- )                        c( process incoming character )
        case    0A of endof                     \ ignore LF
                0D of mycr endof                \ CR
                0C of mycls endof               \ ^L = cls
                dup lcd_data 'char c!           \ display & save character
                col incr                        \ bump column counter
                col @ #cols >= if mycr then     \ handle line wrap
                dup
        endcase ;

\ _end_of_terminal_emulator____________________________________________________

' myemit ' emit rebind   \ redirect terminal output to the LCD
' mycls  ' cls  rebind
' myxy   ' at-xy rebind
init-lcd                 \ initialize the LCD module now

end                                             \ end this block of code
\ END also resolves the length and checksum laid down by PROGRAM.
host

bsave lcd_emit.sb        \ The LCD's boot EEPROM will contain this file.

\ This ASCII source compiles to 486 bytes of tokenized source and boilerplate.
```

Summary

Tiny Open Firmware has several features that distinguish it from other embedded tools and other Forths. TOF provides:

- Simple addition of new hardware through a standard mechanism.
- A means to upgrade firmware in the field.
- Forth’s interactivity and ease of debugging.
- A relatively small memory footprint.

Tiny Open Firmware brings self-installing plug-and-play hardware to small embedded systems. A TOF implementation is small, typically under 32K for 68K/Coldfire and 20K for 8051 processors. Its efficient late-binding mechanism renders all ROM-based routines patchable and enables rapid compilation of processor-independent add-on code. Run-time compilation of add-on code simplifies applications whose configurations will change as customers modify their systems.

Allowing add-ons to equipment in the field protects the customer’s investment in existing equipment. TOF also provides an upward compatibility path that protects add-on hardware from obsolescence. Plug-and-play becomes simple and even fun, resulting in a better user experience.

Further reading

http://www.forth.org/literature.html is a rich source of further reading about Forth.

Two good Forth books are published by Forth, inc. www.forth.com

1. *Forth Application Techniques* is a good tutorial book.

2. *The Forth Programmer’s Handbook* is a good reference for programmers experienced in other languages.

http://www.tinyboot.com contains resources for Tiny Open Firmware and Forth.