Firmware Studio User's Manual

Ultra-flexible firmware for embedded microprocessors and systems-on-a-chip

Author: Brad Eckert

brad@tinyboot.com

August 2001

Contents:

Appendix M. Multitasking

Cooperative multitasking in TOF.

Welcome to Firmware Studio

Firmware Studio is a firmware development system for microcontrollers. A Windows95/NT based program provides the user interface and development environment, while firmware on the target device provides a processor-independent virtual machine. A serial cable or other communication link provides transparent access to the target device for testing.

Firmware Studio and Tiny Open Firmware, or TOF are based on the ANS Forth programming language. But isn't Forth a dead language that came and went back in the '80s? Not quite. Forth may have declined in use since then, but the Forths of today are far better that those available even a decade ago. Forth has been steadily pushed into the embedded realm, where it squeezes the most functionality out of resource constrained systems.

Forth is a programming tool that masquerades as a language, not the other way around as is the case with C, C++ and Java. This distinction lets us mold our tools (and language) to our intended application, not force-fit our application to the constraints of a particular language. Without these shackles, the magic begins.

This system was inspired by the IEEE1275 standard for boot firmware, commonly known as Open Firmware (OFW). TOF is basically the Forth part of OFW pared down to suit small embedded systems. The object oriented device tree part of OFW is absent, replaced by a much simpler pool of methods. Some encapsulation is possible, but that's up to you.

TOF provides many of the benefits of IEEE1275 Open Firmware (OFW), but is geared toward small embedded processors rather than desktop machines. Its emphasis is on flexibility and ease of debugging. Whereas a 32-bit system is recommended for supporting OFW, Firmware Studio easily fits in 8-bit systems. The ROM footprint is typically 10 to 16 Kbytes.

The ability to patch ROM code is a natural byproduct of Open Firmware's implementation. This feature is especially useful in the kinds of applications suited for Firmware Studio. Almost any kind of bug can be patched, taking most of the risk out of committing code to masked ROM.

A very efficient threading scheme adds an extra level of indirection (through a table in RAM) to allow patches to the ROM code. This "binding table" usually consists of jump instructions. Subroutine calls jump into the binding table, which then jumps to the code to be executed.

At startup, the binding table is initialized from ROM. After that, a program called an evaluator loads tokenized source code from a storage device (such as a serial EEPROM) to link patches and added features into the system. Essentially, new code is spliced into the system at boot time. The basic firmware, bugs and all, is always in ROM. The ROM never changes. No parts are scrapped. A small amount of Flash memory or EEPROM holds just the fixes and add-ons.

The RAM-based binding scheme is the keystone of Open Firmware. The evaluator compiles tokens (numeric equivalents of keywords) into machine code in one pass. Compilation is simple and straightforward, giving high throughput. During evaluation, tokens may execute or be compiled as they are encountered.

Compilation throughput in thousands of tokens per second is roughly 1 on an 8031, 50 on a classic 68000, and 100 to 1000 on a Coldfire. This JIT compilation mechanism allows functionality to be archived as compact bytecode and compiled into RAM as needed.

Having an efficient on-board compiler offers new possibilities for product differentiation. Embedded plug-and-play capability and end-user programmability can be exploited to gain an edge in the marketplace and to extend the useful life of products.

The ability to alter the run-time system at any time is a powerful debugging feature. Code on the target device can be tested and modified during live operation. The effects of code changes can be seen immediately. There is no need to stop the system, load the changed code, restart the system and bring the system to the point at which the new code is supposed to have an effect.

Like OFW, Firmware Studio provides a clean, simple solution for plug-and-play add-on peripherals. Add-on hardware may contain boot code that links the new hardware into the system. Hardware that is built to augment a specific application can have firmware with knowledge of that application so that it modifies the application to add the desired functionality. All this can be done with almost no forethought of what possible add-ons may appear in the product’s life cycle.

OFW was designed to be processor independent and to facilitate arbitrary hardware add-ons in the field. Firmware Studio can be processor independent, but the add-ons it supports will generally be application dependent and supplied by the OEM. Since the OEM will have intimate knowledge of the system, Firmware Studio lets you modify things like interrupt vectors and allows machine code to be embedded in the tokenized boot code.

Firmware Studio is based on ANS Forth. Win32forth, a freeware WindowsNT Forth system, compiles the source, which is also free. Firmware Studio combines the productivity advantages of Forth with highly interactive debugging tools.

Firmware Studio presently supports the AVR, 8051 and 68K/Coldfire families of processors. The 8051 and 68K/Coldfire versions have passed the validation suite. All have many subroutines written in assembly for speed.

As of Q2 2000, Firmware Studio runs on Coldfire (MCF5307), 68332 and 8051 evaluation boards. The development tools and debugger (but not TOF) also run on Atmel’s STK200 and STK300 (AT90S8515 and ATmega103) evaluation boards.

Porting to a new processor takes some work, but the effort isn’t unreasonable since the source for the assemblers, disassemblers and builders of other processors is present. Besides, nothing brings you up to speed on a new processor architecture like porting it to Firmware Studio.

Why Firmware Studio?

Microprocessors are small electronic circuits that control many facets of everyday life. Automobiles use them for engine, drive train and climate control. Microwave ovens use them to control cooking. Airplanes use them for navigation and flight control. NASA sends them into space. In today’s marketplace, almost everything that uses electricity has a microprocessor in it.

Microprocessors operate from a stored set of instructions called the operating program. Changing this program changes the behavior of the system. This versatility is what makes them so popular. A microprocessor produced by a given manufacturer can be designed into a wide array of products. An embedded microprocessor is one that performs a dedicated function in a product rather than performing general purpose computing: in other words, a computer that doesn’t look like a computer.

The operating program that controls a microprocessor is known as firmware. A programmer creates firmware by writing a human-readable program (source code) and converting it to machine-readable form (object code) using a compiler. In what is known as the "edit-compile-test-debug cycle", the programmer modifies the program, compiles it, tests it, devises solutions to problems that were found and makes more modifications. When the program has all of the desired features and passes every test the manufacturer can think of, it is considered finished.

The firmware of a microprocessor contains a set of features designed to address the customers’ needs. If a customer is using the product in an application that wasn’t foreseen by the manufacturer and has unique needs, the customer must either live without the desired features or get the manufacturer to spend valuable engineering resources on firmware maintenance.

In embedded microprocessor designs, every penny counts. The lowest cost memory for firmware storage is masked ROM (Read Only Memory). Because of high setup costs, firmware changes in a masked ROM part are very expensive. Although firmware is tested thoroughly before being committed to masked ROM, today’s increasingly complex firmware still presents an element of risk. Other memory technologies can be reprogrammed, but they are more expensive. In-circuit programmability offers even more flexibility, but at the cost of extra support circuitry.

Firmware Studio is a Windows95/NT-based development environment designed to improve the firmware development cycle, provide a means for end-user programmability, and allow firmware fixes and upgrades even to masked ROM.

The main problem with the usual edit-compile-test-debug cycle is that the system must be stopped whenever a change is made. After the change is made, the system must be restarted and brought to the point at which the change is supposed to have an effect. Firmware Studio allows live debugging that is analogous to "hot-swapping" of hardware modules. Just as troubleshooting a car engine is often easier while the engine is running, debugging live firmware quickly reveals the effects of firmware changes.

There is also a provision for single stepping through code to observe its operation. The code under test has its own execution thread, allowing the target system to run at full speed while testing is underway.

Firmware Studio features diagnostic tools for monitoring a live system, making it easy to see what the firmware is doing. A small monitoring program that coexists with the rest of the firmware provides a data link to a host PC running Firmware Studio.

The compiler used by Firmware Studio allows incremental compilation, which means that only a small part of the program needs to be compiled for each test. When used in conjunction with the debugging tools, this reduces compile time and greatly compresses the development cycle.

The code that runs on the microprocessor is loosely based on IEEE1275 Open Firmware (OFW). While OFW was designed primarily for plug-and-play compatibility of workstation add-on cards, Tiny Open Firmware (TOF) was mainly designed for easy maintenance of firmware in embedded systems.

TOF allows the manufacturer to build an application language that the end user, as well as the manufacturer, can use to add more functionality. The internal workings of the application language, which may be proprietary, needn't be revealed to the end user.

Using Firmware Studio, the end user (or the customer's MIS department) can tailor the firmware to his or her own needs. So, the manufacturer needn't spend engineering resources on the endless code maintenance caused by requests for special features. Code maintenance becomes more the customer's responsibility, letting the OEM off the hook.

TOF allows sections of code to be replaced anytime, even though they are in ROM. Upon power-up, these modifications are loaded and compiled from a small memory device and linked into the system. The manufacturer or the end user can place code for additional features in this memory. New functionality can be loaded anytime, from any source, quickly and transparently. This flexibility comes at the expense of a very small reduction in performance and a relatively small amount of extra RAM (Random Access Memory).

Firmware Studio contains a tokenizer that converts source code to tokenized form. The tokenizer is a kind of pre-processor that strips out comments and converts text instructions and data into a compact numeric representation. The tokenized code is portable among processors that implement the same virtual machine.

When tokenized code is loaded, it is translated into machine code and stored in RAM. This code may be removed from RAM at will and replaced by code for other features. Since tokenized code is very compact, many features can be packed into a small memory device. A good analogy is a PC with a compressed hard disk. Programs are loaded and removed from RAM at will, and their stored format is much smaller than the equivalent executable code. The difference is that PC programs are pre-compiled so they are processor dependent.

TOF needs program RAM for its dynamic linking mechanism and space for added features. Depending on the processor and how big the application is, 2K to 4K bytes is about the minimum useful amount of additional program RAM. Most of today's microcontrollers have very limited built-in RAM. For systems that would otherwise get by without an external RAM or a larger external RAM, Firmware Studio represents an additional expense. However, the complex and change-prone systems that Firmware Studio was made for are likely to have sufficient extra resources. As feature sizes shrink, the extra on-chip RAM will become less of an issue.

The proliferation of systems-on-a chip and the emergence of embedded DRAM in the late 1990's produce favorable conditions for Firmware Studio. As systems become ever more integrated and customized (more features packed into fewer components), the need for changeable firmware will grow. On-chip Flash memory is currently the most popular means of providing this flexibility, but it is expensive. Even so, the strong market demand for these devices illustrates the need for flexible firmware. By including some extra program RAM on-chip, Firmware Studio would allow the use of conventional masked ROM and provide powerful debugging and enhancement features.

In many cases, Firmware Studio provides the lowest cost means of in-system programmability.

Firmware Studio is all about flexibility. It allows rapid and efficient development of firmware, enables end users to make their own customizations, and allows field firmware upgrades without ROM changes.

What chips are Firmware Studio well suited for?

Firmware Studio is a good development environment for almost any chip. For example, I write all assembly for small AVR projects. The debugging tools are very useful even if little code is written in Forth. The tools currently support 8051, CF/68K and AVR families. It can be ported to others.

Many of today's microcontrollers are designed to run 'C' code efficiently. Since 'C' means stacks and 16-bit (or wider) arithmetic, they also run Forth efficiently. Self-modifying code is generally frowned upon and 'C' isn't extensible so most micros have very little, if any, program RAM.

Tiny Open Firmware needs a small amount of program RAM, typically 4 to 8 Kbytes. In a microprocessor-based system, this RAM requirement usually isn't a problem. For single-chip designs based on off-the-shelf microcontrollers, the amount of on-chip program RAM is often too little to be practical. This is changing. For example, many modern DSP chips have lots of on-chip RAM. An extra benefit of Firmware Studio here is that the compactness of the whole application can alleviate the need for off-chip memory.

Motorola's standard-product M-core chip, the MMC2001, has 32 Kbytes of RAM on-chip, providing a comfortable fit. Mitsubishi makes the MR32 (which has 2M of embedded DRAM) and the M16C. An Atmel AVR 8-bit RISC would work if it had program RAM. Atmel's FPSLIC is an FPGA/AVR combo that does have RAM-based program memory which would support TOF. Triscend makes an 8052 derivative with sufficient SRAM.

Mitsubishi's M16C 16-bit processor is the best low cost choice for single-chip operation. In addition to having sufficient SRAM, it has a JSRI instruction that can take its jump address from the binding table. TOF can span much more that 64K of program space using 16-bit cells, since XTs aren't CFAs. Too bad there's no M16C port yet.

Firmware Studio can be made to provide powerful debugging tools for any system. On many systems, it reduces the cost of components by drastically cutting the size of the upgradeable boot device. If it makes you add an extra RAM chip then the savings could evaporate, but a large system will usually be able to spare the RAM.

Rev 2.1x of Firmware Studio supports the AVR, 8051 and 68K/Coldfire families. The AVR version does Forth, but not TOF. For other chips, you'll have to do some work. Porting to a new processor takes some doing. Usually I write a disassembler and an assembler, then I write the debugger and a bunch of kernel words in assembly. Most target chips have a cheap or free simulator, and that helps get the debugger code working. Then, I get target hardware up and running with the debugger and use the debugger to test each kernel word. Finally, I run a test suite through the evaluator.

How does Tiny Open Firmware (TOF) work?

The firmware created by Firmware Studio is based on subroutines. Subroutines are small sections of code that perform a particular function. Each subroutine is essentially an extension of the microprocessor's language. As new subroutines are added, more functionality becomes available. Working at a higher level of abstraction, more can be done with less code. In a carefully designed system, functionality grows exponentially with code size. An application language is a group of subroutines designed to address a particular class of problems.

The firmware resident in the target system is similar to IEEE1275 Open Firmware but is geared more toward embedded applications than desktop computers. I call it Tiny Open Firmware, or TOF. TOF is the component of Firmware Studio that resides in a ROM on the target processor. It presents a processor independent virtual machine to the PC-based host.

TOF uses a group of subroutines collectively known as Forth. Forth is particularly well suited to TOF, being heavily subroutine based and inherently reentrant. This time-proven language uses an efficient set of subroutines that pass parameters via a dedicated data stack. To a software designer, Forth is a stack-based language without types. To a hardware designer, Forth is a large collection of small subroutines managed by a simple interpreter. In Forth, subroutines are called *words*.

When managed by the interpreter, this collection of words constitutes both a run-time virtual machine and a compiler. There is no wall between the compiler and the rest of the language. Extending the language (by defining new words) can extend the compiler.

The flexibility of TOF revolves around a special lookup table in RAM. All references to any given word go through this table, which is basically an array of pointers. This table is referred to here as the binding table. The binding table is formed from jump instructions so as to add very little run-time overhead. When a subroutine call is compiled, the destination is somewhere in the binding table. Since most of the subroutines are resident in ROM, these pointers initially point to code in ROM. If one of the subroutines needs to be changed, the replacement code may be placed in RAM and the corresponding element in the binding table modified to point to the new code.

TOF compiles tokenized source code in a manner similar to the IEEE1275 Open Firmware used in most of today's workstation firmware. A boot device or other source contains processor-independent, tokenized source code. An evaluator fetches tokens from a boot device or other input source. For each token, the evaluator either executes the word associated with the token or compiles a call to that word.

The code for each word is preceded by a header byte containing various flags. The evaluator uses the *immediate* flag in the word's header byte and a Boolean variable called *state* to determine whether to compile or execute a token. If *state* is true, the evaluator is in compile mode and incoming tokens will be compiled unless their *immediate* flag is set. If *state* is false, tokens are executed.

For each token, the evaluator needs to extract various pieces of information. A token is simply a number, unique to one word and represented by one to three bytes. The most commonly used words are represented with a single byte. When the evaluator fetches this number, it uses it to compute an address in the binding table. If the token is to be compiled, the evaluator compiles a subroutine call to this address. If the token is to be executed, the evaluator calls this address. In order to get header information (such as the *immediate* flag), the evaluator gets the word's execution address from the binding table. Since the header byte is immediately before the word, subtracting one yields the address of the header byte.

When a token is compiled, the code is usually a call into the binding table. Sometimes the word performs a trivial action that is economical to do with in-line machine code. In this case the evaluator compiles this inline code instead of a subroutine call. If the token number being compiled is the same as the token number of the current definition, a recursive situation arises which may not be desirable. When this happens, the evaluator bypasses the binding table by fetching the destination address from the binding table and compiling a call directly to this address.

Immediate words usually perform some compilation action. For example, IF and THEN are immediate words that compile a control structure. Examples of immediate actions include pushing a number onto the stack, ending compilation, compiling strings and literals, etc.

TOF uses a small non-volatile memory such as serial EEPROM on the target board to hold tokenized boot code. Upon startup, the firmware initializes itself and tries to load boot code from this device. If there is a hardware error or the checksum of the code is wrong, the code isn't loaded. If broken code is somehow placed in the boot device such that the system crashes upon startup, there must be a provision for inhibiting the loader so that a normal startup occurs. This could be as simple as grounding the data line of the serial EEPROM to force a hardware error.

Tokenized code may be loaded from any device. For example, devices on a multidrop serial bus or Ethernet link could be loaded with new code anytime from the bus master.

Since the evaluator operates in a relatively simple and straightforward manner, it has very high throughput. Compilation occurs very quickly, allowing new code to be loaded on an as-needed basis. Many features can be packed into a memory device in tokenized form. When one of these features is needed it can be loaded into RAM, executed and removed without any significant delay from the loading process.

Tokenized code is converted to fast-executing native code. Compared to its equivalent executable code, the tokenized code is quite compact. For example, the 8051 code shown below is 2.6 times as big as its tokenized source. Most 16-bit and 32-bit processors use longer instructions than the 8051, so the resulting code for these processors would use even more memory.

The evaluator evaluates token numbers between 1 and 8191. The most-used tokens are 1-byte values between 32 and 255. Values between 1 and 31 cause a second byte to be fetched and combined with the first to form a number between 256 and 8191. If the first byte is zero, two bytes are fetched to form all remaining 16-bit values.

Token values between 4096 and 8191 are treated as relative tokens. At the beginning of each evaluation, a variable called tokenbase is set to point to the lowest undefined binding table entry. When a relative token is processed, 4096 is subtracted and tokenbase is added so that the token gets mapped onto a free area of the table. The highest absolute token value is tracked in order to set tokenbase the next time around.

Relative tokens allow multiple patches to be loaded without colliding with each other’s token assignments.

Table 1 shows the output of an evaluator for the 8051 family of microprocessors. The principle of operation is the same for other processors. This particular program lists prime numbers. It is represented by 25 bytes of tokenized code, which compiles to 65 bytes of 8051 executable code. Immediate tokens are italicized.

Since parameters are implicitly passed on the data stack, no extra code is needed for parameter handling such as moving data between registers. The object code consists mostly of subroutine calls to the binding table.

Table 1 is followed by a line-by-line description of what happens when this tokenized code is evaluated.

Table 1. Executable 8051 code created from tokenized source code.

Ref	Source	Tokenized	Sample 8051 Disassembled Object Code
#	Text	Source	
1	: PRIMES	EA 02 0E	1387 08
2	*DO*	F6	1388 12FEA0 PRIMES: LCALL 0xFEA0(xt:32) (%DO)
			138B 0213C7 LJMP 0x13C7
3	2	30	138E 12FE70 LCALL 0xFE70(xt:48) 2
4	*BEGIN*	F1	
5	I	28	1391 12FE88 LCALL 0xFE88(xt:40) I
6	OVER	36	1394 12FE5E LCALL 0xFE5E(xt:54) OVER
7	/MOD	B8	1397 12FCD8 LCALL 0xFCD8(xt:184) /MOD
8	PLUCK	37	139A 12FE5B LCALL 0xFE5B(xt:55) PLUCK
9	>=	5A	139D 12FDF2 LCALL 0xFDF2(xt:90) >=
10	OVER	36	13A0 12FE5E LCALL 0xFE5E(xt:54) OVER
11	AND	4E	13A3 12FE16 LCALL 0xFE16(xt:78) AND
12	*WHILE*	F4	13A6 12FEA9 LCALL 0xFEA9(xt:29) (%IF)
			13A9 0213B3 LJMP 0x13B3
13	DROP	11	13AC 12FECD LCALL 0xFECD(xt:17) DROP
14	1+	3D	13AF A3 INC DPTR
15	*REPEAT*	F5	13B0 021391 LJMP 0x1391
16	NIP	32	13B3 08 INC R0
			13B4 08 INC R0
17	IF	ED	13B5 12FEA9 LCALL 0xFEA9(xt:29) (%IF)
			13B8 0213C1 LJMP 0x13C1
18	I	28	13BB 12FE88 LCALL 0xFE88(xt:40) I
19	.	02 09	13BE 12F8E5 LCALL 0xF8E5(xt:521) .
20	*THEN*	EE	
21	*LOOP*	F8	13C1 12FE94 LCALL 0xFE94(xt:36) (%LOOP)
			13C4 02138E LJMP 0x138E
22	;	EB	13C7 22 RET

1: The word associated with **:** begins a new definition. It switches from interpret mode to compile mode by setting *state*. It also lays down a header byte, which initially contains nothing. It saves a pointer to the beginning of executable code (1388) and the current token number in a variable called *last*.

2: **DO** compiles a call to its run-time action, **(%DO)**. It also lays down a forward branch, **02 00 00** and saves the address of this code, 138B, on the stack so that the branch can be resolved later by **LOOP**. When **(%DO)** executes, it adds three to its return address so that the code at 138B is skipped.

3: Since **2** is an often-used literal, it has its own word whose token number is 48. The evaluator compiles a call to the 48th element in the binding table.

4: **BEGIN** leaves the current address, 1391, on the stack for later use by **REPEAT**.

5: The evaluator compiles a call to the 40th element in the binding table.

6 .. 11: The evaluator compiles calls to various elements in the binding table.

12: **WHILE** compiles a call to its run-time action **(%IF)**, lays down the unresolved forward branch **02 00 00** and saves the address of this code, 13A9, on the stack so that the branch can be resolved later by **REPEAT**.

13: The evaluator compiles calls to the 17th element in the binding table.

14: The evaluator lays down machine code that is the equivalent of **1+**. In this implementation, it does this by seeing that 1+ has its *macro* flag set and copying **1+**'s code.

15: **REPEAT** resolves the address of the LJMP compiled by **WHILE** (step 12) and compiles a branch back to the address left by **BEGIN** (step 4).

16: The evaluator lays down machine code that is the equivalent of **NIP**.

17: **IF** compiles a call to its run-time action **(%IF)**, lays down the unresolved forward branch **02 00 00** and saves the address of this code, 13B8, on the stack so that the branch can be resolved later by **THEN**.

18: The evaluator compiles a call to the 40th element in the binding table.

19: The evaluator compiles a call to the 521st element in the binding table.

20: **THEN** resolves the address of the LJMP compiled by **IF**.

21: **LOOP** compiles a call to its run-time action **(%LOOP)**, compiles a branch back to the address left by **DO** (step 2) and resolves the forward branch compiled by **DO**. When **(%LOOP)** executes, it adds three to its return address (when the loop is finished) so that the code at 13C4 is skipped.

22: **;** ends a definition. It compiles an exit by compiling a RET instruction or converting the last LCALL to a LJMP. It uses the information in the variable *last* (step 1) to modify the binding table. This word's execution address is stored at element 0x20E in the binding table. All words that reference token 0x20E will now use the updated code.

For comparison, here is a disassembly of the 68000 version of PRIMES, which compiled to 110 bytes (about 1:4 decompression). Note that many words are inlined.

```
00022222 4EB90004F77E PRIMES: JSR 0x4F77E(xt:363) (%DO)
00022228 60000064             BRA 0x2228E
0002222C 4EB90004FE32         JSR 0x4FE32(xt:77) 2
00022232 2B07                 MOVE.L D7,-(A5)
00022234 2E17                 MOVE.L (A7),D7
00022236 2B07                 MOVE.L D7,-(A5)
00022238 2E2D0004             MOVE.L 4(A5),D7
0002223C 4EB90004FC0A         JSR 0x4FC0A(xt:169) /MOD
00022242 2B07                 MOVE.L D7,-(A5)
00022244 2E2D0008             MOVE.L 8(A5),D7
00022248 BE9D                 CMP.L (A5)+,D7
0002224A 5FC7                 SLE D7
0002224C 49C7                 EXTB.L D7
0002224E 2B07                 MOVE.L D7,-(A5)
00022250 2E2D0004             MOVE.L 4(A5),D7
00022254 CE9D                 AND.L (A5)+,D7
00022256 2007                 MOVE.L D7,D0
00022258 2E1D                 MOVE.L (A5)+,D7
0002225A 4A80                 TST.L D0
0002225C 6700000A             BEQ 0x22268
00022260 2E1D                 MOVE.L (A5)+,D7
00022262 5287                 ADDQ.L #1,D7
00022264 6000FFCC             BRA 0x22232
00022268 588D                 ADDQ.L #4,A5
0002226A 2007                 MOVE.L D7,D0
0002226C 2E1D                 MOVE.L (A5)+,D7
0002226E 4A80                 TST.L D0
00022270 67000012             BEQ 0x22284
00022274 2B07                 MOVE.L D7,-(A5)
00022276 2E17                 MOVE.L (A7),D7
00022278 4EB90004FB1A         JSR 0x4FB1A(xt:209) (.)
0002227E 4EB90004FB2C         JSR 0x4FB2C(xt:206) TYPE.
00022284 4EB90004F772         JSR 0x4F772(xt:365) (%LOOP)
0002228A 6000FFA0             BRA 0x2222C
0002228E 4E75                 RTS
```

Firmware Studio provides a ROM builder and a tokenizer. The ROM builder is a compiler used to create processor-specific executable code for the target device. It converts Forth source code to machine code and builds an initialization table that is used to initialize the binding table. This code may be programmed into a ROM and run on the target processor. Once a ROM image is built, the token list can be saved to a file. The tokenizer uses this token list to determine the token values associated with various words. By supplying this file, the OEM enables the end user to customize the application without the need to supply proprietary source code.

The tokenizer is relatively simple. It parses space-delimited strings from the input stream (usually a file) and either performs a special action or compiles the token number associated with that string. Special actions include numbers, IF, THEN, **:**, etc.

For example, **:** gets the name of the definition from the input stream and adds it to the token list. It also lays down (appends to the ROM image) the token number for **:** and the token number for the new definition.

When a word is not recognized, it is converted to a number. If it isn't a valid number, the tokenizer quits with an error message. Numbers are compiled by laying down the token number of (LIT) and the number. (LIT) is an immediate word that gets the number from the input stream. When the evaluator executes (LIT), (LIT) either pushes the number onto the stack or compiles code to do so, depending on *state*. There are several versions of (LIT), used with numbers of different magnitudes. (LIT8), for example, takes an 8-bit number from the input stream and sign extends it before using it.

The tokenizer has no knowledge of the target processor. It only knows things like "the token number for OVER is 54". If the token assignments are the same for several different processors such that they all present the same virtual machine, the tokenized code will run on all of them.

This is the idea behind IEEE1275 firmware, which provides plug-and-play capability for workstation plug-in cards. The driver for each card is loaded from a small on-card boot device. Achieving total portability of tokenized code for TOF is possible, although the diverse nature of small embedded systems works against this.

For the 8051 processor, there will be roughly a 1:2 ratio between the size of tokenized code and its executable equivalent. Also, the virtual machine (kernel and evaluator) requires about 10K bytes of ROM. Unused parts of the VM can be removed to shrink the footprint to under 7K. The missing parts could be compiled into RAM at run time. In theory you could compile little applications on demand. The evaluator on an 8051 is amazingly slow, about 1K tokens per second. Although the theory is simple, the details gang up on you. This throughput is good for loading patches and drivers for add-on hardware, but would be painful for big applications. A 68K/Coldfire would have no such problem.

Although TOF's virtual machine runs on an 8051 and 68K/Coldfire, it can provide an efficient Forth system for almost any processor. Every new VM inherits the powerful development and debugging tools of Firmware Studio. Porting to a new processor does require some work, of course.

A monitor program (sometimes referred to in Firmware Studio as the backdoor debugger) is resident on the target device. Watching and debugging is done through this monitor program.

The monitor program communicates with the host PC via a serial cable. It accepts one-byte commands from Firmware Studio. For each command it receives, it issues a one-byte response. This 1:1 correspondence eliminates the need for handshaking and interrupts on the target board. The target firmware can service the debugging port (usually a UART) whenever it's not doing anything else. The scheme is very non-intrusive for most applications.

There is a drawback in that Windows has sluggish turnaround response. If you look at the serial data on an oscilloscope, there seems to be a millisecond delay between the reception of a byte and the transmission of the next command. When it comes to communication performance, a pokey old 8051 runs circles around a P166 running Windows.

I have developed a 1-wire debugging protocol called EZbit and an EZbit to RS232 converter. EZbit works using very short pulses. The target board sends pulses out an open-drain I/O pin. At the trailing edge of each pulse, the converter creates an optional pulse. The target's outgoing pulse can be 250 to 6300 nsec wide, with a ‘0’ bit being twice as long as a ‘1’ bit. The EZbit converter tunes itself to your pulse width. It packs and unpacks these bits to form bytes and sends/receives from the host PC at 115K baud. When the target board’s UART is unavailable or non-existent, you can use the EZbit interface to get into the system.

Hardware cost for EZbit on production boards is almost nothing and there are no interrupts and almost no time constraints, allowing EZbit to have the lowest priority of any task in the system.

The commands supported by the background monitor are processor-independent. They allow random, transparent access by Firmware Studio. Thus, Firmware Studio can watch and modify parameters on the target device while the target processor is running, without knowing what kind of processor it's connected to.

Firmware Studio allows for interactive testing of words on the target processor and for loading replacement words. All of this is done live, while the target device is in normal operation.

The host PC talks to the target board via a serial port cable. When you push the "Connect" button or do anything that requires the target, the host sends a 0xE0 byte at 115200 bps out the serial port and listens for a response. COM1 through COM4 are scanned until a target board is found. The target returns a 0xFF at its own baud rate, which the host uses to set its baud rate.

As of release 2.07, Internet support is included. A host PC acts as a gateway to the target board. A server program runs under Windows95/NT on the remote host. At the local end, your PC runs Firmware Studio and talks to the remote target board over the Internet. If you're operating across firewalls, coordinate activity with the appropriate network administrators.

At the remote end, run HOST.EXE. If you're wired, it will tell you its IP address. Write this number down. Press the "Plug" button to connect to the target board. If the target board connects, press the Net button. A 10-second delay is included to allow you to minimize the window. Once it starts waiting for a connection, none of the window buttons (including minimize, close, etc.) will work. This is inherent to the Windows winsock "accept" function, so you'll just have to live with it. Use the task manager to stop the server if you have to.

The following commands are available to set the IP address in Firmware Studio:

IP=	Sets the new IP address. Example: IP= 123.45.67.128
IP?	Displays the current IP address.

To connect over the Internet, run Firmware Studio and use IP= to enter the address of the remote server. Click on Target Internet to select Internet connectivity, then press the "Connect" button or any button that talks to the target.

Getting Started with Firmware Studio

Get the Firmware Studio archive from http://www.tinyboot.com. The directory structure is:

`.\`	Main directory contains executable files and firmware source files
`.\Source`	Source files needed to re-build the executables
`.\Docs`	Documentation: Manual

Unzip the archive to a directory on your PC, such as C:\FF. All executable and source files are present. Firmware Studio may be re-compiled by a public domain Forth system for Windows95/NT, called Win32Forth. It is available at http://www.forth.org in the compilers section.

Create a shortcut that executes **FF.EXE**. After creating the shortcut, right click on it and select Properties. Click the Shortcut tab and Change Icon button and Browse button to choose one of Firmware Studio’s cute little icons.

After starting Firmware Studio, on the left side of the screen you’ll see the search order along with the contents of the data stack. The current vocabulary is in green, and the search order is in blue. There are several modes of operation, many of which place a special vocabulary (colored gray) at the top of the search order.

Forth uses a simple syntax, which consists of keywords separated by spaces. An interpreter parses keywords from an input stream such as the console or a file. Firmware Studio has a number of different interpreters. They are used for normal Forth evaluation, building executable ROM code, tokenizing source code and interactive testing of the target board. Although they differ in function, the principle of operation is the same for all of them. The following table shows the available interpreter modes.

MODE	***Purpose***	***Top Vocabulary***	***Stack Display***	***Destination***
NORMAL	Miscellaneous	---	Host	Image
BUILDING	Compile ROM code	BUILDER	Host	Image
TOKENIZING	Compile user code	TOKENIZER	Host	Image
TESTING	Test ROM code	TESTER	Target	Target
FORTHING	Test user code	TOKENIZER	Target	Target

Destination is where compiled code is ultimately sent. *Display* is which stack information you see on the left side of the console window. Respective shorthand for the above modes is NORMAL, BI, TI, TE and FI. Clicking the "Home" button gets back to normal.

The interpreters are relatively simple, but you need to understand how they work in order to use them (and the search order) effectively. A typical Forth interpreter works as shown below.

A variable called **STATE** determines whether the interpreter is interpreting code or compiling a new definition. **STATE** is manipulated by some compiling words and is generally not accessible to the user. For each keyword parsed from the input stream, the interpreter searches each vocabulary in the search order for the keyword. The following actions are taken:

Found:	The keyword's header contains an **IMMEDIATE** flag. If **STATE** Indicates interpret mode or the **IMMEDIATE** flag is set, the keyword is executed immediately. Otherwise it is compiled, i.e. its run-time semantics are added to the word being defined.
Not Found:	The keyword is converted to a number. If the number contains a decimal, it is considered a double number. In interpret mode, this number gets pushed onto the data stack. In compile mode, the number is compiled as code whose run time action is to push the number onto the data stack. If the keyword is not a number, an error message is generated.

Token structures have their own vocabularies on the host PC. For example, the **CORE** vocabulary contains the word **DUP**, whose run-time action is to load a pointer into a variable called **PFA-local**. When **DUP** is found in the CORE vocabulary and executed, it leaves a pointer in **PFA-local**. Since **PFA-local** is cleared before executing **DUP**, we can tell whether or not **DUP** was a token word. You have to pay attention to the search order, since a search could find the wrong **DUP** (like Forth's **DUP**) and cause an undesired effect.

Whenever a new token is defined, it's **CREATE**d in the Forth dictionary. Various parameters, such as the token number, address, file name, file position, and a comment field are saved for each token.

To get a feel for what's going on, **FLOAD** the file **DEMO51.FF**. This file generates a ROM image suitable for burning into EPROM and running on an 8051 microcontroller. If you prefer, load **DEMOCF.FF** (ColdFire) instead of **DEMO51.FF**. There will be several vocabularies listed on the left side of the screen. Since the normal interpreter will be selected, executing a word in the token list (such as **DUP**) won't have an apparent effect. You can reset the search order by clicking the "home" button (or hit F7), but don't click it yet.

Click the Token Catalog button, the one with the big magnifying glass. A list of all of the tokens in the search order will pop up. To sort the list, click on the field label along the top of the window. Clicking on Name sorts by Name, XT sorts by XT, etc. Press Ctrl-Shift-G to create a glossary file from the catalog window contents and data from each word's source file.

This window lists all words in the search order shown at the left side of the console. To change the search order, use PREVIOUS, ALSO, etc. PREVIOUS removes the top vocabulary of the search order. <vocab> (where <vocab> is a vocabulary like CORE, USER, etc.) replaces the top of the search order. ALSO duplicates the top of the search order. ONLY ALSO CORE clears the search order and places CORE in it. The foreground and background colors have special meanings.

Gray background:	Code is tokenized source.
Cyan background:	Code is executable machine code.

Blue foreground:	Normal word.
Green foreground:	Call-only word. The compiler can't convert call to a jump.
Dark Red foreground:	Immediate word.
Bright Red foreground:	This word is in the watch list.

To disassemble a word, click on its XT or CFA. To browse the source of a word, click on its name. To add a word to the watch list, click on its datatype field.

Disassemble the word **TST1** using the Catalog window or typing **SEE TST1**. The source code for this word is:

```
: tst1 10 begin ?dup while 1- dup >digit emit repeat ;
```

This compiles to the following code:

```
13FB 12FED0 TST1: LCALL 0xFED0(xt:16) DUP
13FE 90000A MOV DPTR, #000A
1401 12FE94 LCALL 0xFE94(xt:36) ?DUP
1404 12FBBB LCALL 0xFBBB(xt:279) (%IF)
1407 021419 LJMP 0x1419
140A 12FE5E LCALL 0xFE5E(xt:54) 1-
140D 12FED0 LCALL 0xFED0(xt:16) DUP
1410 12FD05 LCALL 0xFD05(xt:169) >DIGIT
1413 12FCD5 LCALL 0xFCD5(xt:185) EMIT
1416 021401 LJMP 0x1401
1419 22     RET
```

Notice that the calls are all to the same region of memory. The addresses are all destinations in a table of LJMP instructions, called the binding table. The disassembler calculates the xt (eXecution Token) and label from the address.

The binding table, kept in RAM, provides a lot of flexibility. In the above example, if the address of a word other than **1-** is written to location **0xFE5F**, the behavior of **TST1** and all other words that use **1-** is immediately altered.

The ROM-building mode is entered using the **BUILDING** or **BI** directive.

The "builder" interpreter, used for creating an executable ROM image, compiles to the ROM image instead of the Forth dictionary and places a special vocabulary called **BUILDER** at the top of the search order. This interpreter may be found in the file **TBUILD.G**.

This interpreter works similar to the one described previously, but with a few exceptions. For example, a target word can't be executed on the host since its code is usually for a different processor. An exception is a token whose run-time action returns a literal. These can be simulated since their values are stored with the other token information.

Some compiling words, like **`IF`** and **`THEN`**, are in the **`BUILDER`** vocabulary at the top of the search order. During compilation, they are executed only if they are immediate. Otherwise, the order is searched again but without the **`BUILDER`** vocabulary. This search is expected to turn up a token or a number. The token number is compiled into the ROM image. The target address compiled depends on a flag called **`DYNAMIC?`**. This flag is set using the **`DYNAMIC`** directive and cleared using the **`STATIC`** directive. In dynamic mode, the target address is computed using the xt of the token so as to point to the binding table. In static mode, the target address points to the actual executable code of the token.

With subroutine threading, a definition can be ended using a return instruction. If possible, the last call is converted to a jump to save an instruction and a few clock cycles. This trick can't be used on calls to words that manipulate the return stack. These words are marked with a call-only flag to prevent this.

Much of the kernel is defined in static mode, since it will be needed to initialize the binding table. Words using dynamic binding are useless until the binding table is initialized. The **`BINDINGS,`** directive builds a table of executable addresses. The phrase **`PCREATE INITTABLE BINDINGS,`** is used near the end of the ROM source (see **`END.F51`**) and compiles the binding initialization table to the ROM image. At startup, the data at **`INITTABLE`** is used to create the **`LJMP`** instructions that form the binding table.

The tokenizing mode is entered using the **`TOKENIZING`** or **`TI`** directive.

The "tokenizer" interpreter, used for creating a tokenized bytecode image, compiles to the ROM image instead of the Forth dictionary and places a special vocabulary called **`TOKENIZER`** at the top of the search order. This interpreter may be found in the file **`TTOKEN.G`**.

The tokenizer interpreter is the simplest of them all, since it doesn't distinguish between compile and interpret modes. For each keyword taken from the input stream, it either executes the keyword or compiles bytecode for a number. Most keywords are going to be token header words, which don't change the stack but leave a pointer to header data. If a pointer is left, the token's xt is converted to a bytecode and compiled to the image.

To compile a number, the bytecode for **`(LIT)`** is compiled, followed by the number. On the target, the word **`(LIT)`** is an immediate word that takes its data from the input stream. There are several versions of **`(LIT)`**, such as **`(LIT8)`** and **`(LIT16)`** that take an 8-bit or 16-bit argument and sign extend it if necessary.

There is a token decompiler that displays tokenized code. Try disassembling a tokenized word by clicking on its xt in the Catalog window or by using the **SEE** command. **SEE STARS** decompiles a single word, while **SEE MYPROGRAM** decompiles an entire tokenized program. For example:

```
see stars
1436 : STARS 0 DO STAR LOOP ;

see myprogram
1422 program: checksum=2394 length=24
42 CONSTANT ASTERISK
142F : STAR ASTERISK EMIT ;
1435
1436 : STARS 0 DO STAR LOOP ;
143E END
```

Myprogram is a program structure that contains a byte count and checksum followed by the tokenized code. The word **EVALUATE** in the kernel verifies the checksum before attempting to evaluate any bytecode.

If you want to examine the raw data not shown by **SEE**, click on the IMG HEX button to open the image's hex dump window. Hit F4 to start the dump at 0x1000 and pull the vertical scroll bar down until address 0x1422 (or whatever the address is) is in view.

You can **VIEW** any word using the Winview editor. Shift-F9 enters edit mode, or you can toggle modes using the right-click menu.

All of this can be done without a target board present. You'll need a target board connected to see the debugging features of Firmware Studio. This is described in the next section.

The following chapters are mostly processor specific. Go to the one that covers your target processor.

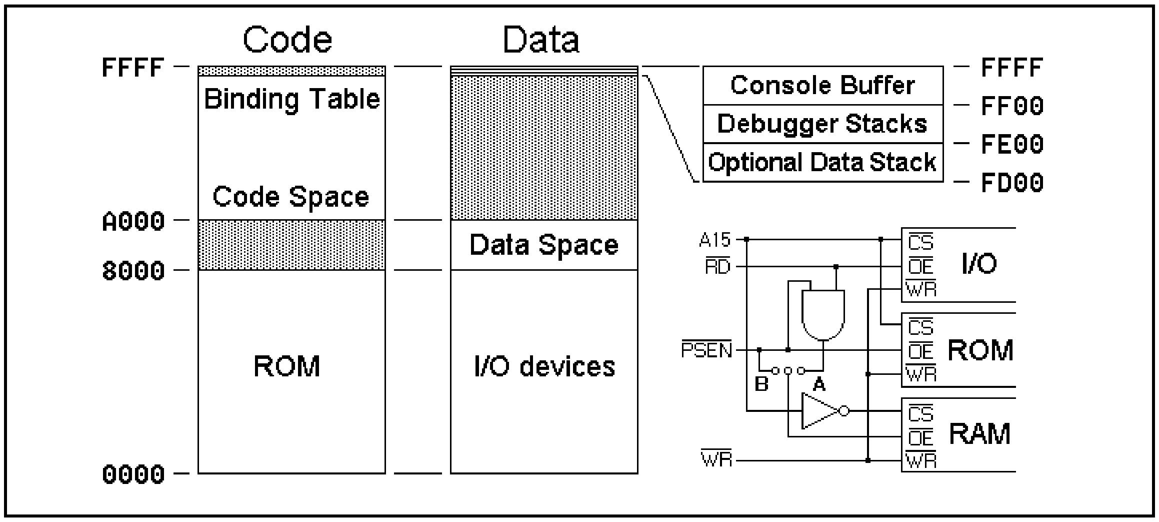
8051-based virtual machine

The 8051 has several address spaces, each operated on by different instructions. Since the 8051 can't write to code memory, some glue logic is needed to overlap the code and data spaces. Figure 2 shows a sample circuit. The jumper block allows two possible configurations. Configuration A makes the RAM readable using both the MOVX and MOVC instructions. Configuration B saves you a gate but only allows the RAM to be read by a MOVC instruction. Position "A" allows you to put the data stack in external RAM while "B" requires the stack to be in internal RAM. Using an external data stack generally slows the system by 20% but frees up IRAM.

The virtual machine presented here is fairly sophisticated and flexible:

- The data stack may reside in either internal or external memory.
- For compatibility reasons, cells may be either 16 bits or 32 bits wide.
- Many Forth primitives are coded in assembly for speed.
- It includes a single stepping machine code debugger that lets you debug code words even while the main program runs at full speed.
- When performing multiply and divide operations, small operands are treated as special cases that execute much faster than full cell-wide operations.
- Variables and values are initialized from ROM upon startup.

Figure 2. Memory spaces of the 8051 virtual machine implementation.



Access to program memory always uses MOVC for read. Access to data memory tests bit 15 of the address and uses MOVX to read from low addresses and MOVC to read from high addresses. This allows peripherals to reside on the data bus between 0000 and 7FFF. The RD line enables

I/O devices and the PSEN line enables the ROM and RAM. So, the 64K data space covers both SRAM and I/O devices.

On the 8051 virtual machine, code and data spaces don’t overlap. It was designed this way to enforce separation of these spaces during the design of the kernel and to allow access to I/O and RAM using the same memory operators.

You can use the hex dump windows to view various memory spaces:

CODE	ROM is between 0000 and 7FFF External SRAM is between 8000 and FFFF
DATA	I/O is between 0000 and 7FFF External SRAM is between 8000 and FFFF
REG	Internal RAM is between 000 and 0FF Special Function Registers are between 100 and 1FF
EE	An optional serial EEPROM is between 0000 and 1FFF (or whatever the size is)

Notice that in code words that access the stack, the instruction MOVY A, @R0 is used. Depending on the flag INTERNALSTACK, the assembler treats this as a MOV A, @R0 or a MOVX A, @R0. See the board configuration section of DEMO51.FF.

Firing up the target board

The target board for **DEMO51.FF** is based on an 8031 microcontroller. This board is easy enough to wire wrap or otherwise fabricate. You'll need a 32K x 8 static RAM, an RS232 transceiver chip, an 8031 or derivative, an EPROM and a 7404 or equivalent inverter. A 2-wire 24C01 to 24C256 serial EEPROM is optional. I recommend 22.11 or 14.75 MHz for the crystal. Specify the crystal frequency and other parameters at the beginning of **DEMO51.FF**.

Code in **BEGIN.F51** selects the timer reload value needed to use the highest possible autobaud frequency. It picks the lowest T1 reload value that will give a baud rate error of less than 2%. The target board's UART must run at 2.4, 4.8, 14.4, 19.2, 38.4, 57.6 or 115.2 kBPS. An arbitrary crystal frequency may be used but you'll probably end up with a low baud rate. For example, a 12.0 MHz crystal will only allow a 4800 BPS rate.

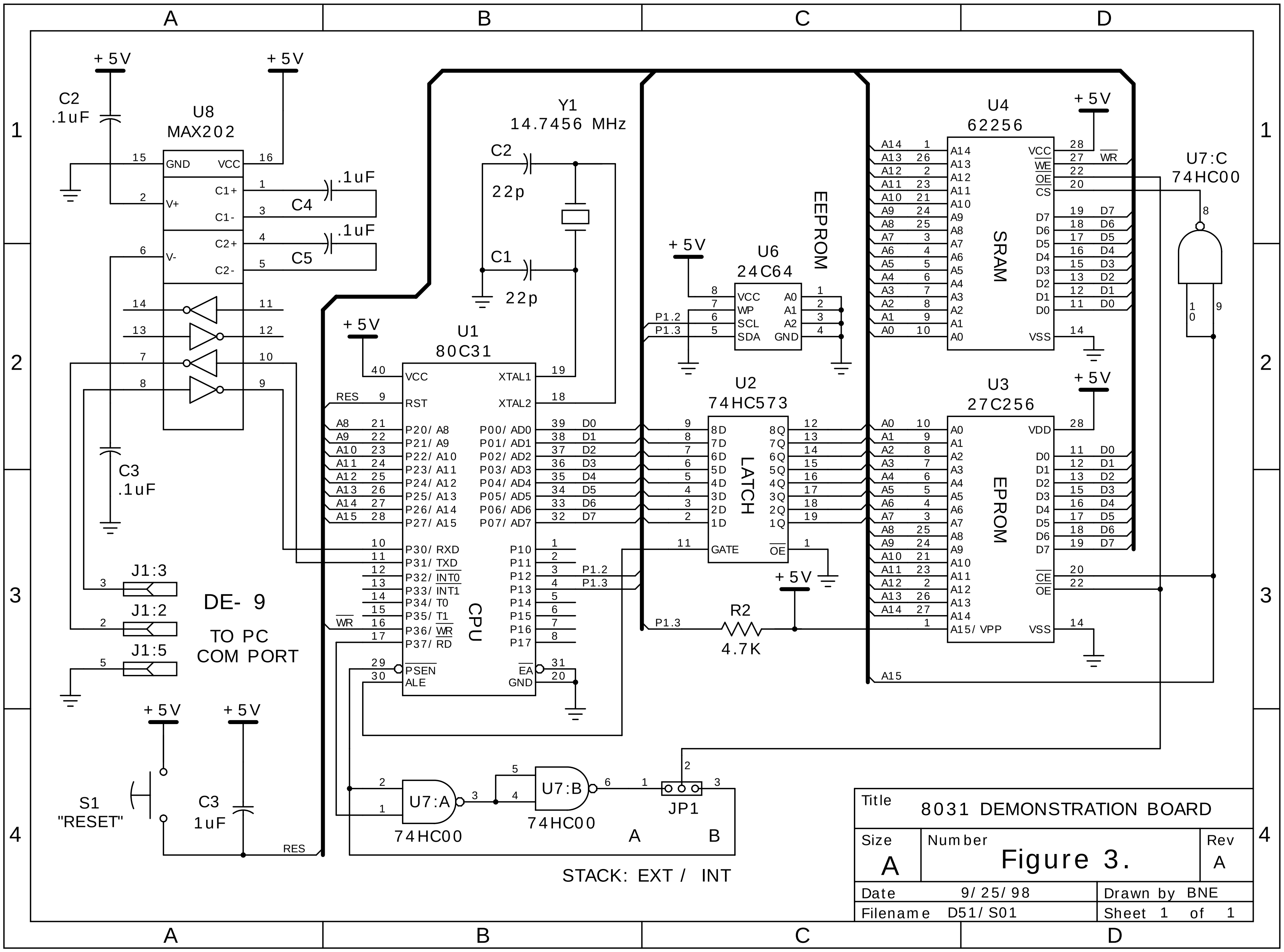
Connect the UART pins to the host's serial port (any port from Com1 to Com4) through the RS232 transceiver chip. Note that most PCs will accept a 0 to 5V swing as valid RS232 input, so for demonstration you could use a 74HC04 as the transceiver chip (100k in series with PC's TXD line) instead of the MAX202.

Firmware Studio's "reset" command resets the target board by dropping the DTR line (pin 4 of the PC's DE9) for ½ second. This is a very useful feature to have in a live debugging environment.

There are all kinds of cheap 8051 development boards available. Figure 3 shows a schematic for a target board. You can wire wrap this board in just a few hours. If you're under a lot of stress, wire wrapping using a hand tool might be valuable for its therapeutic effect. An improved version of this board with SPIX bus support is available from www.tinyboot.com.

The demo program is set up for stacks in external RAM, so put a shorting block in position 'A' of JP1.

If the board doesn't come up, pull the micro and short pins 10 and 11 of the socket together. Run a terminal program to verify that keystrokes echo back through the PC's serial port. Then try it again with the micro plugged in. At the right baud rate (38400) you should see 'R' echoed back whenever you press a key.



Title	8031 DEMONSTRATION BOARD	
Size A	Number Figure 3.	Rev A
Date 9/25/98		Drawn by BNE
Filename D51/S01		Sheet 1 of 1

The demo program was made for an 8031 system with 32K RAM, 32K ROM, an RS232 level converter, a 14.7456 MHz crystal and a single inverter for glue logic. You can make this board yourself.

The easiest way to connect the RAM is to invert A15 and use it as the chip select line so that RAM resides at 8000..FFFF. Connect WR to the 8031's WR line and OE to the 8031's PSEN line. The 8031's RD line isn't used. For an external ROM, use A15 as the ROM's chip select.

If you use the AND gate shown in Figure 1, you can place stacks in XRAM by setting InternalStack to 0 (see the file DEMO51.FF). An external stack is probably necessary if you use 32-bit cells. An internal stack seems to work well with a 16-bit Forth, but may not be safe for large applications. Of course, an 8032 would allow a much bigger internal stack.

The EEPROM's SDA line should be pulled up with a 4.7K resistor. See **`DEMO51.FF`** for pin assignments for the EEPROM. The EEPROM is optional. You don't need it to run the Firmware Studio demo. Of course, the effect is a lot better with a boot device.

`FLOAD DEMO51.FF` creates the object file **`ROM.HEX`**. If you change the source and want to recompile, type **`OK`**. Program the file **`ROM.HEX`** into the target board's program ROM. You can **`SHELL`** to DOS if necessary, to run your device programmer. Plug the target board into the host's serial port and apply power.

At this point, you can click on the "connect" button or any of the hex dump buttons and Firmware Studio will look on Com ports 1 through 4 for the target board and will auto-baud at 2, 4, 14, 19, 38, 57 or 115 kbps. Once communications has been established, the personality of the target board is read out so that Firmware Studio can send the appropriate commands to the target board.

The REG HEX window is one of the more interesting ones, since the numbers on the screen will be constantly changing. Addresses 000..0FF are mapped to IRAM, addresses 100..1FF are mapped to the SFRs. You can press F2 to enter edit mode, then change registers at will. Remember, the main program is running and can be crashed if you don't know what you're doing. But, it is always safe to browse.

Most windows pop up a short "Help" window if you press the F1 key. This help window disappears as soon as you "unfocus" (click outside of) it. On most windows, editable data is displayed on a white background. Pressing F2 in a Hex window toggles edit mode, in which you can enter hex or ASCII data. Pressing ESC in any popup window closes it.

After building a ROM image, token assignment and addresses of each word are known. If you start up Firmware Studio without building a ROM image, you **`FLOAD`** the header file **`ROM.HH`** to build up the token list. Type **`TE`** to enter the test mode, which lets you interactively test words on the target board. Click the "Virtual Console" button (the dumb terminal icon) to open the virtual console window.

Type **`-1 u.`** and **`65535`** will pop up in the console window. The virtual console is a section of memory on the target board used to simulate a console. Firmware Studio continually reads this memory and displays its contents in the window. This section of memory isn't very big. See **`AV0.FF`** for an example of how to allocate memory for a bigger console buffer.

BUG MYWORD invokes a high level debugger, which opens the source file containing **MYWORD** and lets you step through the code. You can switch back and forth between the debug window and the console by clicking the mouse, but be careful where you click in the debugger since click is used to reposition the "instruction pointer".

There are a lot of code words in the 8051's kernel, so that you can get decent speed using Forth. Subroutine threading is used for speed at the expense of code size. A traditional inner interpreter is painfully slow on an 8051 in contrast to subroutine threading which is quite fast. Subroutine calls are usually 3 bytes while calls in a traditional Forth list would have been 2 bytes each. Fortunately, the VM's architecture allows easy addition and removal of functionality. Seldom-used features needn't waste valuable code memory.

The low-level debugger for the 8051 allows you to single step instructions in order to debug code words. The registers shown on the left side of the window aren't the real registers. They are a register image that is swapped out with the real registers before and after an instruction is executed. Upon entering the debugger, you need to set up SP' and R0' to use an unused part of the stack space, so that your testing doesn't step on stack data. The main program is always running, so you must be careful about stepping through code that modifies global variables. Changing the wrong thing can crash the main program.

Accessing internal RAM and SFRs can't be done from high level Forth. It's best to write your own code words for this, using **KERNEL.F51** as an example. IRAM can be reached using @R1. SFR access is trickier since the SFR number is a hard-coded direct address. The debugger does generic access using self-modifying code.

Multitasking is tough on an 8051 because context switches are very clumsy. Therefore, the only multitasking support is the word **PAUSE**. **PAUSE** is invoked whenever a word is doing nothing, such as **EMIT** waiting for an output device to be ready. **PAUSE** invokes **DOORPAUSE**, which calls the backdoor debugger polling routine. Whatever you do with **PAUSE**, you should make sure it invokes **DOORPAUSE**. Otherwise, you'll lose the debugger. Polling routines and medium priority tasks are good candidates for **PAUSE** activity, since **PAUSE** is invoked so often.

Enter the Forthing mode by using the **FI** command. Now, when you type a command line, it will be processed by the tokenizer. The tokenized code will be sent to the target board and evaluated by the target board's evaluator. Click on the Virtual Console button to open the console window. Now, when you type **-1 U.**, the string **65535** will pop up in the console window. **CLS** clears the window.

This mode is much like traditional Forth. The main difference is that the text source is pre-digested into its tokenized equivalent by the host, sent the target and evaluated by the target.

The files **AV0.FF**, **AV1.FF**, etc. are test files, based on the John Hayes' ANS test suite. Fload **AV0.FF** and observe the console window for test results. You can Fload the other **AV*.FF** files in sequence to verify that the kernel words are working. These tests are important to have for porting to other processors.

You can concatenate all of the **AV*.FF** files using the DOS copy command and tokenize and evaluate the whole thing. If you select Host Evaluation speed to be line-at-a-time, the test file will be uploaded and evaluated a line at a time. This is slow, but it's useful if the target board hangs on a particular line.

You can extend the test suite to cover your own application words. Being able to periodically verify your work helps keep out bugs. The test suite also serves as a functional specification.

The 8051 demo board contains a serial 2-wire EEPROM for storing boot code. At startup, it attempts to evaluate boot code from each of eight possible devices on the IIC bus. See the file **`BOOT.FF`** to see some demo code. At startup, the ROM image built by **`DEMO51.FF`** evaluates the boot program stored in the EEPROM. A secondary image can be placed in the boot EEPROM by loading **`BOOT.FF`**.

`DEMO51.FF` builds a hyperlink index for the Winview editor. As you expand the system, you can end up with a huge number of keywords. In Winview, if you're not sure how a word is supposed to behave, place the cursor on it and hit F9. The source code for the word will pop up. In the Firmware Studio console, you can **`VIEW FOO`** to browse the source or **`SEE FOO`** to disassemble the ROM image of the word **`FOO`**.

Interrupt Service Routines

The 8051 services interrupts by jumping to locations in ROM such as 0003, 000B, etc. At each of these addresses is a LJMP instruction pointing into the binding table. The binding table directs execution to the desired ISR.

`0x0003 LJMP %INT0`	interrupt vector for EXT0
`0x000B LJMP %INT1`	interrupt vector for T0
`0x0013 LJMP %INT2`	interrupt vector for EXT1
`0x001B LJMP %INT3`	interrupt vector for T1
`0x0023 LJMP %INT4`	interrupt vector for UART
`0x002B LJMP %INT5`	interrupt vector for T2

If you look at MAIN in END.F51, you'll see how ISRs are linked to interrupt sources:

```
[CFA] b_timebase ['] %INT1 bind!
```

[CFA] b_timebase gets the address of the ISR code called b_timebase.
['] %INT1 bind! changes the destination of %INT1's LJMP in the binding table.

Performance Issues

The 8051 is a slow processor by today's standards, especially when doing 16-bit operations. The evaluator is written in Forth. It takes roughly 1000 machine cycles to evaluate each token. About half of this time is used to actually evaluate the token (compile or execute it) and the rest for housekeeping and **PAUSE**ing in the loop.

Based on experience with the test suite, large blocks of bytecode evaluate at a rate of about 2Kbytes per second on a 12 MHz 8031.

Most Forth primitives are written in assembly for speed. You can't wring much more speed from the system, although the evaluator has room for improvement.

Forth tends to be more compact than C because C spends a lot of code creating and destroying stack frames and shuffling parameters between registers and the stack. The amount of this code depends on the quality of the C compiler. Parameters are implicitly passed in Forth so all of this extra code goes away.

Types in Forth are all the same size, 16-bit or larger. On 8-bit processors, this puts Forth at a speed disadvantage with respect to typed languages like C.

Since Forth code tends to be finely factored, most of the application's time will be spent executing a relatively small number of subroutines. These can be re-coded in assembly. The end result is compact, testable code that's often faster than its C equivalent because of the lack of parameter-shuffling code.

You can extend the compiler to invoke executable code created by a C compiler. For example, you can write a filter program that extracts names and addresses from the map file left by the linker. I haven't done it yet, but it looks pretty easy. Unfortunately, each compiler vendor has different ways of passing parameters so it won't be portable.

The good news, though, is that it is very easy to call Forth words from C. Simply call into the binding table using the table’s origin (constant) minus 3*xt as the pointer value. So, TOF can be used to insulate a C application from hardware changes.

8031 Derivatives

The 8031 is the baseline processor of most 8051 families. Whatever the chip, if it can address external program memory it can probably run TOF with no modification. High speed CPUs are available from many vendors such as Dallas, Winbond, Philips and Cygnal.

Beware that some high speed chips (like Cygnal) don't support external program memory. So far, I haven't seen a chip with sufficient internal program RAM to run TOF. Chips with dedicated address and data ports are available, which will enable a 2-chip system.

Assembler labels for derivatives may be defined using ASMLABEL. For example, a DPTR selector at address 0x86 could be accessed with the following code:

```
0x86 asmlabel DPS

code foo  ( -- c ) call FALSE  mov dpl, dps  ret c;
```

Many of the newer chips have an extra DPTR to speed up block moves. You can re-code MOVE to use it, then either paste it into the kernel or redirect all MOVEs to use the new version:

‘ MyMove is Move	if building a ROM
[‘] MyMove [‘] Move rebind	if changing at run time in a subroutine definition
‘ MyMove ‘ Move rebind	if changing at run time from the console

If you're not cramped for space, you can include the new code in a new source file and use **MyMove is Move** so that the original code remains untouched.

DEMO51.FF uses Timer1 as a baud rate generator and Timer0 as a timebase. It uses the Timer1 interrupt flag for machine tracing. All other resources are free for your use. Kernel code starts at **0x0030**, which is set by an **ORG** directive near the beginning of **KERNEL.F51**. You might have to change this if you want to use interrupts supplied by new on-chip peripherals. Also, append **MAIN** found in **END.F51** to handle the new interrupt sources.

68000-based virtual machine

The 68000/ColdFire virtual machine uses subroutine threading with inlining. A peephole optimizer omits the useless code combinations that sometimes result when consecutive words are inlined. Cells are 32-bit. Code and data are aligned on 4-byte boundaries. From a programmer's perspective, the ColdFire is like a stripped down 68000 optimized for long data. It allows only long operations with most instructions, has some address mode restrictions and removes some obsolete instructions.

The Coldfire has better hardware multiply than 68K but not as good as CPU32. It has no hardware divide (shorter interrupt latency), so division will be slow.

The file **`DEMOCF.FF`** demonstrates virtual machines for the 68332 and the MCF5307.

Coldfire:

To demonstrate the Coldfire virtual machine, you'll need Motorola's MCF5307 evaluation board and at least one PC. The board's AUX port talks to Firmware Studio, the TERMINAL port talks to a terminal. Two PCs on a network is good for development, since you can make code on one PC and test it on the other.

`FLOAD` the file **`DEMOCF.FF`** to create a s-record file **`TEST.S`** suitable for running on Motorola's MCF5307 evaluation board. Connect a PC running a terminal program (19200,n,8,1) to the terminal port. From the terminal program, enter **`DL`** to start downloading code to the evaluation board. Send the **`TEST.S`** file when prompted by the EVS debugger. When the download is complete, type **`GO 20000`** to start the virtual machine. If you're using a version of the EVS debugger older than V1.3.4 (very unlikely), use the DOS command **`COPY TEST.S+TEST.S TEST.SS`** and remove the first S8 record in **`TEST.SS`** to make a file that will download correctly. I tested with an old Arnewsh evaluation board.

Connect the PC with Firmware Studio on it to the evaluation board's auxiliary serial port.

68332:

Change the board type (at the top of the file **`DEMOCF.FF`**) to 2. **`FLOAD`** the file as described above to create a ROM image. Connect a PC running a terminal program (9600,n,8,1) to the terminal port. From the terminal program, enter **`LO`** to start the downloading mode. Send the **`TEST.S`** file. When the upload is complete, hit carriage return a couple of times until you see the prompt. Type **`GO`** to start the virtual machine.

At this point, you can click on the "connect" button or any of the hex dump buttons and Firmware Studio will look on Com ports 1 through 4 for the target board and will auto-baud at 2, 4, 14, 19, 38, 57 or 115 kbps. The demo uses 115K, of course. Once communications has been established, the personality of the target board is read out so that Firmware Studio can send the appropriate commands to the target board.

After building a ROM image, token assignment and addresses of each word are known. If you start up Firmware Studio without building a ROM image, you **`FLOAD`** the header file **`ROM.HH`** to build up the token list. Type **`TE`** to enter the test mode, which lets you interactively test words on the target board. Click the "Virtual Console" button (the dumb terminal icon) to open the virtual console window.

Type **`-1 u.`** and **`4294967295`** will pop up in the console window. The virtual console is a section of memory on the target board used to simulate a console. Firmware Studio continually reads this memory and displays its contents in the window.

`BUG MYWORD` invokes a high level debugger, which opens the source file containing **`MYWORD`** and lets you step through the code. You can switch back and forth between the debug window and the console by clicking the mouse, but be careful where you click in the debugger since click is used to reposition the "instruction pointer".

The low-level debugger for the 68K/ColdFire allows you to single step instructions in order to debug code words. The registers shown on the left side of the window aren't the real registers. They are a register image that is swapped out with the real registers before and after an instruction is executed. The main program is always running, so you must be careful about stepping through code that modifies global variables. Changing the wrong thing can crash the main program. Try **`TRACE FOO`** to try out the machine debugger. Click on the **`NOP`** just before the **`RTS`** instruction, then press 'N' and watch the show.

Enter the Forthing mode by using the **`FI`** command. Now, when you type a command line, it will be processed by the tokenizer. The tokenized code will be sent to the target board and evaluated by the target board's evaluator. This mode is much like traditional Forth. The main difference is that the text source is pre-digested into its tokenized equivalent by the host, sent the target and evaluated by the target.

The files **`AV0.FF`**, **`AV1.FF`**, etc. are test files, based on the John Hayes' ANS test suite. Fload **`AV0.FF`** and observe the console window for test results. You can Fload the other **`AV*.FF`** files in sequence to verify that the kernel words are working. These tests are important to have for porting to other processors.

You can concatenate all of the **`AV*.FF`** files using the DOS copy command and tokenize and evaluate the whole thing. If you select Host Evaluation speed to be line-at-a-time, the test file will be uploaded and evaluated a line at a time. This is slow, but it’s useful if the target board hangs and you want to find the line the line that caused the problem.

You can extend the test suite to cover your own application words. Being able to periodically verify your work helps keep out bugs. Plus, the test suite serves as a functional specification.

As of version 2.08 (8/00), the ColdFire VM runs on the MCF5307 evaluation board and passes the test suite. The timebase interrupt isn't working yet, but that's more of a system function and wasn't necessary for testing. The 68332 BCC version works, including the timebase.

`DEMOCF.FF` builds a hyperlink index for the Winview editor. As you expand the system, you can end up with a huge number of keywords. In Winview, if you're not sure how a word is supposed to behave, place the cursor on it and hit F9. The source code for the word will pop up. In the Firmware Studio console, you can **`VIEW FOO`** to browse the source or **`SEE FOO`** to disassemble the ROM image of the word **`FOO`**.

Performance Issues

Doing some rough comparisons, the MCF5307 at 90 MHz was five times as fast as the 68332 at 16 MHz. This sluggishness was caused by the MCF5307's cache being disabled. Code was executing from DRAM. The MCF5307 has a unified cache that can be made to work with self-modifying code, so it should be able to run a lot faster.

Cells are 32-bit, so on a 16-bit bus there is a speed penalty. Since most of your time will be spent in a small section of code, re-coding the appropriate subroutines in assembly will address the speed issue. You still get the benefit of implicit parameter passing, resulting in code that's both compact and fast.

The code optimizer is better than nothing, but much worse than the amazing optimizers that come with today's commercial Forths. So, I tried to compensate by providing a decent assembler. It uses Motorola's syntax, so an algorithm coded in C can be compiled to assembly and then (in theory) cut-and-pasted into your application.

AVR development - Getting Started

The AVR family of microcontrollers is Forth-friendly, having been designed with another popular stack-based language in mind: 'C'. The typical AVR doesn't have SRAM-based program memory, so it won't support TOF. However, Firmware Studio serves as a good development tool for AVR code whether it be Forth or assembly.

You can compile code with Firmware Studio and download it using Atmel's AVRisp utility. Then, you may interactively test code from Firmware Studio's console. If you need to, AVRstudio (free from Atmel) lets you simulate execution of a hex file. You currently can’t download code from Firmware Studio, but switching between windows isn’t a hassle.

To get started right away, you'll need Atmel's STK200 or STK300 starter kit. The STK200 comes with an AT90S8515. It's available for about $50 from any Atmel distributor. Digi-Key has them for $49, part# ATSTK200-ND. Atmel sales reps have been known to give these away, since they're such great promotional items and the cost of making the sales call swamps the cost of the STK anyway.

You'll also need a straight-through 9-pin serial cable. This is sometimes called a Monitor Extension Cable, but make sure it's not the 15-pin VGA type. If you build your own cable using male and female DE-9 connectors, you only need to wire pins 2, 3 and 5.

Plug a wall-wart power supply into the STK board, connect the serial port to your PC using the serial cable, and plug the STK's dongle into the PC's parallel port and the STK's programming header. Install Atmel's tools, especially AVRISP. If you're running Windows NT, you may need to get a special version of AVRISP from Atmel.

For the STK200, start Firmware Studio and enter **`FLOAD DEMOAVR.FF`**. This will compile the demo program and save a hex file DA.HEX. Then, launch Atmel's AVRISP utility. Pick Project New from the menu, then select the AT90S8515 from the device list and fill in the name and comment information. There will be several windows on the screen. Click on the "Program Memory" window to make the rest of the menus active. Pick File Load to load the file DA.HEX. Do a Project Save. Now, hit F5 to download the code to the board. Switch back to Firmware Studio but don't close AVRISP.

Whenever you recompile, you can switch to AVRISP and hit F5 to reload the hex file and program the AVR part.

Click on the Target button. Firmware Studio will look for the board on all serial ports, then connect to it. Once this happens, you can type numbers and words at the console and have them execute on the target board. Try the following:

```
1 2 3 +
```

On the left side of the screen, you'll see the stack data resident on the target board.

The word '+' takes two 16-bit numbers from the data stack, adds them, and pushes the result back onto the stack. The top of the data stack is the 16-bit X register. To see the disassembled code, type **SEE +**.

```
00078 9009     +:       LD    R0,Y
0007A 9009              LD    R0,Y+
0007C 0DA0              ADD   R26,R0
0007E 1DB1              ADC   R27,R1
00080 9508              RET
```

BUG MYWORD invokes a high level debugger, which opens the source file containing **MYWORD** and lets you step through the code. You can switch back and forth between the debug window and the console by clicking the mouse, but be careful where you click in the debugger since click is used to reposition the "instruction pointer".

AVR Register Usage

Forth uses the registers listed in the following table. The other registers are available for your application. You are free to use the scratchpad registers, but be aware that they may be changed by calls to Forth subroutines.

Reg	Aliases	Usage	Reg	Aliases	Usage
R0	WL, W	scratchpad	R16		scratchpad
R1	WH	scratchpad	R17		scratchpad
R2	UL, U	User pointer	R18		scratchpad
R3	UH	for multitasking	R19		scratchpad
R4			R20		
R5			R21		
R6			R22		
R7			R23		
R8			R24	AL, A	'A' pointer
R9			R25	AH	
R10			R26	XL,X,TOSL	Top of data stack
R11			R27	XH,TOSH	
R12			R28	YL, Y	Data stack pointer
R13			R29	YH	
R14			R30	ZL, Z	scratchpad
R15			R31	ZH	scratchpad

Using the assembler

The assembler uses Atmel's instruction syntax. An instruction consists of an instruction name and an operand list. The operand list mustn't contain spaces. You can put multiple instructions on the same line. Subroutines start with **CODE** and end with **C;** or **END-CODE**. Browse KERNEL.FAR for lots of examples.

Inline assembly can be placed in **CODE** definitions by enclosing it within **C[** and **]C**.

R20 to R23 are free for your use, and they make good state machine pointers. Consider the following example of a simple state machine. The **VECTOR** macro compiles two **LDI** instructions to load register pair R21:R20. Each call to DOSTATE changes state. The code is efficient enough for use in interrupt-driven state machines.

```
code state3     ldi R16,1  out PortB,R16
                vector{             reti c;
code state2     ldi R16,2  out PortB,R16
                vector  R20,state3   reti c;
code state1     ldi R16,4  out PortB,R16
                vector  R20,state2   reti c;
                }vector R20,state1
code dostate    mov ZL,R20   mov ZH,R21  ijmp c;
```

The code in the demo file MAVR.FF contains a state machine that handles half-duplex RS485 communication. A packet-based link protocol complete with error checking and CRC generation is handled in the background by an ISR.

You can run it on the STK200 board (with AT90S8515) by FLOADing MAVR.FF and programming the board with the hex file. Use a straight-through serial cable to connect the board to the host PC’s COM1. Firmware Studio supports the new mode of communication with the Target Multidrop menu command. With minor modification, you can hook up RS485 converters and operate multiple target boards over a twisted-pair. For more details, see Appendix E.

The CYCLE_DELAY macro lays down code to produce exact time delays of 0 to 770 cycles. For example, **49 CYCLE_DELAY** compiles code to waste 49 clock cycles. R16 is cleared.

The LDI_R16[macro compiles an LDI R16,N instruction using a list of bit positions. It's useful for defining initialization code for ports. A list of bit labels is delimited by a bracket. For example, if RXEN is 4 and TXEN is 3, **LDI_R16[RXEN TXEN]** sets bits 4 and 3 to form **LDI R16,0x18**.

The LDIW macro compiles two LDI instructions to load a register pair with a 16-bit constant.

The LDIP is like LDIW but divides the constant by 2 to serve as a program address.

JSR compiles an RCALL if possible, otherwise it compiles CALL. Note that some AVRs don't support CALL. The ICALL instruction may be needed in that case, so you might have to fix JSR to compile the necessary code.

GOTO is the RJMP / JMP equivalent of JSR.

An immediate operand can an assembly-label, number, local label, code-address, or ASCII character in that order. For example, if you define a Forth word called '0', rcall 0 calls address 0x0000 because it found '0' as a number before it found it in the code address list. When in doubt, try it and SEE the result. Local labels are @@0 thru @@9 (see below). ASCII characters are characters between two tick marks. For example, **`LDI R16,'A'`**.

Branches are compiled using control structures. You can extend the compiler BLDAVR.G to support local labels, but I've found control structures sufficient for defining any kind of branching I need to do. They are also more readable.

`FOR … NEXT Rn` compiles code to do something 1 to 256 times. The NEXT lays down code to decrement Rn and branch back if not zero.

`IF_Z <code…> THEN` compiles a **`BRNE`** past the <code> instructions. So, the code only executes if the Z flag is set. IF_NZ, IF_C, IF_NC, etc. are similar.

`IF_Z <code1…> ELSE <code2…> THEN` is similar. If the Z flag is set, <code1> executes. Otherwise, <code2> executes.

`NEVER` is a version of IF that branches around code. **`NEVER THEN`** and **`NEVER ELSE THEN`** are useful when they are preceded by a conditional skip instruction.

`BEGIN <code…> AGAIN` compiles a forever loop. A skip is useful before the **`AGAIN`**.

`BEGIN <code…> UNTIL_Z` compiles a loop ending in a **`BRNE`** instruction. UNTIL_NZ, UNTIL_C, UNTIL_NC, etc. are similar.

`BEGIN <code1…> WHILE_Z <code2…> REPEAT` is good for doing something zero or more times. **`WHILE_Z`** compiles a **`BRNE`** past <code2> and **`REPEAT`** compiles a branch to the beginning of <code1>.

`NOWAY` is a version of **`WHILE`** that compiles an unconditional jump. Put a skip in front of it.

`CONTINUE` can be placed between **`WHILE`** and **`REPEAT`** to branch back to **`BEGIN`**.

`MULTI Rn <code…> REPEAT` is good for doing something zero or more times. Similar to a **`FOR`** loop but used to do something 0 to 128 times. MULTI lays down DEC Rn and BRPL.

`REGISTER:` defines a new register name. For example, **`23 register: flags`** creates an alias for R23 called flags.

`JUMP[ R16 label1 label2 … labelN ]JUMP` compiles code for a jump table that uses R16 as the index. Any register may be used. This must be a one-liner.

CASE R16 # OF … ENDOF
ENDCASE compiles a CASE structure consisting of CPI R16,# instructions and branches. Registers are restricted to R16 to R31.

Example of case usage:

```
case R16 10 of rcall ten    endof
         11 of rcall eleven endof
         12 of rjmp twelve |endof
         rcall otherwise
endcase
```

Local assembly labels @@0 thru @@9 are available for compile-time calculations by the Forth interpreter. For example, **`{{ asmlabel? PortA 1+ >@@0 }}`** sets the value of local label @@0. The following example uses this label: **`ldi R16,@@0`**

Words between brackets {{, }} are assumed to be in the home vocabulary and not target words. Since this is a cross compiler, target words don't execute. There are some exceptions, such as words that behave like literals are simulated.

The LPM and SPM instructions need a parameter list, even though they may be implicit. Just use a | character in this case. Example: **`LPM |`** .

>CS and CS> push and pop branch parameters. IF_Z … >CS … IF_C … CS> THEN … THEN compiles the following:

```
     BRNE L0
     …
     BRCC L1
     …
L0   …
L1   …
```

Browse BLDAVR.G to see the assembler and builder source. You can add you own directives and see how the existing ones work.

Using Forth

The builder compiles subroutine threaded code. Words that are very short are simply inlined. Some words preceded by literals are optimized. +, -, AND, OR, @, C@, ! and C! are optimized when used in conjunction with literals. Note that variables and constants are considered literals. For example, the definition **`: HEX 16 base ! ;`** compiles to something like:

```
00578 E000     HEX:     LDI   R16,0
0057A 93000101          STS   101,R16
0057E E00A              LDI   R16,10
00580 93000100          STS   100,R16
00584 9508              RET
```

There's a flag in BLDAVR.G that enables speed optimization. Literals are inlined for speed. They require 4 instructions and execute in 6 cycles. With optimization turned off, literals are encoded using 2 or 3 instructions depending on the value.

Cooperative multitasking relies on the use of PAUSE. PAUSE takes a lap around the task queue, so you should PAUSE whenever you're waiting for I/O. Context switching is fairly quick. With an 8 MHz xtal, PAUSE takes about 4us to do a context switch and 1.5us to skip over each sleeping task -- not bad for an 8-bit micro. The AVR demo demonstrates multitasking.

Every task must have a PAUSE or a word that calls PAUSE in it, or it will hang the system. When tasks are I/O bound, cooperative multitasking provides a very efficient way to use CPU time. A sleeping task takes very little CPU time, so you should SLEEP a task when it's doing nothing. Ideally you'd use an ISR to wake a task when it needs to do something. Use an ISR to do the time-critical part of a task, then wake up a task to finish the job and do clean up.

IFCLR and IFSET are special versions of IF. They compile efficient bit tests using the SBRS and SBRC instructions.

Sample Usage: **`[ 3 ] IFCLR SWAP THEN`** is the same as **`DUP 8 AND 0= IF SWAP THEN`**.

A compact CASE-like structure is similar. The restrictions are: Only the low byte of **c** is used and each case must be short enough to be covered by a short branch. Also, there is nothing following the QCASE structure. **]?** compiles an exit.

For example:

```
: test          ( c -- )
         qcase: [ 1 ] ?[ sqrt ]?
                [ 3 ] ?[ dist ]?
                [ 5 ] ?[ dup over rot ]?
                ;
```

This is equivalent to the ANS Forth CASE structure:

```
: test          ( c -- )
           case 1 of sqrt endof
                3 of dist endof
                5 of dup over rot endof
                ;
```

DEMOAVR.FF builds a hyperlink index for the Winview editor. As you expand the system, you can end up with a huge number of keywords. In Winview, if you're not sure how a word is supposed to behave, place the cursor on it and hit F9. The source code for the word will pop up. In the Firmware Studio console, you can **VIEW FOO** to browse the source or **SEE FOO** to disassemble the ROM image of the word **FOO**.

Your own hardware

You can use the file DEMOAVR.FF as a starting point for your code. The demo code includes the debugger and Forth words. It uses Timer0 for a timebase interrupt, which can be stripped out if you need that timer. The token browser has a usage field that tells how many times a word is referenced. If you need to strip out code to make your application fit in flash, use this tool to see where the dead code is. You don't want to omit all dead code, since some of it is useful for testing. For small devices, you might want to develop code using a part with more ROM space and then remove the debugger to fit it in the final design. Notice that you can easily use conditional compilation to include debugging tools or not.

Code in **`BEGIN.FAR`** selects the timer reload value needed to use the highest possible autobaud frequency. It picks the lowest baud rate divisor that will give a baud rate error of less than 2%. The target board's UART must run at 2.4, 4.8, 14.4, 19.2, 38.4, 57.6 or 115.2 kBPS. An arbitrary crystal frequency may be used but you'll probably end up with a low baud rate.

Connect the UART pins to the host's serial port (any port from Com1 to Com4) through the RS232 transceiver chip. Note that most PCs will accept a 0 to 5V swing as valid RS232 input, so for demonstration you could use a 74HC04 as the transceiver chip (100k in series with PC's TXD line).

Firmware Studio's "reset" command resets the target board by dropping the DTR line (pin 4 of the PC's DE9 connector) for ½ second. This is a very useful feature to have in a debugging environment. Provide for this in your hardware if possible.

BUILDER glossary

When in BUILDING mode, the BUILDER lexicon is in effect. While interpreting, these words take precedence over anything else in the search order. While compiling, only those words having a compile action take precedence. For example, : FOO ALIGN ; expects ALIGN to be a target word.

Stack Picture:	*Compile*	*Host interpret*	*Target Interpret*
'	---	(<name> -- xt)	---
	Get the token# of a word. "Tick".		
+LOOP	(addr f --)	---	(n -- | R: loop --)
	Compile code to end a DO..LOOP		
,	---	(x --)	---
	Append a cell to the dictionary. Use RAM and ROM directives to select the address space: RAM mode lays down a cell to the data image that will be used by the startup code to initialize RAM data space. ROM mode appends a cell to the ROM image.		
,"	---	(<string"> --)	---
	Appends a counted string to the dictionary. The incoming string is delimited by a quote.		
."	(<string"> --)	(<string"> --)	(--)
	Compiles a string to the ROM image along with code to invoke TYPE. If not compiling, types string on the console like Forth79.		
2CONSTANT	---	(d <name> --)	(-- d)
	Compile code to return a double literal.		
2VARIABLE	---	(<name> --)	(-- addr)
	Compile code to return the address of a double variable.		
:	---	(<name> -- f)	(? -- ?)
	Start a definition. Compiles a header byte to the ROM image. The new definition is placed in the current (green) vocabulary.		
;	(f --)	---	---
	End a definition. Compiles a return instruction or, if possible, converts the previous call to a jump instruction.		

?DO	(-- addr f)	---	(hi lo -- | R: -- loop)	Compile code to start a ?DO..LOOP structure.
AFTER:	---	(<name> -- f)	(? -- ?)	Extends a definition that already exists. Compiles a header byte to the ROM image. **`;`** will compile a jump to the old definition. Usage: **`after: INIT  your-code ;`** This can be used to extend startup or other code. Ideally, a hardware device should be supported by a single file. In OO lingo, you could say the methods for this object are contained in one file. The file to support a UART chip, for example, would extend startup code to initialize certain registers. See BEFORE:.
AGAIN	(addr f --)	---	(--)	Compile a backward unconditional branch for BEGIN..AGAIN structure.
AHEAD	(-- addr f)	---	(--)	Compile a forward unconditional branch. Similar to IF.
ALIGN	---	(--)	---	Adjust HERE upward to bring it to the next aligned address.
ALIGNED	---	(addr -- addr')	---	Adjust addr upward to bring it to the next aligned address.
ALLOT	---	(n --)	---	Allocate n bytes in the dictionary. If RAM mode, these bytes are set to zero.
ALSO	---	---	---	Duplicate the top of the search order.
ARRAY	---	(n <name> --)	(-- addr)	Compile code to return the address of an array of size n. Allots n bytes of storage.
ASSEMBLE	---	(-- f)	---	Starts in-line compilation of machine code. Usage: In-line code: : FOO swap [assemble your_code_here c;] over ; Macro: assemble your_code_here c; macro: YourMacro code FOO YourMacro next c;

BEFORE: --- (<name> -- f) (? -- ?)
Extends a definition that already exists. Compiles a header byte and a call to the old definition to the ROM image.
Usage: **`before: INIT  your-code ;`**
See AFTER:.

BEGIN (-- addr f) --- ---
Marks the beginning of a BEGIN UNTIL or BEGIN WHILE REPEAT structure.

BINDINGS, --- --- ---
Appends a data structure to ROM. The first cell of the structure contains the number of entries. The second cell contains the address of the executable code associated with token 0. Subsequent cells are addresses of token1 to token n-1.

C" --- (<string"> --) (-- addr)
Appends a counted string to the dictionary. The incoming string is delimited by a quote. Runtime code skips over the data and returns its address.

C(--- (<string> --) ---
Changes the comment field of the last defined token to the string delimited by close-parenthesis. The comment field is typically 32 characters maximum.

C, --- (c --) ---
Appends a byte to the dictionary.

CALL (<name> --) --- (? -- ?)
Compiles a call to an external procedure.

CALL-ONLY --- --- ---
Sets the call-only bit in the header of the last defined word. This bit prevents **;** from converting a call to this word to a jump.

CASE (-- addr f) --- ---
Marks the beginning of a CASE .. OF .. ENDCASE structure.

CFA --- (<name> -- addr) ---
Gets the actual address of a word.

CODE	---	(<name> -- f)	(? -- ?)	Starts a code definition. Compiles a header byte to the ROM image. The new definition is placed in the current (green) vocabulary.
CODE-BOUNDS?	---	(-- lo hi)	---	Returns the lower and upper limit of the user's code space.
CONSTANT	---	(x <name> --)	(-- x)	Compiles code to return a literal.
CREATE	---	(<name> --)	(-- addr)	Creates a pointer for data structure in data space. Compiles code to return the address of the next data. On processors other than AVR, compilation is placed in RAM mode. That means that c, and , will lay down data structures to RAM. Startup code will initialize this RAM. You can conserve code space by using PCREATE.
DATA-BOUNDS?	---	(-- lo hi)	---	Returns the lower and upper limit of the user's data space.
DECIMAL	---	---	---	Uses 10 as the host's radix for numeric conversion.
DEFER	---	(<name> --)	---	Creates a token but doesn't lay down any data for it. IS will be used later to bind code to this token.
DEFINITIONS	---	---	---	Makes the top vocabularty of the search order the current vocabulary.
DO	(-- addr f)	---	(hi lo -- | R: -- loop)	Compile code to start a DO..LOOP structure.
DYNAMIC	---	---	---	Switches to dynamic mode, which causes all calls to go through the binding table.
ELSE	(a1 f1 -- a2 f2)	---	---	Compiles forward branch, resolve IF's forward branch as part of an IF .. ELSE .. THEN structure.

END*CODE	---	---	---	Does nothing. This word is used to embed assembly code in tokenized bytecode.
ENDCASE	(addr f --)	---	---	Finishes the CASE .. OF .. ENDOF .. ENDCASE structure.
ENDIF	(addr f --)	---	---	Same as THEN. Resolves forward branch left by IF or ELSE.
ENDOF	(a1 f1 -- a2 f2)	---	---	Resolves OF branch, lays down forward branch to ENDCASE.
EXIT	(--)	---	---	Exits a definition. Compiles a return instruction or changes the last call instruction to a jump.
FI	---	---	---	Enters the interactive Forth mode.
FORTHING	---	---	---	Enters the interactive Forth mode.
HEADERS-OFF	---	---	---	Turns off header generation. Saves a byte or two per word.
HEADERS-ON	---	---	---	Turns on header generation. Any word without a header can't be used by the evaluator. Useful for words defined after the binding table has been resolved.
HERE	---	(-- addr)	---	Returns the address of the next free byte in code or data space, depending on whether you're in RAM or ROM mode.
HEX	---	---	---	Uses 16 as the host's radix for numeric conversion.
HOST	---	---	---	Enters the host's Forth mode.
IF	(-- addr f)	---	(f --)	Compiles a conditional forward branch.

IMMEDIATE	---	---	---
	Sets the immediate bit in the header of the last defined word. This bit forces this word to execute instead of being compiled.		
IS	---	(xt <name> --)	---
	Assigns xt's action to <name>.		
ISA	---	(addr <name> --)	---
	Assigns the action at addr to <name>.		
LITERAL	(x --)	---	(-- x)
	Compiles code to push a literal onto the stack.		
LOCO	---	(<name> --)	---
	Begin a local CODE definition. Works like CODE but doesn't create a header or assign a token. Instead, it creates an assembler label.		
LOOP	(addr f --)	---	(n -- | R: loop --)
	Compile code to end a DO..LOOP		
MACRO	---	---	---
	Sets the macro bit in the header of the last defined word. This bit enables this word to copied into the dictionary as inline code instead of being invoked with call instruction.		
MULTI	(-- a1 f1 a2 f2)	---	(| R: x -- x-1)
	The MULTI ... REPEAT structure is equivalent to BEGIN R@ 0< 0= WHILE R> 1- >R ... REPEAT. It's much more efficient than ?DO ... LOOP.		
NOBIND	---	---	---
	Sets the nobind bit in the header of the last defined word. This bit forces calls to this word to not use the binding table.		
OF	(-- addr f)	---	(n1 n2 -- n1 | m m --)
	Compiles code for OF, which skips the OF .. ENDOF phrase if n1<>n2.		
ONLY	---	---	---
	Clears the search order so it only contains the HOME wordlist.		

OPTIM	(n --)	(n --)	---	Stores n to the byte immediately after the header byte. On some processors, a second byte is necessary for alignment reasons. This byte can be used for compiler optimization information.
ORG	---	(addr --)	---	Sets the pointer to the next free byte in the dictionary.
PCREATE	---	(<name> --)	(-- addr)	Creates a pointer for data structure in program space. Compiles code to return the address of the next available code byte. If @ and ! can access program space, you can use PCREATE instead of CREATE to compile data structures to ROM, thereby saving some RAM space.
PREVIOUS	---	---	---	Drops the top vocabulary in the saerch order.
RAM	---	---	---	Set RAM mode, which causes comma and related words to append data to data space. Startup code will inilialize RAM from this data. Some builder words leave the compiler in ROM mode, so when in doubt use RAM.
RECURSE	---	---	---	Compiles a static call to the current definition.
REPEAT	(a f --)	---	---	Compile backward branch to resolve a BEGIN .. WHILE .. REPEAT structure.
RETRY	---	---	---	Compiles a jump to the current definition.
ROM	---	---	---	Set ROM mode, which causes comma and related words to append data to the ROM image space.
ROM-BOUNDS?	---	(-- lo hi)	---	Returns the lower and upper limit of the ROM image space.
S"	---	(<string"> --)	(-- addr len)	Appends a string to the dictionary. The incoming string is delimited by a quote. Runtime code skips over the data and returns its address and length.

STATIC	---	---	---	Switches to static mode, which causes all calls to be direct.
STRING	---	(len <name> --)	(-- addr len)	Compile code to return the address and length of an array of size len. Allots len bytes of storage.
TE	---	---	---	Enters testing mode. For interactive testing of target words.
TESTING	---	---	---	Enters testing mode. For interactive testing of target words.
TEST{	---	---	---	Marks beginning of a block of test code. Forces STATIC mode
THEN	(addr f --)	---	---	Resolves forward branch left by IF or ELSE.
TI	---	---	---	Enters tokenizing mode.
TO	(<name> --)	---	(--)	Compiles code to store to a value. TO XYZ compiles to Call XYZ Call (%TO!).
TOKENIZING	---	---	---	Enters tokenizing mode.
UNTIL	(addr f --)	---	(f --)	Compiles a conditional backward branch to resolve a BEGIN .. UNTIL structure.
VALUE	---	(x <name> --)	(-- x)	Compiles code to return a value. Unlike a CONSTANT, you can use TO to change its value.
VALUES,	---	---	---	Appends a data structure to ROM. The first cell of the structure contains the start address of initialized RAM. The second and third cells contain a repeat count and a value. Subsequent cell pairs contain further run-length encoded data. The structure is terminated by a repeat count of zero.

VARIABLE	---	(<name> --)	(-- addr)
	Compiles code to return the address of the next data and allocates aligned data space storage for a cell.		
VOCABULARY	---	(<name> --)	---
	Creates a word that replaces the top of the search order with its own vocabulary.		
W,	---	(n --)	---
	Appends a 16-bit value to the dictionary.		
WHILE	(a1 f1 -- a2 f2 a1 f1)	---	(f --)
	Compiles a conditional forward branch as part of a BEGIN .. WHILE .. REPEAT structure.		
WORDS	---	---	---
	Displays all words in the vocabulary at the top of the search order.		
[	---	---	---
	Switches to interpret mode. A typical usage is : Foo ... [YourCalculation] Literal ... ;		
[']	---	(<name> --)	(-- xt)
	Compile code that returns the token# of a word.		
[CFA]	---	(<name> --)	(-- addr)
	Compile code that returns the execution address of a word.		
[CHAR]	---	(<char> --)	(-- c)
	Compile code that returns a character.		
[COMPILE]	---	(<name> --)	(-- addr)
	Compile code that returns the execution address of a word.		
[DYNAMIC]	---	---	---
	Switches to dynamic mode, which causes all calls to go through the binding table. It has the same effect regardless of STATE.		
[STATIC]	---	---	---
	Switches to static mode, which causes all calls to be direct. It has the same effect regardless of STATE.		
]	---	---	---
	Enters compilation mode.		

}TEST	---	---	---

Downloads image code between TEST{ and }TEST to the same address on the target board. Then enters TESTING mode.

TOKENIZER glossary

When in TOKENIZING mode, the TOKENIZER lexicon is in effect. These words take precedence over anything else in the search order.

Interpreter algorithm:

```
    Search the search order for the next blank delimited string
    Found?
    IF      Execute it
            Was this a target word?
            IF      Compile a token for it
            ENDIF
    ELSE
            Try to convert the word to a number
            Converted ok?
            IF      Compile the token for literal
            ELSE    Error: word is unrecognized
            ENDIF
    ENDIF
```

Tokenizer Lexicon:

#ELSE	(--) Lays a "0 #IF", resolves the previous #IF in the structure #IF ... #ELSE ... #THEN
#ENDIF	(--) Resolves the displacement left by #IF.
#IF	(--) Lays token for #IF# and 2-byte forward displacement.
#THEN	(--) Resolves the displacement left by #IF.
%%	(<name> --) Declares a token, lays this token#.
,"	(<string"> --) Lays the token for ," and a counted string.
.	(--) Lays the tokens for (.) and TYPE..
."	(<string"> --) Lays the token for ." and a counted string.

.R	(--) Lays the tokens for (.R) and TYPE..
2CONSTANT	(<name> --) Compiles the token for 2CONSTANT. If outside a definition, creates a new token for <name> and compiles it.
2VARIABLE	(<name> --) Compiles the token for 2VARIABLE. If outside a definition, creates a new token for <name> and compiles it.
:	(<name> --) Starts a new definition. Compiles the token for **:**. If <name> is already defined, an error message occurs.
::	(<name> --) Redefines an existing definition. Compiles the token for **:** and the token for <name>.
;	(--) Ends a definition. Compiles token for **;**.
ALSO	(--) Duplicates the top of the search order.
ARRAY	(<name> --) Compiles the token for ARRAY. If outside a definition, creates a new token for <name> and compiles it.
BI	--- --- --- Enters the ROM builder mode.
BUILDING	--- --- --- Enters the ROM builder mode.
C"	(<string"> --) Lays the token for C" and a counted string.
CHAR	(<char> --) Compile token for CHAR and a byte for <char>.
CODE	(-- magic# f) Begins a code definition using the current assembler. Usage: code FOO YourCode c; end*code.

CODE{	(<name> --) Begins a code definition without an assembler. Sample usage: **`CODE{ FOO 1 HEX[ F3 DE 22 ] }CODE`** where the number immediately after FOO is the target CPU type. The target's evaluator will only compile this code if the CPU type matches. Example: Given the following code, the tokenizer will define **`3+`** in machine code if the target is an 8051. Otherwise, it will use the high level definition. **`: 3+  3 + ;`** **`code{ 3+ 1 HEX[ A3 A3 A3 ] }CODE`**
CONSTANT	(<name> --) Compiles the token for CONSTANT. If outside a definition, creates a new token for <name> and compiles it.
CREATE	(<name> --) Compiles the token for CREATE. If outside a definition, creates a new token for <name> and compiles it.
D.	(--) Lays the tokens for (D.) and TYPE..
DEFINITIONS	--- --- --- Makes the top vocabulary of the search order the current vocabulary.
END	(--) Resolves the length and checksum of a block of bytecode inside a PROGRAM ... END structure.
END*CODE	(magic# --) Ends a code definition, resolves the length of the code string.
FI	(--) Enters the interactive Forth mode.
FORTHING	(--) Enters the interactive Forth mode.
HOST	(--) Enters the host's Forth mode.

MARKER	(<name> --) Compiles the token for MARKER. If outside a definition, creates a new token for <name> and compiles it.
ONLY	(--) Clears the search order so it only contains the HOME wordlist.
POSTPONE	(<name> --) Compile the token for POSTPONE and the token for <name>.
PREVIOUS	(--) Drops the top vocabulary in the search order.
PROGRAM	(<name> --) Creates a token for <name>. Marks the beginning of a tokenized program with a 7-byte header. See END.
S"	(<string"> --) Lays the token for S" and a counted string.
STRING	(<name> --) Compiles the token for STRING. If outside a definition, creates a new token for <name> and compiles it.
TE	(--) Enters testing mode. For interactive testing of target words.
TESTING	(--) Enters testing mode. For interactive testing of target words.
TI	(--) Enters tokenizing mode.
TOKENIZING	(--) Enters tokenizing mode.
U.	(--) Lays the tokens for (U.) and TYPE..
U.R.	(--) Lays the tokens for (U.R) and TYPE..
UD.	(--) Lays the tokens for (UD.) and TYPE..

VALUE	(<name> --) Compiles the token for VALUE. If outside a definition, creates a new token for <name> and compiles it.
VARIABLE	(<name> --) Compiles the token for VARIABLE. If outside a definition, creates a new token for <name> and compiles it.
VOCABULARY	(<name> --) Creates a word that replaces the top of the search order with its own vocabulary.
WORDS	(--) Displays all words in the vocabulary at the top of the search order.
[']	(<name> --) Compile a literal for the xt of <name>.
[CHAR]	(<char> --) Compile a literal for <char>.
[COMPILE]	(<name> --) Compile the token for [COMPILE] and the token for <name>.
}CODE	(magic# --) Terminates an embedded machine code definition. See CODE{.

TESTER glossary

When in TESTING mode, the TESTER lexicon is in effect. These words take precedence over anything else in the search order. The target device's stack is used for testing.

Stack Picture:	*Target Interpret*
'	(<name> -- xt) Gets the token# of a word. "Tick".
ALSO	(--) Duplicates the top of the search order.
BI	(--) Enters the ROM builder mode.
BUILDING	(--) Enters the ROM builder mode.
DEFINITIONS	(--) Makes the top vocabulary of the search order the current vocabulary.
FI	(--) Enters the interactive Forth mode.
FORTHING	(--) Enters the interactive Forth mode.
HOST	(--) Enters the host's Forth mode.
ONLY	(--) Clears the search order so it only contains the HOME wordlist.
PREVIOUS	(--) Drops the top vocabulary in the search order.
TI	(--) Enters tokenizing mode.
TO	(x <name> --) Stores x to the value <name>.

TOKENIZING	(--) Enters tokenizing mode.
WORDS	(--) Displays all words in the vocabulary at the top of the search order.

HOME glossary

The HOME vocabulary is at the bottom of the search order.

Home Lexicon:

!	(addr - x) Stores a 32-bit value.
#ELSE	(--) Skips source until #THEN is encountered.
#IF	(f --) If f=0, skips source until #THEN or #ELSE is encountered.
#THEN	(--) Does nothing.
$DIS	(t-addr --) Starts disassembly at an arbitrary address. Disassembly will proceed until the address of a known word is reached or the ESC key is pressed.
$HLOAD	($filename --) Loads an Intel HEX file into the ROM image. $filename is a counted string.
$XLOAD-BL51	($filename --) Loads external C labels generated by Keil's 8051 C linker. $filename is a counted string.
(	(--) Skip source until) is encountered.
((	Skip source until)) is encountered.
(SEE)	(<name> --) Decompiles a host Forth word.
(WORDS)	(--) Lists words at the very top of the search order. For example: when in BUILDING mode, you can use (WORDS) to list the builder lexicon.

*****	(x1 x2 -- x3) Multiply. x3 = x1 * x2.
***/**	(x1 x2 x3 -- x4) Multiply and divide. x4 = x1 * x2 / x3.
+	(x1 x2 -- x3) Add. x3 = x1 + x2.
-	(x1 x2 -- x3) Subtract. x3 = x1 - x2.
.	(x --) Displays signed number from top of stack.
.(	(--) Displays source until) is encountered.
.ASMLABELS	(--) Lists all assembler labels.
.FILES	(--) Displays all files included in a build.
.HERE	(--) Displays the value of the target's code pointer.
.HOT	(--) Lists the editor's "hot" keys.
.PROCS	(--) Lists external procedures created by somebody else's assembler or C compiler.
.S	(--) Displays numbers on the stack.
.SCRAM	(--) Displays the ROM scramble table. See SCRAMBLE.
.STATUS	(--) Displays ROM image information.
.VOCS	(--) Lists all created vocabularies.

/	(n1 n2 -- quotient) Signed, floored divide n1 by n2.
//	(--) Skips source until end of line. Like the C version.
/MOD	(n1 n2 -- remainder) Signed, floored divide n1 by n2. Returns remainder.
0<	(x -- f) Returns T if x < 0.
0=	(x -- f) Returns T if x = 0.
1+	(x -- x+1) Adds 1 to top of stack.
1-	(x -- x-1) Subtracts 1 from top of stack.
2*	(x -- x*2) Multiplies top of stack by 2.
2/	(x -- x/2) Divides top of stack by 2 (signed).
<	(x1 x2 -- f)
<=	(x1 x2 -- f) Returns T if x1 <= x2.
<>	(x1 x2 -- f) Returns T if x1 <> x2.
=	(x1 x2 -- f) Returns T if x1 = x2.
>	(x1 x2 -- f) Returns T if x1 > x2.
>=	(x1 x2 -- f) Returns T if x1 >= x2.

>@@0	(x --)
>@@1	(x --)
>@@2	(x --)
>@@3	(x --)
>@@4	(x --)
>@@5	(x --)
>@@6	(x --)
>@@7	(x --)
>@@8	(x --)
>@@9	(x --) Store to local assembler label.
>ADDRNIBBLES	(x --) Set the target's debugger address width in nibbles.
>CELLBITS	(x --) Set the target's bits per cell.
>CHARBITS	(x --) Set the target's bits per character.
>FLAGS	(x --) Store to the flags field of the last created header.
>LITERAL	(x --) Store to the literal field of the last created header.
>PORT#	(x --) Set the parallel port# for a ROM emulator.
>SHOWME	(f --) Stores a flag to a variable that's sometimes used by the compiler to show what's going on. Mostly for debugging compilers.
>TOKEN#	(n --) Store to TOKEN#. Same as {{ TO TOKEN# }}.
@	(addr -- x) Fetch 32-bit value from address.

ADD-TOKEN	(xt spos sid dt cfa <name> --) Adds a token header to the current vocabulary. xt = token#. spos, sid = file position and source file id. dt = datatype cfa = image address.
ADDFILE	(id# <filename> --) Add a filename to the file ID list. See NOFILES.
ADDRNIBBLES	(-- x) Get the target's debugger address width in nibbles.
ADDWATCH	(<name> --) Adds a word to the watch list.
ALIGNMENT	(-- x) Gets the larger alignment value of code and data space.
ALSO	(--) Duplicates the top of the search order.
AND	(x1 x2 -- x3) Logical AND. x3 = x1 & x2.
ANEW	(<name> --) If <name> exists, execute it. Otherwise, compile a word that removes everything after itself in the dictionary.
ASMARRAY	(#bytes <name> --) Defines an assembler label and allots space for it. Use HERE and ORG to set start addresses, etc.
ASMBYTE	(<name> --) Defines an assembler label and allots one byte for it. Use HERE and ORG to set start addresses, etc.
ASMLABEL	(x <name> --) Defines an assembler label, value is x.
ASMLABEL?	(<name> -- x) Look up an existing assembler label.
ASMLONG	(<name> --) Defines an assembler label and allots four bytes for it.

ASMWORD	(<name> --) Defines an assembler label and allots two bytes for it.
B	(--) Browses the current file.
BASE	(-- addr) Address of numeric conversion radix.
BETWEEN	(n1 n2 n3 -- f) Signed compare. Returns T if n1 <= n3 <= n2.
BI	(--) Enters the ROM builder mode.
BINARY	(--) Sets BASE = 2.
BOUNDS>NA	(addr-hi addr-lo -- #cells addr) Converts address bounds to base address and cell count.
BRANCHBYTES	(-- x) Gets # of bytes of a control structure branch, set by the target-specific builder file.
BROWSE	(<filename> --) Browses a file using Winview.
BSAVE	(<filename> --) Saves the ROM image (up to HERE) in binary format.
BUG	(<name> --) Invokes the high level debugger. Opens the source file to the start position and makes sure target is in test mode.
BUILDING	(--) Enters the ROM builder mode.
BYE	(--) Exits to Windows. Same as Alt-F4.
BYTE-JOIN	(lo hi -- x) Joins two bytes to form a 16-bit value.
BYTE-SPLIT	(x -- lo hi) Splits a 16-bit value to form two bytes.

C(	(string --) Store a comment string, up to 32 characters long and delimited by ')', to the comment field of the last created header.
CALIGNMENT	(-- n) Alignment value for code space.
CELL	(-- n) Cell size in address units on the target.
CELLBITS	(-- n) Target bits per cell.
CELLS	(n1 -- n2) Multiplies by the target's cell address units.
CHARBITS	(-- n) Target bits per character.
CHARS	(n1 -- n2) Multiplies by the target's character address units.
CHECKSUM	(addr len -- checksum) Computes the checksum of a block of image data.
CLEARWATCH	(--) Clear the watch list.
CLS	(--) Clear the console.
CODE-BOUNDS	(addr-lo addr-hi --) Set boundaries for code space.
CODE-BOUNDS?	(-- addr-lo addr-hi) Get boundaries for code space.
CODETEMPLATE	(<filename> --) Creates a listing that can be used as a template for somebody else's assembler for porting to a new processor. Not very useful, but you can modify this word to generate other formats.

CONSTANT	(x <name> --) Creates a constant.
CPUTYPE	(-- x) Returns the CPU type being compiled for.
CR	(--) Starts a new line on the console.
CREATE	(<name> --) Creates a data structure.
DALIGNMENT	(-- n) Alignment value for data space.
DATA-BOUNDS	(addr-lo addr-hi --) Sets boundaries for data space.
DATA-BOUNDS?	(addr-lo addr-hi --) Gets boundaries for data space.
DATATYPE	(x --) Stores to the datatype field of the last created header.
DECIMAL	(--) Sets BASE = 10.
DEFINITIONS	(--) Makes the top vocabulary of the search order the current vocabulary.
DROP	(x --) Discards top stack item.
DUP	(x -- x x) Duplicates top stack item.
DYNAMIC	(--) Sets compiler to use binding table.
E	(--) Edits the current file.
EDIT	(<filename> --) Edits a file using Winview.

FI	(--) Enters the interactive Forth mode.
FLOAD	(<filename> --) Loads a Forth source file.
FORTH	(--) Places the FORTH wordlist at the top of the search order.
FORTHING	(--) Enters the interactive Forth mode.
FSEND	(<filename> --) Send a bytecode file to the target and evaluate it.
HEX	(--) Sets BASE = 16.
HEX[	(--) Lays down incoming blank-delimited hex numbers until an unrecognizable number or end of line is encountered. Usage: hex[0AB 07F ... 0FC]
HLOAD	(<filename> --) Loads an Intel HEX file into the ROM image.
HOME	(--) Places the HOME wordlist at the top of the search order.
HOMEORDER	(--) Makes the HOME wordlist the only one in the search order and directs host definitions to the HOME wordlist.
HOST	(--) Enters the host's Forth mode.
HSAVE	(<filename> --) Saves the ROM image (up to HERE) in Intel HEX format.
INCLUDE	(<filename> --) Includes a Forth source file. Same as FLOAD.
INCLUDE-BINARY	(<filename> --) Appends raw binary data to the image.

INVERT	(x1 -- x2) Bitwise-inverts x1. Same as -1 XOR.
IP?	(--) Displays the current IP address of the remote server.
IP=	(<string> --) Sets the current IP address of the remote server.
IP:	(--) Fetches the current IP address of the remote server.
LOADLABELS	(--) Uploads addresses from the target's binding table to enable low level debugging.
LOADROM	(--) Blasts image data out the parallel port to a ROM emulator.
LOW-TOKENS	(--) Selects the low set of token values.
LSHIFT	(n1 n2 -- n3) Logical left shift. n3 = n1 << n2.
MAIN-TOKENS	(--) Selects the main set of token values.
MAKEDEFINING	(<name> --) Places a new defining word in the TOKENIZER wordlist.
MARKER	(<name> --) Compiles a word that removes everything after itself in the dictionary.
MAXUINT	(-- x) Maximum unsigned integer available on target.
NEEDS	(<filename> --) FLOADs a file if it hasn't been loaded yet.
NEGATE	(n -- -n) Negates n.
NEW-IMAGE	(--) Clears the image and pointers.

NIP	(n1 n2 -- n2) Discards second item on stack.
NOFILES	(--) Clears the file ID list.
NOSLACK	(-- addr) Flag: True if warnings abort compilation.
OFF	(addr --) Stores 0 to address.
OK	(--) FLOAD the current file.
ON	(addr --) Stores -1 to address.
ONLY	(--) Clears the search order so it only contains the HOME wordlist.
OR	(x1 x2 -- x3) Logical OR. x3 = x1 | x2.
ORDER	(--) Displays the current search order.
ORG	(t-addr --) Sets new start address.
OVER	(n1 n2 -- n1 n2 n1) Copies second item on stack.
PICK	(... k -- ... n[k]) Copies kth item on stack.
PREVIOUS	(--) Drops the top vocabulary in the search order.
RESET	(--) Drops the target board’s DTR line for ½ second.
ROM-BOUNDS	(addr-lo addr-hi --) Sets boundaries for ROM image.

ROM-BOUNDS?	(addr-lo addr-hi --) Gets boundaries for ROM image.
ROT	(n1 n2 n3 -- n2 n3 n1) Rotate top 3 items on stack.
RSHIFT	(n1 n2 -- n3) Logical right shift. n3 = n1 >> n2.
SCRAM-A	(cpu-pin rom-pin --) Re-map an address bus pin.
SCRAM-D	(cpu-pin rom-pin --) Re-map a data bus pin.
SCRAMBLE	(--) Scramble ROM image for byte-wide ROM
SCRAMBLE-INIT	(--) Clear the ROM scramble table.
SCRAMBLE16	(--) Scramble ROM image for 16-bit-wide ROM
SCRAMBLE32	(--) Scramble ROM image for 32-bit-wide ROM
SEE	(<name> --) Disassembles <name>. If <name> is tokenized code, the bytecode viewer is used. Otherwise, the disassembler is used. Hit ESC to quit early.
SF	(<fracnumber> -- n) Gets a signed fraction between -1.0 and +1.0 from the input stream and converts it to integer for the target's cell width. Example: on a 16-bit target, SF +0.5 maps to 16384.
SHELL	(--) Opens a DOS window.
SSAVE	(<filename> --) Saves the ROM image (up to HERE) in Motorola S format.

STATIC	(--) Sets compiler to ignore (bypass) binding table.
SWAP	(n1 n2 -- n2 n1) Swap the top two stack items.
TAR?	(--) Display rather comprehensive target information.
TE	(--) Enters testing mode. For interactive testing of target words.
TEMP-TOKENS	(--) Selects the temporary set of token values.
TESTING	(--) Enters testing mode. For interactive testing of target words.
TI	(--) Enters tokenizing mode.
TO	(x <name> --) Stores to a value.
TOKEN#	(x --) The next free token#.
TOKENIZING	(--) Enters tokenizing mode.
TRACE	(<name> --) Invokes the low level debugger. Opens the debugger window and displays the disassembled code.
TSAVE	(<filename> --) Saves the token list as an FLOADable file.
TUCK	(n1 n2 -- n2 n1 n2) Tuck the first item under the second.
U2/	(n1 -- n2) Unsigned divide by 2. Same as 1 RSHIFT.

UF	(<fracnumber> -- n) Gets a signed fraction between 0 and +1.0 from the input stream and converts it to integer for the target's cell width. Example: on a 16-bit target, UF 0.5 maps to 32768.
UF1.0	(-- d) Returns double cell value: 2^cellsize.
UM*	(u1 u2 -- ud) Unsigned multiply, result is a double.
UM/MOD	(ud u -- remainder quotient) Unsigned divide: double / single.
UNDERALSO	(--) Duplicates the second wordlist in the search order.
UNDERDEFS	(--) Makes the second wordlist of the search order the current wordlist.
UNDERPREVIOUS	(--) Removes the second wordlist in the search order.
VALUE	(x <name> --) Creates a 32-bit value. Acts like CONSTANT except that you can use TO VALUE to change it.
VARIABLE	(<name> --) Creates a 32-bit variable.
VIEW	(<name> --) View/browse the source code of <name>.
VMTEMPLATE	(<filename> --) Creates a listing that can be used as a template for somebody else's Forth for implementing a new VM. Not very useful, but you can modify it to generate other formats.
VOCABULARY	(<name> --) Creates a word that replaces the top of the search order with its own vocabulary.

WARNING	(-- addr) Flag: True if redefinitions cause a warning message.
WATCHCODE"	(address format string" --) Adds a memory watch record to the watch list. Address is the target memory address to watch, format is how it's displayed, and string" is a quote-delimited description. The format number is two bytes: the upper byte is the display format and the lower byte is the byte count. Formats are: 0=byte, 1=16bit-LE, 2=16bit-BE, 3=32bit-LE, 4=32bit-BE, 5=ASCII.
WATCHDATA"	(address format string" --) Adds a data memory watch record. Example: 0xC000 0x102 WATCHDATA" Location C000 = " adds an item to the watch list that fetches 2 bytes from data memory location 0xC000 and displays the data as a 16-bit little-endian number.
WATCHREG"	(address format string" --) Adds a register watch record.
WATCHEE"	(address format string" --) Adds a non-volatile memory watch record.
WITHIN	(u1 u2 u3 -- f) Unsigned compare. Returns T if u1 <= u3 < u2.
WORD-JOIN	(lo hi -- x) Joins two 16-bit values to form a 16-bit value.
WORD-SPLIT	(x -- lo hi) Splits a 16-bit value to form two 16-bit values.
WORDS	(--) Displays all words in the vocabulary at the top of the search order.
XLOAD-BL51	(<filename> --) Loads external C labels generated by Keil's 8051 C linker.
XOR	(x1 x2 -- x3) Logical exclusive OR. x3 = x1 ^ x2.
[ELSE]	(--) Skips source until [THEN] is encountered.

[IF]	(f --) If f=0, skips source until [THEN] or [ELSE] is encountered.
[THEN]	(--) Does nothing.
{{	(? -- ?) Interpret the input stream until }}. Uses only HOME in the search order and assumes there is no compilation.
\	(--) Skip source until end of line.
\\S	(--) Skip the rest of the file.
}}	(--) Ends a {{ ... }} phrase.
~BOOT0	(--) Downloads the tokenized program in the image to the target board's primary boot area.
~BOOT1	(--) Downloads the tokenized program in the image to the target board's primary boot area.
_DBG	(<name> --) Starts Win32forth's debugger, includes Forth at the top of the search order. To use this, you must set OldStartup to 1 and recompile the Firmware Studio source. This is good for debugging Firmware Studio itself.
_DEBUG	(<name> --) Similar to _DBG but doesn't start debugger until <name> is executed.

Overview:

The basic Firmware Studio debugger protocol is very simple: the host PC sends out a command byte (0xC0..0xFF) and waits for a response byte. This 1:1 correspondence eliminates handshaking and simplifies timing. Firmware Studio performs these byte-wise transactions in groups.

In this packetized extension of the basic protocol, N bytes are encapsulated in a packet and sent to the target board. The target board sends back a packet containing an N-byte response.

The following is a description of what I call BMP, or Brad's Multidrop Protocol. For Firmware Factory, BMP uses 8-bit characters for compatibility with Windows serial port drivers. Debugger commands are 6-bit values, so they are sent out as 7-bit characters 0x40..0x7F. Responses consist of 8-bit data that must be sent using 7-bit characters. Seven bytes are sent as a packed MSB byte followed by seven characters. Bit6 of the MSB byte contains the LSB of the first byte.

8-bit BMP protocol:

The MSB distinguishes addresses from data:

Data:	0x00..0x7F
Addresses:	0x80..0xFB

Address bytes 0xFC thru 0xFF are reserved for physical layer commands:

0xFC = Put RS485 driver into transmit mode.
0xFD = Put RS485 driver into receive mode.
0xFF = Not used, could be caused by a false start bit.

A 9-bit version would use 0x000..0x0FF as data and 0x100..0x1FB for address.

MASTER packet types:

POLL	Addr Addr id:0
RESET	Addr Addr id:1 0 crc crc crc
DEBUG	Addr Addr id:2 length data crc crc crc
DATA	Addr Addr id:3 length data crc crc crc

SLAVE packet types:

NACK	Addr id:6
READY	Addr id:6
EMPTY	Addr id:7
BUSY	Addr id:8
RESPONSE	Addr id:4/5 length data crc crc crc

CRC is 16-bit CCITT. It includes the address, all bytes up to CRC and the ID bit. The CRC is represented by three bytes: [6:0] [13:7] [15:14].

The slave's ID value is updated only when an incoming DATA packet is good. When the slave responds with data of its own, the master will see this ID if the data was received. The master bumps the ID value with each successful transaction. This is done to recognize duplicated packets.

Master	Slave
POLL	BUSY if busy, such as the last packet hasn't been processed yet. EMPTY if ready to receive but nothing to transmit. READY if ready to receive and there is data to send.
DATA	NACK if bad CRC or other reception error. RESPONSE after processing data string (only if the packet ID has changed), save new ID. If no response data is pending, send a RESPONSE to serve as an ACK.
RESET	The slave reboots and comes up with ID=0. Doesn't send anything back to the master.

RESPONSE looks at the incoming ID to recognize repeated packets. The master will repeat a packet if communication times out (Windows steals too much time) or if it misses a mangled RESPONSE packet. A duplicate packet has the same ID, so the data will be discarded. The slave will just send the same response as last time. If the ID is different, the slave will save the new ID value, clear the transmit buffer, process the data and send the response. A repeated packet is identified as type 5 instead of type 4.

If you disrupt communications, the master will keep sending DATA packets until it hears a good RESPONSE packet. Then, it will bump the packet ID. Timeouts, unknown packets, NACKs and mangled RESPONSEs all have the same effect.

A normal bus transaction looks like this:

```
zzzAAILDDDDCCCzzzAILDDDDDCCCzzzz
```

Where z = bus is floating, A=address, I=ID, L=length, D=data, C=CRC.
The first packet is the host, second packet is the target board. The RTS line of the host PC's COM port controls the direction of an RS485 converter dongle. RTS is high while transmitting. The host will allow the PC's TX data to be looped back (the host ignores its own packets).

Host setup and hardware connection:

The default baud, port and address is 19200, COM1 and 0. The multidrop communication mode is selected by the Target Multidrop menu option or by the RS485 command.

485port	(port# --)	Sets the desired COM port, usually 1..4.
485addr	(address --)	Sets the UUT address. 0..123 are usable.
485baud	(baud --)	Sets the baud rate for multidrop communication.

To connect to a target board whose address is 3 on COM2 at 9600 baud, you can enter the following from the keyboard or file:

```
2 485port  3 485addr  9600 485baud  commo=multidrop
```

The RS485 line driver circuit can be built to recognize the T/R commands, so it can operate without the RTS line. This allows remote operation via modem.

The following words may be used to control a modem with the Hayes command set.

MODEM	(--)	Selects modem communication.
BEGIN-COMM	(--)	Opens the modem COM port.
END-COMM	(--)	Closes the modem COM port.
MODEM”	(string” --)	Sends a string to the modem.
MS	(n --)	Produces a time delay of n milliseconds.
RS485	(--)	Selects RS485 communication.
>SmartDongle	(f --)	Enables/disables smart dongle commands.

I haven’t tried running over a modem (as of 8/00), so some of these words aren’t tested. In theory, you should be able to do it without much trouble and it may require tweaking COMMO.G a little.

MAVR.FF contains source for an interrupt-driven state machine that implements the BMP protocol on an AVR microcontroller. It pauses for two character lengths before transmitting to allow for automatic T/R switching in the RS485 converter. A converter with automatic Send Data (SD) control uses a retriggerable one-shot to drive the RS485 bus for at least one character length when a character is transmitted.

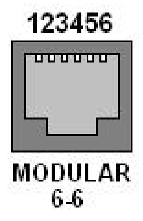
B&B Electronics makes several suitable converters. Visit www.bb-elec.com for application notes about wiring and terminating RS485 multidrop busses.

The SPIX bus:

The SPIX bus is a 4-wire SPI bus that uses a special protocol to allow multiplexed operation. Individual SPI devices are addressed by exploiting the state transitions that are unused in normal SPI operation.

Specified to operate at 100 kHz, the bus is designed to use low cost parts and require neither interrupts, tight timing nor a UART.

Functional units (let's call them boxes) on the SPIX bus are daisy chained. All signals feed straight through except for SS, which is delayed by a D flip flop. A chain of boxes forms a shift register. While SCK is low, the host can twiddle MOSI so as to clear this shift register or clock in a bit pattern that addresses the desired box.

Socket View	Pin#	Upstream connector	Downstream connector
	1	SS-in	SS-out
	2	+5V	+5V
	3	MOSI	MOSI
	4	GND	GND
	5	SCK	SCK
	6	MISO	MISO

When a box contains multiple SPI devices, it addresses individual SPI devices by either decoding more state transitions in hardware or implementing a link protocol using a microcontroller. A box can contain a few SPI peripherals and some decoding logic (no micro) if communication errors won't be a problem.

Each box is assumed to contain an SPI type serial EEPROM containing boot code, or a microcontroller that emulates such an EEPROM.

I implemented some SPIX compatible LCD displays using an SPI type serial EEPROM and an Atmel ATF750LC programmable logic device. This particular CPLD has the footprint of a 22V10 but with higher logic density and independent clock terms. The asynchronous nature of the state decoder made independent clocks necessary, which ruled out using simple PLDs such as a 22V10 or smaller.

The Atmel ATF750LC allowed me to implement a simple SPI interface and some parallel I/O lines, fitting all of the logic onto one chip. The logic is simple enough to economically implement with discrete logic parts if necessary.

Bit errors aren't handled by the SPIX bus. You can re-read an input as a sanity check, but you have to remember that outgoing data may be corrupted by noise or ESD events. The only way around this is to use a microcontroller to implement a link protocol on the slave device.

The slave's boot EEPROM can be re-read, so errors can be eliminated there. Boot code in an error correcting device could modify the I/O routines such that they include a link protocol when talking to the error correcting device. As it stands, the CPLD based decoder doesn't do error correction. It does, however, echo back incoming data so that the SPI firmware can re-send the data before latching it.

The SPIX driver code illustrates the sequencing of the control lines. See the file **`SPIX.F51`** in the **`SPIX8051`** folder.

.

TOF contains many non-ANS words that are used for more efficient execution on small processors. An ANS compliant Forth can be used to test TOF code by loading the file **ANS_TOF.F.** It was tested using Win32forth, which itself contains some non-ANS words that may not be covered here.

When you're trying to write portable Forth code, the question isn't "Is this ANS"? The question is, "What will it take for an ANS Forth to handle this"?

ANS_TOF.F is listed here, with commentary inserted.

MULTI..WHILE..REPEAT

```
\ MULTI..WHILE..REPEAT provides a means to do something zero or more times.
\ For example, 4 >R MULTI R@ REPEAT R>DROP puts 3 2 1 0 on the stack.
\ It translates cleanly to machine code: bump the cell on top of the return
\ stack and branch conditionally.
\ Literal >R and R>DROP also reduce to small code sequences.

: MULTI ( -- )        ( -- R: x -- x' )
        postpone BEGIN
        postpone R>  postpone 1-  postpone DUP  postpone >R
        postpone 0<  postpone 0=
        postpone WHILE ; immediate
```

Since **?DO** does bounds checking (among other things) **?DO ... LOOP** is a very inefficient way to do something zero or more times. **MULTI ... REPEAT** bumps the top of the return stack downward to zero, so **R@** can be used as an index or count value if needed.

Native code to perform this loop structure can be very small. The 68000 TOF kernel uses the following definition for _TYPE, which represents MULTI with two instructions.

```
: _TYPE ( addr len --)
     >r multi   count emit
        repeat  r>drop drop ;

00021802 2F07         _TYPE:   MOVE.L D7,-(A7)              \ >R
00021804 2E1D                  MOVE.L (A5)+,D7
00021806 5397                  SUBQ.L #1,(A7)               \ MULTI
00021808 6B000012              BMI 0x2181C
0002180C 4EB90004FD60          JSR 0x4FD60(xt:112) COUNT
00021812 4EB90004FB80          JSR 0x4FB80(xt:192) EMIT
00021818 6000FFEC              BRA 0x21806                  \ REPEAT
0002181C 588F                  ADDQ.L #4,A7                 \ R>DROP
0002181E 2E1D                  MOVE.L (A5)+,D7              \ DROP
00021820 4E75                  RTS
```

The AVR builder has an optimization that reduces a literal followed by >R to four machine instructions. Similar optimizations can be done on most processors.

MACHINE FORTH POINTERS

```
\ The A register is an idea adapted from machine Forth. Having an A register
\ is very handy for indexing and temporary storage.

variable A
: A!    ( addr -- )     A ! ;                      \ set A register
: A@    ( -- addr )     A @ ;                      \ get A register
: @A    ( -- x ) A @ @ ;                           \ fetch cell from A stream
: @A+   ( -- x ) A @ @  [ 1 CELLS ] LITERAL A +! ; \ lift cell from A stream
: C@A   ( -- c ) A @ C@ ;                          \ fetch char from A stream
: C@A+  ( -- c ) A @ C@ [ 1 CHARS ] LITERAL A +! ; \ lift char from A stream
: !A    ( x -- ) A @ ! ;                           \ store cell to A stream
: !A+   ( x -- ) A @ !  [ 1 CELLS ] LITERAL A +! ; \ append cell to A stream
: C!A   ( x -- ) A @ C! ;                          \ store char to A stream
: C!A+  ( x -- ) A @ !  [ 1 CHARS ] LITERAL A +! ; \ append char to A stream
: @R    ( -- x ) ( R: a -- a )    R@ @ ;           \ fetch cell from R stream
: @R+   ( -- x ) ( R: a -- a+4 )  R@ @             \ lift cell from R stream
                     R> [ 1 CELLS ] LITERAL + >R ;
: !R    ( x -- ) ( R: a -- a )    R@ ! ;           \ store cell to R stream
: !R+   ( x -- ) ( R: a -- a+4 )  R@ !             \ append cell to R stream
                     R> [ 1 CELLS ] LITERAL + >R ;
```

BIT TESTING

```
\ IFSET and IFCLR perform non-destructive bit tests on the top of the stack.
\ They are good for parsing a packed field of bit flags. On processors with bit
\ tests, this produces efficient in-line code.
\ IFSET and IFCLR take a parameter at compile time:
\ [ 3 ] IFSET MyStuff THEN is the same as DUP 8 AND IF MyStuff THEN

: IFSET ( bit# -- )   ( n -- n )
        1 swap lshift
        postpone LITERAL
        postpone OVER postpone AND
        postpone IF ; immediate

: IFCLR ( bit# -- )   ( n -- n )
        1 swap lshift invert
        postpone LITERAL
        postpone OVER postpone AND
        postpone IF ; immediate
```

These are good for decoding bit flags in a packed field. A few machine instructions are all that's necessary to encode one of these structures. Some targets are limited to 8-bit jumps, so be careful not to put a lot of code between **`IFCLR`** or **`IFSET`** and **`THEN`**.

COMPACT CASE STATEMENT

```
\ The [ n ] ?[ ... ]? structure conditionally executes a snippet of code.
\ It compares A to a literal and executes if it's a match. Then it exits.

: QCASE: ( -- )  ( c -- )
        postpone A! ; immediate

: ?[    ( c -- )
        postpone LITERAL
        postpone A@ postpone =
        postpone IF ; immediate

: ]?    ( -- )
        postpone EXIT
        postpone THEN ; immediate
```

`QCASE:` loads an 8-bit value into a register for subsequent tests by **`?[`**. Since some Forth words could trash this register, there are some restrictions on usage. All **`?[ ... ]?`** groups should be grouped together, with the first immediately following **`QCASE:`**.

This can best be illustrated by the following example:

```
: test          ( c -- )
         qcase: [ 1 ] ?[ sqrt ]?
                [ 3 ] ?[ dist ]?
                [ 5 ] ?[ dup over rot ]?
                ;

00A64 2F0A     TEST:    MOV   R16,R26         \ Load R16 with test character (AVR)
00A66 91B9              LD    R27,Y+
00A68 91A9              LD    R26,Y+
00A6A 3001              CPI   R16,1           \ [ 1 ] ?[
00A6C F409              BRNE  0xA70
00A6E CFB5              RJMP  0x9DA:SQRT      \ sqrt  ]?
00A70 3003              CPI   R16,3           \ [ 3 ] ?[
00A72 F409              BRNE  0xA76
00A74 CFDE              RJMP  0xA32:DIST      \ dist ]?
00A76 3005              CPI   R16,5           \ [ 5 ] ?[
00A78 F421              BRNE  0xA82
00A7A 93AA              ST    -Y,R26          \ dup
00A7C 93BA              ST    -Y,R27
00A7E DADA              RCALL 0x34:OVER       \ over
00A80 CD64              RJMP  0x54A:ROT       \ rot ]?
00A82 9508              RET
```

The branches and comparisons built by the qcase structure aren't broken up by Forth calls. As with IFSET, branches can be 8-bit so there will be size restrictions on code within **`?[ .. ]?`**.

The **`QCASE`** structure lays down multiple exit points, so it's usually the last thing in a definition.

MEMORY OPERATORS

```
\ Other TOF kernel words not in ANS or Win32forth:

: C@P           ( addr -- c )   C@ ;    \ char fetch from program memory
: @P            ( addr -- x )   @ ;     \ fetch from program memory
: W@P           ( addr -- x )   W@ ;    \ 16bit fetch from program memory
: C!P           ( c addr -- )   C! ;    \ char store to program memory
: !P            ( x addr -- )   ! ;     \ store to program memory
: W!P           ( x addr -- )   W! ;    \ 16bit store to program memory
: PCREATE       ( <name> -- )   create ;
```

In Forth, code and data memory spaces aren't required to overlap. We assume the ANS Forth that we're extending overlaps code space and data space. Forth operators typically operate on data space. TOF sometimes copies data from ROM to RAM at startup to accommodate this. TOF has special operators to access program space, since data memory and program memory aren't necessarily the same thing.

If you want to create a table in ROM, you should use **PCREATE <name>** to create it and **@P** etc. to read it. These operators work on code (program) space, not data space. A data structure built with **CREATE** uses startup code to initialize RAM from ROM. The data structure is then free to be modified. If this flexibility isn't needed, using **PCREATE** will save ROM and RAM.

BOOLEAN TESTS

```
\ Variables used as Boolean flags are more readable with Boolean-like tests:

: ON?           ( addr -- f )   @ 0<> ; \ flag at addr <> 0?
: OFF?          ( addr -- f )   @ 0= ;  \ flag at addr = 0?
```

These are the complements of ON and OFF, which operate on cell-wide Booleans. For more natural readability, I like to use variable names that are adjectives or end in *ing*.

```
\ Embedded micros can efficiently test bits in memory. So...

: (setupbit)    ( bit# addr -- addr n mask )  DUP >R  1 SWAP LSHIFT  SWAP R> C@ ;

: BIT-ON        ( bit# addr -- )        \ set bit# of char at addr
                (setupbit) OR SWAP C! ;
: BIT-OFF       ( bit# addr -- )        \ clear bit# of char at addr
                (setupbit) INVERT AND SWAP C! ;
: BIT?          ( bit# addr -- f )      \ test bit# of char at addr
                (setupbit) AND NIP 0<> ;
```

These operators work on bit variables in memory. They take advantage of the bit tests provided by most microcontrollers. Bit numbers 0..7 are valid, with 7 being the MSB of a byte.

Note that if you have some control over how the hardware is designed, you can assign each I/O pin its own cell address (use one bit per cell) that's compatible with ON and OFF.

MIXED AND MISC. ARITHMETIC

```
: M/MOD         ( d n -- r q )          ( floored )     \ signed   d/n --> r q
                DUP 0<  DUP>R
                IF      NEGATE >R DNEGATE R>
                THEN    >R DUP 0<
                        IF      R@ +
                        THEN    R> UM/MOD R>
                IF      SWAP NEGATE SWAP
                THEN    ;

: MU/MOD        ( ud# un1 -- rem d#quot )               \ unsigned ud/u --> ur udq
                >R  0  R@  UM/MOD  R>  SWAP
                >R  UM/MOD  R> ;

: U/MOD         ( u1 u2 -- r q )  DROP UM/MOD ;         \ unsigned u/u --> r q

: UD2/          ( d -- d/2 )
                d2/ [ -1 1 rshift ] literal and ;       \ strip MSB
: >>A           ( x count -- )                          \ arithmetic right shift
                -1 OVER RSHIFT INVERT >R  OVER 0< >R
                RSHIFT R> R> AND OR ;

: D0<>          ( d -- f )  D0= 0= ;                    \ true if double-cell <> 0
```

OTHER OPERATORS

```
\ Openboot has COMP, which is the F83 compare, not the ANS compare

: COMP          ( a1 a2 len -- f )   TUCK COMPARE ;


: UNDER1+       ( x1 x2 -- x1' x2 )  >R 1+ R> ;         \ add 1 to NOS
: UNDER1-       ( x1 x2 -- x1' x2 )  >R 1- R> ;         \ subtract 1 from NOS
: >DIGIT        ( n -- c ) DUP 9 > 7 AND + [CHAR] 0 + ; \ convert digit to ASCII

hex
: C>N           ( c -- n )      dup   80 and 0<>   -80 and or ;  \ sign extend
: W>N           ( c -- n )      dup 8000 and 0<> -8000 and or ;
: BYTE-SPLIT    ( n -- cl ch )                          \ split into lo and hi bytes
                DUP 0FF AND SWAP 8 RSHIFT 0FF AND ;
: BYTE-JOIN     ( cl ch -- n )  8 LSHIFT OR ;           \ join lo and hi bytes
: BYTE-SWAP     ( n -- n' ) BYTE-SPLIT SWAP BYTE-JOIN ; \ swap lower 2 bytes of n

: WORD-JOIN     ( nl nh -- n ) 10 LSHIFT OR ;           \ join  lo-16 hi-16 --> 32
: WORD-SPLIT    ( n -- nl nh )                          \ split 32 --> lo-16 hi-16
                DUP 0FFFF AND SWAP 10 RSHIFT 0FFFF AND ;
decimal

: LW,           ( n -- ) byte-split swap c, c, ;        \ comma 16-bit little endian
: C(            postpone ( ; immediate                  \ catalog (summary) comment
: CVARIABLE     variable ;                              \ allots space for 1 character

: {{ ; : }} ; \ ignore {{ }}
```

```
variable debugging
: \D debugging @ 0= if postpone \ then ; immediate
: \ND debugging @   if postpone \ then ; immediate
```

These are in the HOME wordlist. To enable debugging, you can use **`{{ debugging on }}`** anytime. Use **`\D`** to (sometimes) comment out debugging code.

```
: timestamp     ( -- n )
\ Get 32-bit time stamp starting at Jan 1st, 2000.  1-second resolution.
                time&date 2000 -        ( s m h d m y )
                12 * swap +             \ Rolls over every 133 years.
                31 * swap +
                24 * swap +
                60 * swap +
                60 * + ;
```

You can use this to timestamp the ROM image.

FRACTIONAL MATH

```
: uf1.0         ( -- d )   0 1 ;  \ unsigned 1.0

: sf            ( <number> -- n )
\ get signed fractional number ( -1.0 to +1.0 ) from input stream,
\ convert to signed integer ( minint .. maxint )
                bl parse >float 0= abort" Expecting fractional number .XXXXX"
                fdup -1e0 f<
                fdup  1e0 f> or abort" Range must be -1.0 .. +1.0"
                uf1.0 d>f 2e0 f/ f* f>d 0=
                if      dup uf1.0 d2/ drop = if 1- then \ clip +1.0 to maxint
                then    ;

: uf            ( <number> -- n )
\ get unsigned fractional number ( 0 to +1.0 ) from input stream,
\ convert to umsigned integer ( 0 .. umaxint )
                bl parse >float 0= abort" Expecting fractional number .XXXXX"
                fdup  0e0 f<
                fdup  1e0 f> or abort" Range must be -1.0 .. +1.0"
                uf1.0 d>f f* f>d
                if      1-              \ clip +1.0 to umaxint
                then    ;
```

These operators are used to represent fractional numbers in a cell-width independent way. A value between 0 and 1 scales to an integer between 0 and $2^{cellwidth}$ if unsigned. Signed arithmetic represents a value between –1 and 1 as an integer between $-2^{cellwidth-1}$ and $2^{cellwidth-1}$.

In each case, the integer version of +1.0 is bumped down by 1 to avoid an overflow condition. For example, signed +1.0 is changed to +0.99997 with 16-bit cells, or +0.9999999995 with 32-bit cells. If this presents an accuracy problem, scale so as to avoid +1.0.

Local Variables

In Forth, parameters are passed implicitly on the data stack. If there are more than two or three incoming parameters, you can end up generating a lot of stack noise (also called stacrobatics). You can use the return stack to simplify stack manipulation by transferring some data to the return stack. If you use the return stack for temporary storage, it is very important to restore it to its initial state. Any imbalance will most likely crash the system. Treat the return stack like a sharp knife.

DO...LOOP also uses the return stack. If you use >R before a DO loop, R@ will return a different result inside the loop. You can use the return stack inside a DO loop as long as you take off whatever you put on before LOOP.

If stack manipulation gets tedious, you probably haven't factored the problem properly. It should cause you to ask, "What can I factor out of this"? There are some situations where the answer is "Nothing". For cases like these, there is local variable support.

The ANS standard provides for simple local variable support. At the beginning of a subroutine, parameters may be moved to the return stack. Locals are variables that reside on the return stack. They act like VALUEs in that you use TO to store to them. At the end of the subroutine, the return stack is cleaned up. Locals are basically a functional replacement for return stack manipulation.

The proposed ANS syntax is **`LOCAL| arga argb |`**
where **`arga`** is the value on the top of the data stack and **`argb`** is next on the stack.

This is backwards from the stack picture comment **`( argb arga -- ? )`**.

We use a modified version whose syntax is:

- **`{ argb arga \ argc -- }`**

Here, { behaves like LOCAL| except that it reverses the stack order and allows comments. So, it looks like a stack picture. The example shown above is handled as follows:

1. Uninitialized storage is allocated for argc on the return stack. Marked by **`\`** .
2. Arga and Argb are removed from the data stack and placed on the return stack.
3. Everything between **`--`** and } is treated as a comment.

Locals are especially useful when working with data points. For example, imagine coding the following with the usual stack manipulation words:

```
: distance { x0 y0 x1 y1 – dist }  \ d = sqrt(deltaX^2+deltaY^2)
     x1 x0 – abs dup um*
     y1 y0 – abs dup um* d+ sqrt ;
```

Here is an example showing what the 68K builder compiles when locals are used.

```
: dum+ { a b \ c -- sum } \ a version of +
     a b + to c
     c ;
```

```
DUM+: SUBQ.L #4,A7         Uninitialized storage for  c
      MOVE.L D7,-(A7)      Push  a  and  b  onto the return stack.
      MOVE.L (A5)+,-(A7)
      MOVE.L (A7),D7       Fetch  a
      MOVE.L D7,-(A5)
      MOVE.L 4(A7),D7      Fetch  b
      ADD.L  (A5)+,D7      Add
      MOVE.L D7,8(A7)      TO c
      MOVE.L 8(A7),D7      Fetch  c
      ADDQ.L #8,A7         ;  clears the return stack.
      ADDQ.L #4,A7
      RTS
```

It is assumed that the CPU will use stack-relative addressing to access locals. R@, R>, R> and other return stack manipulators may throw off the indexing so be aware of this.

The compiler compensates inside of DO..LOOP structures so that locals may be used inside of DO loops.

Locals support for various CPUs are handled via four defered words:

Local-begin	Compiles code to allocate uninitialized storage and transfer data to the return stack.
Local-end	Compiles code to remove data from the return stack.
Local-fetch	Compiles code to fetch a local variable to TOS.
Local-store	Compiles code to store TOS to a local variable.
Local-base	Compiles code to fetch the base address of a local array.

For 8051, AVR and 68K CPUs, locals need the files LOCALS.F51, LOCALS.FAR and LOCALS.FCF, respectively. They contain run-time code for the above-named words. For the 8051 and 68K, they also contain support for locals in tokenized code.

Firmware Studio and Win32forth support both syntaxes. The { } syntax is easily implemented and fairly common, so you should be safe using it. In other words, your code won't be locked into a particular Forth.

On the 68K, TOF allows a word to allocate an array of cells on the data stack.

- **`{CELLS}  ( #cells -- )`**

This is an immediate word that compiles code to return a base address. For example:

```
: FOO { \ mydata -- }
  [ 16 ] {cells} to mydata
;
```

Multitasking in Firmware Studio

Cooperative multitasking uses **`PAUSE`** to switch between active tasks. Each task has a unique task ID (tid), which is the address of the task's TCB (Task Control Block). The structure is shown below, with the offset in cells.

Offset	Contents	Save/Reload sequence
0	SP0: initial data stack pointer	
1	RP0: initial return stack pointer	
2	PC: Current program counter	**1** **2**
3	SP: Current data stack pointer	**1** **2**
4	RP: Current return stack pointer	**1** **2**
5	Status: Lowest byte in memory = **`-1`** to activate.	**2**
6	Link: points to status of the next TCB	**1**

`PAUSE` is the key to Forth's multitasking. A context switch is very fast because there's not much context to switch. With applications typically being I/O bound, real-time events can get quick response with low multitasking overhead. Each task must contain a **`PAUSE`** so as to share the CPU with the others. As with other things Forth, the price of this kind of performance is more responsibility on the part of the programmer.

All tasks must periodically invoke **`PAUSE`** by performing an I/O word or by executing **`PAUSE`** or **`STOP`**. **`STOP`** is like **`PAUSE`** except that it deactivates the current task before performing **`PAUSE`**.

`PAUSE` performs one pass around the round robin queue. Quick response to real-time events relies on rapid handoff between tasks, which **`PAUSE`** does in less than a microsecond on modern 32-bit processors.

`PAUSE` saves the context of task 1 and tests the new status for task 2. If this status is inactive, **`PAUSE`** skips to the next TCB and so on until it finds an active task. When an active task is found, **`PAUSE`** loads the context for the task.

Only the first byte of the status cell is used. This makes it easy to compile assembly code to wake up a task. An ISR that wants to wake up a task only needs to clear this byte. A Task called **`MyTask`** can be activated by **`0 MyTask C!`**.

The 8051 version doesn't have a multitasker because the stacks are pretty limited. As of 8/99, the basic 68000 multitasker is in place but the rest hasn't been defined yet. In an 8051 application, use PAUSE to handle background stuff.

One of the CPU’s registers is reserved for the task pointer. This points to offset 2 (PC) in the current TCB. PAUSE quickly traverses the task list to find the next active task. For example, the 68000 code for PAUSE is:

```
code PAUSE ( -- )
           move.l (sp)+,(tp)+    \ save PC
           move.l tos,-(s)
           move.l s,(tp)+        \ save data stack
           move.l sp,(tp)+       \ save return stack
           addq.l #4,tp
   begin   movea.l (tp),tp       \ point to next task's status
           tst.l (tp)+           \ until active task is found
   until_mi subq.l #4,tp
           movea.l -(tp),sp      \ load new return stack
           movea.l -(tp),s       \ load new data stack
           move.l (s)+,tos
           move.l -(tp),a0       \ load new PC
           jmp (a0) c;
```

The task queue consists of a circularly linked list of TCBs pointed to by a task pointer. PAUSE saves the state of the current task, looks around for an active task, and loads the state of the new task.

A task must invoke PAUSE periodically. Since most tasks are I/O bound, they tend to PAUSE a lot. Context switches occur rapidly and with low overhead. This avoids the heavy processing demands of preemptive schedulers, giving more resources to the application. One common technique is to use an ISR to handle the time critical part of a task. The ISR wakes up a task that finishes the job.

As with all things Forth, it is the programmer’s responsibility to ensure that no task monopolizes the CPU and none are starved for CPU time.

Firmware Studio multitasking words:

`NEWTASK  (usize ssize rsize <name> -- ) ( -- tid )`
Creates a TCB containing space for user variables, data stack and return stack. Sizes are in bytes. When evaluating tokenized code, **`<name>`** is replaced by **`<xt>`**.

`ALSOTASK ( tid -- )`
Adds a given TCB it to the circular list of tasks. The TCB is inserted into the queue's linked list between the currently executing task and the next task. You can't remove a TCB from the queue, but you can **`SLEEP`** or **`REASSIGN`** it. A word to "kill" a TCB (remove it from the queue) is feasible, but not necessary since a sleeping TCB isn't much more expensive than a dead TCB.

`REASSIGN ( xt tid -- )`
Reassign a task that's in the queue and clear its stacks.

`SLEEP    ( tid -- )`
Put a task to sleep.

`WAKE     ( tid -- )`
Wake up a task.

`LOCAL    ( tid n -- a )`
Access another task's user variables

`PAUSE    ( -- )`
Perform one pass around the round robin queue.

`STOP     ( -- )`
Sleep current task, do pause.

`USER     ( offset -- ) ( -- a )`
Compiles a word whose runtime semantics are similar to **`VARIABLE`** but returns a base address plus offset. An offset of zero gives the address of cell 7 of the TCB.

`SEMAPHORE ( -<name>- )`
Define a semaphore for resource arbitration. Same as **`VARIABLE`**.

`SEM-GET  ( semaphore -- )`
Get a semaphore, locking out other tasks from using the resource related to this semaphore.

`SEM-RELEASE ( semaphore -- )`
Release a semaphore, allowing other tasks to access the resource related to this semaphore.

Sample usage:

```
 0 cells user X
 1 cells user Y
 2 cells user Z

 3 cells constant usersize
16 cells constant sstacksize
16 cells constant rstacksize

usersize sstacksize rstacksize NEWTASK taskA
usersize sstacksize rstacksize NEWTASK taskB


: TaskAdemo     ( -- )  begin X @ Y @ 2* + Z ! PAUSE again ;
: TaskBdemo     ( -- )  begin X @ Y @    - Z ! PAUSE again ;

: InitStuff     ( -- )
      taskA ALSOTASK
      taskB ALSOTASK
      ['] TaskAdemo taskA REASSIGN
      ['] TaskBdemo taskB REASSIGN
      taskA WAKE
      taskB WAKE ;
```