$10 Volume XVIII, Number 2 July 1996 August

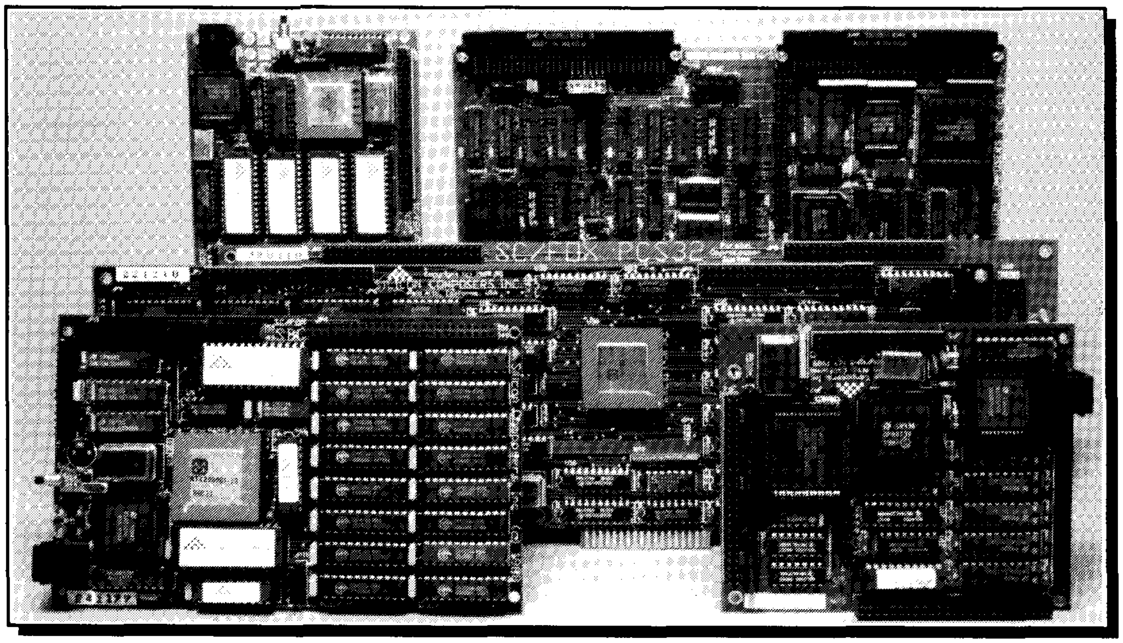
FORTH DIMENSIONS

SILICON COMPOSERS INC

FAST Forth Native-Language Embedded Computers

DUP

>R



C@

R>

Harris RTX 2000tm 16-bit Forth Chip
- •8 or 10 MHz operation and **15 MIPS** speed.
- •1-cycle 16 x 16 = 32-bit multiply.
- •1-cycle 14-prioritized interrupts.
- •two 256-word stack memories.
- •8-channel I/O bus & 3 timer/counters.

SC/FOX PCS (Parallel Coprocessor System)
- •RTX 2000 industrial PGA CPU; 8 & 10 MHz.
- •System speed options: 8 or 10 MHz.
- •32 KB to 1 MB 0-wait-state static RAM.
- •Full-length PC/XT/AT plug-in (6-layer) board.

SC/FOX VME SBC (Single Board Computer)
- •RTX 2000 industrial PGA CPU; 8, 10, 12 MHz.
- •Bus Master, System Controller, or Bus Slave.
- •Up to 640 KB 0-wait-state static RAM.
- •233mm x 160mm 6U size (6-layer) board.

SC/FOX CUB (Single Board Computer)
- •RTX 2000 PLCC or 2001A PLCC chip.
- •System speed options: 8, 10, or 12 MHz.
- •32 KB to 256 KB 0-wait-state SRAM.
- •100mm x 100mm size (4-layer) board.

SC32tm 32-bit Forth Microprocessor
- •8 or 10 MHz operation and **15 MIPS** speed.
- •1-clock cycle instruction execution.
- •Contiguous 16 GB data and 2 GB code space.
- •Stack depths limited only by available memory.
- •Bus request/bus grant lines with on-chip tristate.

SC/FOX SBC32 (Single Board Computer32)
- •32-bit SC32 industrial grade Forth PGA CPU.
- •System speed options: 8 or 10 MHz.
- •32 KB to 512 KB 0-wait-state static RAM.
- •100mm x 160mm Eurocard size (4-layer) board.

SC/FOX PCS32 (Parallel Coprocessor Sys)
- •32-bit SC32 industrial grade Forth PGA CPU.
- •System speed options: 8 or 10 MHz.
- •64 KB to 1 MB 0-wait-state static RAM.
- •Full-length PC/XT/AT plug-in (6-layer) board.

SC/FOX SBC (Single Board Computer)
- •RTX 2000 industrial grade PGA CPU.
- •System speed options: 8, 10, or 12 MHz.
- •32 KB to 512 KB 0-wait-state static RAM.
- •100mm x 160mm Eurocard size (4-layer) board.

For additional product information and OEM pricing, please contact us at:
SILICON COMPOSERS INC 655 W. Evelyn Ave. #7, Mountain View, CA 94041 (415) 961-8778

Contents

Editorial

Input and Outreach

In our continuing search for articles on a variety of topics, we were pleased to receive Ron Kneusel's contribution about Forth CGI scripts for Web pages. The Forth Interest Group's collection of pages already incorporates form handlers written in Forth, and it was good to find additional work being done in this contemporary field. See the article in this issue to learn how CGI expands the capabilities of Web publishers and users.

A key to the World Wide Web's dynamic nature is the hyperlink. (Mouse users click on links to follow pathways of information from computer to computer.) What gives a Web page value—its content—doesn't mean it can be found easily by net denizens. To get readers, one can publish the Internet-style address (e.g., FIG's http://www.forth.org/fig.html) and convince designers of other, related Web pages to include a hyperlink to your site. In the first case, people already need to be reading information you publish in order to find your Web site; in the second case, your contact information is included on sites where potentially thousands of new people can find you.

Now that Forth has a distinct presence on the net, it's important to let others know about it. During your Internet explorations and research, keep in mind that you can ask a relevant site's administrator to include a link to the non-profit Forth Interest Group's site. Once there, browsers will encounter a plethora of info about Forth and FIG, a good deal of software, and links to other Forth pages. Asking someone to install a link to the FIG Home Page will make it—and Forth—accessible to many more people.

As one letter writer notes in this issue, people only learn about Forth if we educate them. And a larger community will preserve our ability to continue and expand *all* our activities, both on-line and off.

—Marlin Ouverson, Editor
editor@forth.org

dot-quote

...Forth in itself is an application. In my opinion, Forth would be an excellent shell in a Un*x environment....it probably will not replace C in such systems, but it can find its place in a toolbox among *sh*, *awk*, and *perl*.

—András Zsótér (h9290246@hkuxa.hku.hk)

After reading *The UNIX Philosophy* by Mike Gancarz, I reached similar conclusions. Forth fits in, as I view it now, as a scripting language that really hits its stride in hardware-oriented or constrained applications, or where operations on files are largely irrelevant. When Forth programmers complain that the code they wrote with such painstaking attention to readability was rewritten several years later in C, they are complaining that it has the typical life cycle of scripting languages. An SL quickly proves that a problem can be solved, and does so with minimal development resources. Later, when other priorities (like availability of non-specialist programmers to maintain it and fine-tune performance) become higher, they re-code it in C and assembly. Sometimes the SL fits the problem so well that this final stage is unnecessary or is a step backwards in performance.

...Forth, to some extent, developed the notions behind Un*x *alias* and MS-DOS *doskey* but earlier and more comprehensively and efficiently, building a whole language around short, modular macros. The language the Forth programmer has available for writing macros and compiler directives, unlike C, is a complete language with sequential, iterative, and logical alternative flow control available, as well as the full set of normal operators. Hence, the language can transmute itself as needed without having to write a special preprocessor file operation, and simulates the ability of the human mind to alter its beliefs, viewpoints, and knowledge as experience dictates. I believe this last point is part of the fascination of Forth.

—Dallas E. Legan (dlegan@heart.engr.csulb.edu)
Adapted from comp.lang.forth with permission.

Forth Dimensions
Volume XVIII, Number 2
July 1996 August

Published by the
Forth Interest Group

Editor
Marlin Ouverson

Circulation/Order Desk
Frank Hall

Forth Dimensions welcomes editorial material, letters to the editor, and comments from its readers. No responsibility is assumed for accuracy of submissions.

Subscription to *Forth Dimensions* is included with membership in the Forth Interest Group at $45 per year ($53 Canada/Mexico, $60 overseas air). For membership, change of address, and to submit items for publication, the address is: Forth Interest Group, P.O. Box 2154, Oakland, California 94621. Administrative offices: 510-89-FORTH Fax: 510-535-1295

The Forth Interest Group
The Forth Interest Group is the association of programmers, managers, and engineers who create practical, Forth-based solutions to real-world needs. Many research hardware and software designs that will advance the general state of the art. FIG provides a climate of intellectual exchange and benefits intended to assist each of its members. Publications, conferences, seminars, telecommunications, and area chapter meetings are among its activities.

"*Forth Dimensions* (ISSN 0884-0822) is published bimonthly for $45/53/60 per year by the Forth Interest Group, 4800 Allendale Ave., Oakland, CA 94619. Second-class postage paid at Oakland, CA. POSTMASTER: Send address changes to *Forth Dimensions*, P.O. Box 2154, Oakland, CA 94621-0054."

Letters

Send your feedback, questions, criticisms, and other responses to editor@forth.org or to the editor in care of the Forth Interest Group, P.O. Box 2154, Oakland, California 94621. Letters may be edited for clarity and length.

Mousetrap Myth

Dear Editor,

I enjoyed reading Phil Monson's letter in the May/June issue. His is an interesting and sadly typical Forth story.

My question to Phil is, who knew about his great success? Where did he publish the papers, in what magazines did the articles appear? If he was counting on his colleagues and superiors to spread the word, I am not surprised he was disappointed that the word didn't spread!

We are all too ready to believe the myth that excellence is its own reward, "if you build a better mousetrap the world will beat a path to your door." The problem is that, somehow, the world needs to know about this mousetrap.

The answer to Phil's basic question, which is an important one, is that Forth "has not come into wider acceptance" because successes such as Phil's are not communicated to the world at large. Any one of us who knows such stories, and cares about the world knowing, should assume responsibility for telling the world. There are many ways to do this, from articles submitted to academic publications and the trade press, to letters to the editor, to Internet newsgroups (and not just comp.lang.forth!).

Every Rochester conference, and many issues of *FD*, features such success stories. Unfortunately, in these forums we are only preaching to ourselves. Tell everyone else! If you need specific suggestions as to how, please let me know. I'd love to help!

—*Elizabeth Rather (erather@forth.com)*

Java Learns from Forth, Forth May Benefit, Too

Reading the feature story on Sun executive Scott McNealy in *Business Week* (22 Jan. '96), one comment stuck in my mind. The son of an American Motor Corporation executive stated, "If you're going to make automobiles, you have to make your own engine."

At the time, I thought this dealt with Sun's attempts to buy out Apple. What he really meant was clarified by a one-page article in the April 1996 issue of *Electronic Products*, "Hardware processors poise to widen Java appeal" by Rodney Myrvaagnes, announcing Sun's alternative to their own Java Virtual Machine. A careful rereading of the *Business Week* article showed that this possibility was very briefly mentioned. The gradually accelerating hype, including the "Net Computer" and the sudden bombardment of books by SunSoft Press, then snapped into focus for me.

The *Electronic Products* article spent about a paragraph discussing the precedent set by Forth for language-specific processors, briefly mentions Sun's previous involvement with Forth—and then backpedals, arguing that these points were of little help, since Java is an "object language." (I'm always a bit shocked at people's inability to realize what a truly extensible language is capable of.) There was no mention in *Electronic Products* that this language-specific processor development has been going on for roughly ten years in the Forth world.

One of the main themes of the *Business Week* article was Sun's carefully calculated assault on the Wintel computer model, and I think this may be good news for the future of other language-specific chips. There already has been a fair amount of posting on comp.lang.forth for Forth-to-Java byte-code compilers and vice-versa. I don't really know what finding another language generating byte code for its machines (virtual or actual) would do to Java's security measures, or whether this capability would suddenly open a lot of new opportunities for Forth. I think it would have big implications for both. Remote procedure calls would seem to be fairly trivial compared to sending an entire program over a network, and being able to send Forth object code through a Java byte-code verifier to check safety might be useful for some Forth applications. I remember someone, at a Los Angeles FIG meeting years ago, saying that Forth, as an extension of the debug monitor when you first turn the processor on, will always be at the cutting edge of hardware development.

—*Dallas E. Legan (dlegan@heart.engr.csulb.edu)*

Product Watch

May 1, 1996—Forth, Inc. signed an agreement with Creative Solutions, Inc. (CSI) to acquire CSI's Forth systems for Macintosh computers. These include MacForth Plus (first introduced in 1984) and Power MacForth (1994) for Power Macintoshes. Both are fully integrated with the Mac environment, with simple access to Mac Toolbox functions.

The original MacForth was the first resident software development product for Macintoshes, and has been used in large-volume spreadsheets, rendering and design, CAD/CAM, games, medical diagnostics, image enhancement, accounting, desktop planetariums, and process control. Power MacForth is a native Power Macintosh system that features high-speed execution, internal multitasking, and a RISC assembler providing direct access to the native CPU architecture for maximum performance.

Forth, Inc.'s product lines include polyForth for PCs, chipForth cross-development systems for microcontrollers, Express for industrial control applications, and ProForth for Windows-based applications. The company also provides application design, programming courses, and custom software development.

Forth, Inc.
111 N. Sepulveda Blvd. #300
Manhattan Beach, California 90266
800-55-Forth or 310-372-8493
http://home.earthlink.net/~forth

Stretching Forth ☐ STANDARD FORTH

Circular String Buffer

Wil Baden
Costa Mesa, California

This month's article features a classic gem of Forth code—a circular string buffer. It or an equivalent implementation is built into several veteran systems and is well known to most Forth masters. However, it is valuable to realize it in Standard Forth. To me it is necessary for the kind of work I do with Forth: massaging text files and macro processing. I want to make sure that it is available for me in future articles.

I use it to make a replacement for the function of PAD without the shortcomings of PAD.

It is used to move strings from the input source or other transient area to a safe place.

An immediate use of >PAD is to implement interpretation semantics for S" as in the File Access wordset. The circular string buffer provides buffers as needed, so S" can be safely used several times interpretatively without fear of overwriting.

As long as we're in the neighborhood, we tweak S" to let us quote quotes. S is like S" except that it takes as its delimiter the next character in the parse area. PARSE-CHAR is a Standard Forth kludge to get the next character from the parse area. Eventually we'll want something better.

Listing One. Circular string buffer.

```
( Circular String Buffer )

    1000 CONSTANT /CSB

    CREATE CSB   0 ,   /CSB CHARS ALLOT

    : GET-BUF                                  ( n -- c_addr )
         DUP CSB @ > IF   /CSB CSB !   THEN
         NEGATE CSB +!                          ( )
         CSB CELL+  CSB @ CHARS +               ( c_addr)
    ;

: >PAD                                         ( s u -- s' u )
    DUP GET-BUF SWAP                            ( s s' u)
    2DUP >R >R
         CHARS MOVE                             ( )
    R> R>                                       ( s' u)
;
```

/CSB is the size of the buffer. The value used here has been more than adequate for everything I've done, and could probably be much less.

GET-BUF takes a number and returns a character address within the circular string buffer. The number is how many characters can be moved there. Error checking is left as an exercise to the reader.

GET-BUF is an implementation factor of >PAD and S+ (catenation). GET-BUF has CSB and /CSB as implementation factors.

>PAD finds room for a string in the circular string buffer, copies it there, and returns the address and length.

For instance,

```
    S |Don't say "Hello".| TYPE
```

displays

```
    Don't say "Hello".
```

as desired.

The delimiter for S can be a blank, as in S file-name INCLUDED for example. Note that there are exactly two spaces after S but only one space is needed before INCLUDED.

S "ccc" is the same as S" ccc".

Listing Two. String literals.

```
: S"                          ( "ccc<quote>" -- | c_addr u )
    [CHAR] " PARSE                  ( somewhere_in_input_buffer u)
    STATE @ IF
         POSTPONE SLITERAL ( )
    ELSE
         >PAD                  ( c_addr u)
    THEN
; IMMEDIATE

    : PARSE-CHAR                                    ( -- char )
         ( Beware: Undefined at end of parse area. )
         SOURCE DROP >IN @ CHARS + C@               ( char)
         1 >IN +!
    ;

: S                           ( "<char>ccc<char>" -- | c_addr u )
    PARSE-CHAR PARSE                ( somewhere_in_input_buffer u)
    STATE @ IF
         POSTPONE SLITERAL ( )
    ELSE
         >PAD                  ( c_addr u)
    THEN
; IMMEDIATE
```

The circular string buffer is used for string catenation. For instance, to put quotes around a string:

```
      S '"' 2SWAP S+ S '"' S+
```

To replace the last three characters of a string with "4th", shorten the string and catenate "4th":

```
      3 - S" 4th" S+
```

S+ finds room in the circular string buffer for the catenated string and moves it in.

Listing Three. String catenation.

```
: S+                                       ( s1 u1 s2 u2 -- s3 u3 )
     2 PICK OVER + ( = u1+u2) >R                 ( R: u1+u2)
     R@ GET-BUF >R                               ( R: u1+u2 buf)
          2 PICK ( = u1) CHARS R@ +         ( . . s2 u2 buf+u1)
          SWAP CHARS MOVE                   ( s1 u1)
          R@ SWAP ( s1 buf u1) CHARS MOVE   ( )
     R> R>                                  ( s3 u3) ( R: )
;
```

You may want to replace PAD with a definition based on CSB. This PAD returns the same address as the last >PAD or GET-BUF.

(Continued.)

Wil Baden is a professional programmer with an interest in Forth.
wilbaden@netcom.com

Listing Four. Example.

```
: PAD   CSB CELL+  CSB @ CHARS + ;      ( -- buf )

     128 CONSTANT line-length
     : checked ABORT" (File Access Error) " ;
     0 VALUE IN   ( File-id )

: numbered-list                         ( -- )
     0 BEGIN                                 ( lnum)
          line-length 2 + GET-BUF            ( lnum buf)
               line-length IN READ-LINE checked
     WHILE                                   ( lnum len)
          1 UNDER+
          DUP IF
               OVER 4 .R   SPACE
          THEN
          PAD SWAP ( . buf len) TYPE         ( lnum)
     REPEAT                                  ( lnum len)
     2DROP                                   ( )
     IN REWIND
;

S  csb.4th R/O OPEN-FILE checked TO IN
numbered-list
```

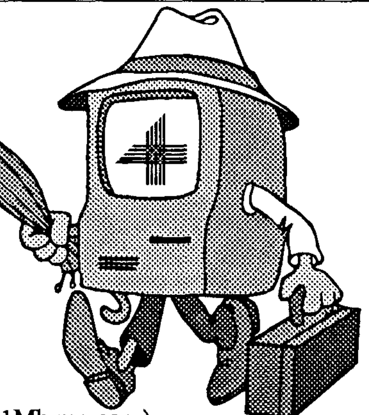
Appendix

```
: UNDER+   ROT + SWAP ;                    ( a b c -- a+c b )
: REWIND   ?DUP IF 0 0 ROT REPOSITION-FILE ABORT" Can't rewind. " THEN ;
```

NOW from FORTH, Inc....

MacForth & Power MacForth

...give you the easiest-to-use programming software for the easiest-to-use PCs!



- MacForth for all Macs (>1Mb memory)
- Power MacForth for fast, optimized native Power PC code
- Full Mac Toolbox support, including System 7 PPC interface
- Powerful multitasking support
- Integrated source editor, trace & debugging tools
- High-level graphics and floating point libraries
- Wealth of demo programs, source code & examples
- Extensive documentation, including online Glossary
- Turnkey capability for royalty-free distribution of programs

FORTH, Inc.
111 N. Sepulveda Blvd, #300
Manhattan Beach, CA 90266
800-55-FORTH 310-372-8493
FAX 310-318-7130 forthsales@forth.com
http://www.forth.com

The Computer Journal

Support for older systems
Hands-on hardware and software
Computing on the Small Scale
Since 1983

Subscriptions
1 year $24 - 2 years $44
All Back Issues available.

TCJ
The Computer Journal
P.O. Box 3900
Citrus Heights, CA 95611-3900
800-424-8825 / 916-722-4970
Fax: 916-722-7480
BBS: 916-722-5799

PART TWO

Safety Critical Systems

Paul E. Bennett
Bristol, United Kingdom

[Mr. Bennett's article in the preceding issue should have been accompanied by an illustration depicting an engineering organization—see Figure Three of this installment. We regret its previous omission and any confusion this may have caused.]

System Architecture and Organisation

Systems, societies, and manufacturing companies are all organised in some fashion. There are many models of implementing the management structure. Likewise, a control system, like a management system, can be based on many models. Two main models seem to predominate in both management and control systems. These are the Flat Model and the Hierarchical Model.

Flat Model Control Systems

A few types of control system requirements are simple enough for a flat control model (see Figure One) to be adopted. A flat model incorporates the user interface, process interface, and controller all in the same package (probably with the same microprocessor at its heart). Examples of such a control system may be the oven timer for a cooker unit or the controls of a washing machine. This is not necessarily the case, but it may be.

Hierarchical Process Control Systems

For many process control situations, there is a distinctive hierarchy to the plant (see Figure Two). This hierarchy should also be reflected in the control system organisation. The top of the pyramid will be the Control Room and Management Information Systems. At the bottom layer will be the measuring instrumentation, control actuators, and the protection systems. In between may be group control systems, which organise the gathering and grouping of collected data and the overseeing that limits are not exceeded by the underlying controllers (by calling upon protective measures to be taken).

In the diagrams for both the above models, I have shown a Safety Interlocking Network. This can be likened to the outside influences on a company, such as the manufacturing company of Figure Three, by imposition of standards and legislation through the normal legal processes. It is where a lot of analysis and design effort needs to be concentrated to establish a truly protective system. For many installed systems, it can be difficult to identify these protective measures because they seem to be part of the control system itself.

Hardware and Software Selection

Choice of hardware and software for a Safety Critical System should be made as a combination. No software can run without supporting hardware, and programmable hardware does not do anything without a programme. Little thought should be given in the early stages of a design as to whether functions will be performed by hardware or software. Looking on the design exercise as the generation of a specification that will enable the construction of a system to perform the required functions,

Figure One. Flat-model-architecture control system.

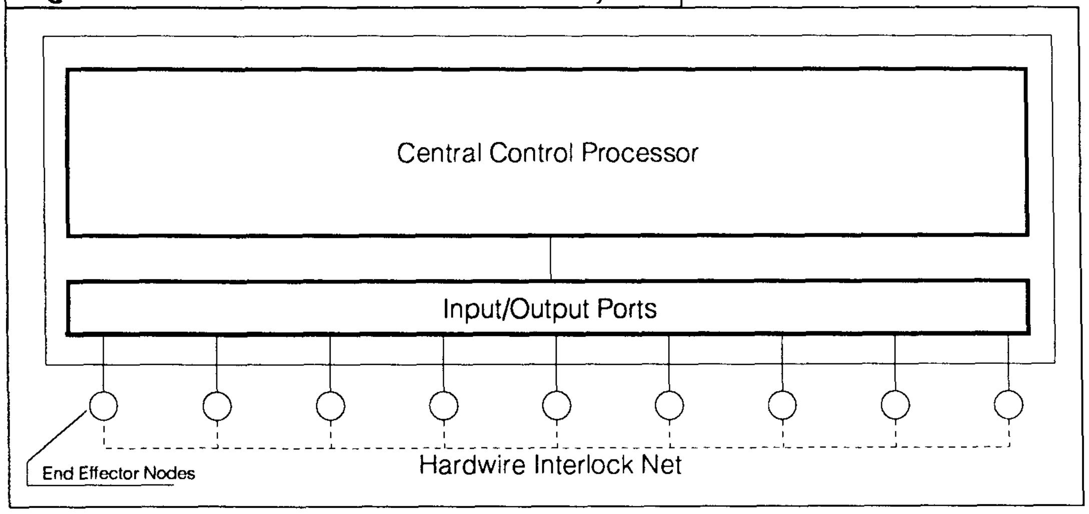


Figure Two. Hierarchical-model control system.

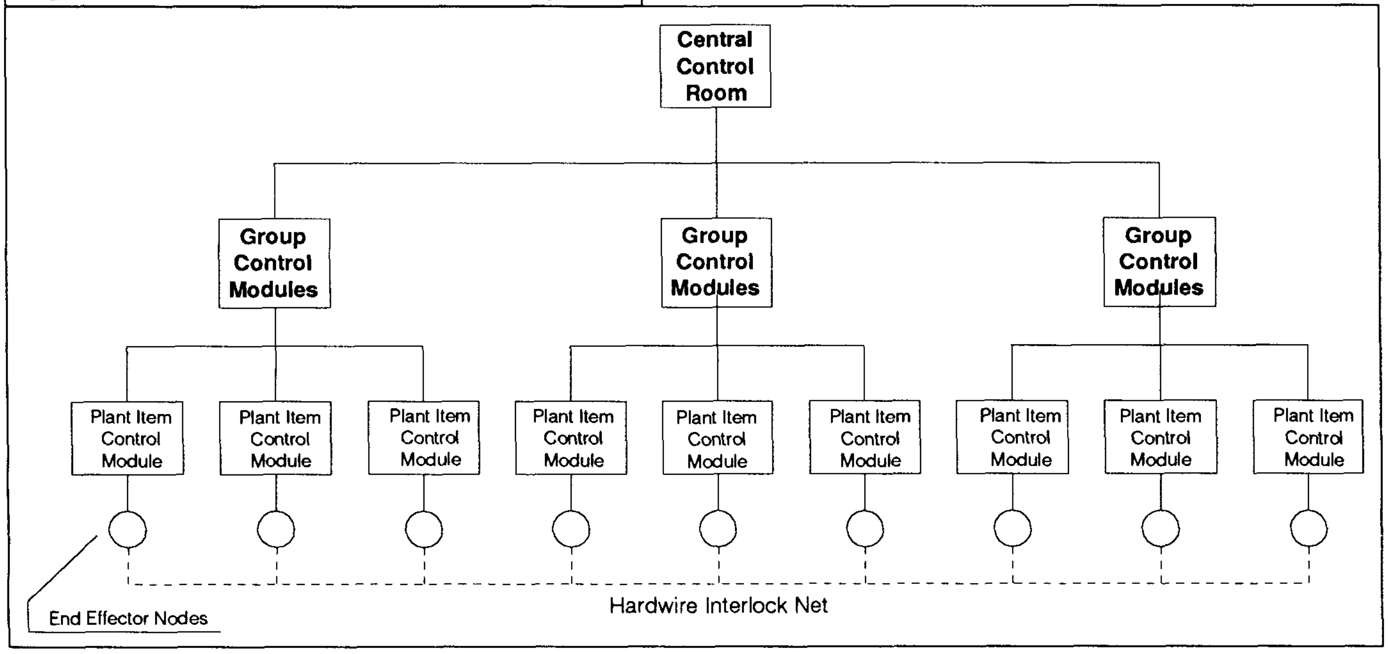
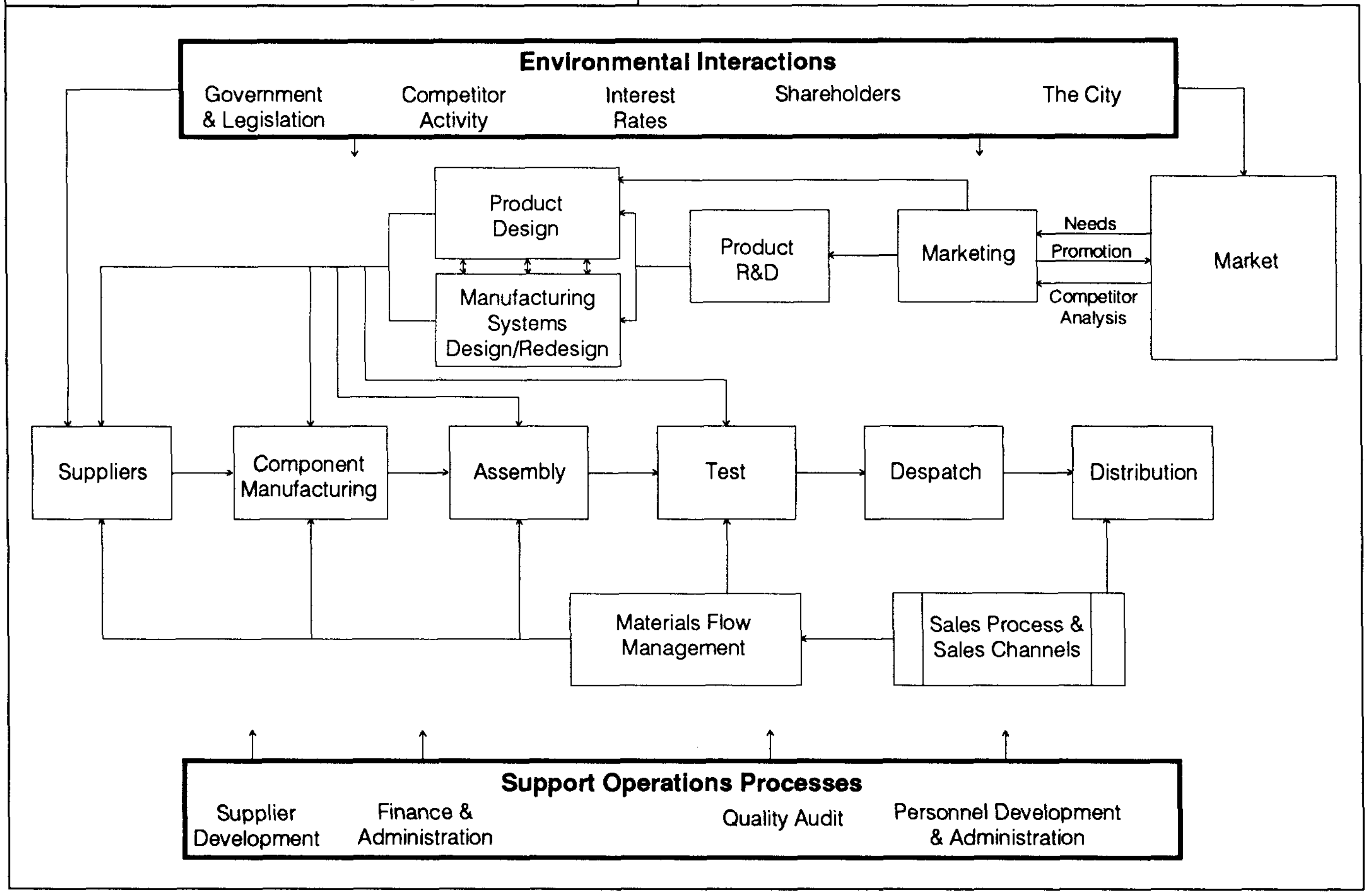


Figure Three. Engineering organisation model.



the whole task can be considered software (or, better still, logics) until definite decisions can be made about what to migrate to hardware and what to keep in software.

It is important that the system hierarchy, once designed, should be carried through the system design. This also enables the partitioning of the design personnel into separate design teams (with a multiplicity of disciplines). A whole systems philosophy will emerge for the system because of the hierarchy.

It is important to review the direction of each team on a very regular basis and to keep control over what documents they are generating, what level they are working at, and whether or not the section they are doing fits with the rest of the design by other teams. Often a choice will need to be made on a selection of options. Good information is required about the options on offer, and rapid prototyping can provide some data about the problem. *Do not* use the rapid prototype in production designs, but use the data it generates to produce a better design.

Design Stages

Whatever the model chosen, the design stages should follow a lifecycle model for the design, implementation, and modification or decommissioning. The lifecycle model considers all stages from concept through analysis, design, implementation, verification and validation testing, commissioning operation, maintenance, and decommissioning. Some tasks will be performed throughout the whole lifecycle, although at varying levels.

These continuous tasks are hazard identification, risk analysis and evaluation, risk monitoring, and incident reporting. HAZOPs, fault trees analysis, event tree analysis, and failure modes and effects criticality analysis are all means by which the disparate information about hazards and potential hazards can be evaluated, compared, and mitigating measures evaluated for effectiveness. This not only covers hardware, but software and maintenance procedures as well.

Software Issues (and Where Forth Fits)

At present, the best figures that can be achieved for a single channel programmable electronic control system is a 10E-2 probability of failure. It does not matter how well designed the software is, or even whether it is mathematically proven.

This is a limitation of the underlying hardware. This is probably because programmable electronic control systems do not perform any internal checks of correct processor operation. Of the only attempt that the author is aware, problems were encountered with proof of correctness by anyone other than the designers of the device. That was the Viper chip.

However, poorly designed and implemented software can make the Safety Integrity Level of the system far worse than the calculated figures may suggest. It is the author's contention that Forth is one of the few programming environments which, if applied from the design phase with sufficient rigour, can produce a system that is significantly more resilient than the hardware on which it will most likely run. It does require an element of defensive programming and a high degree of rigour applied to the design, coding, and testing of the software.

Those who are Internet connected may have been following the comp.lang.forth newsgroup, and will have seen a discussion on the readability of Forth. To enable a high degree of confidence in the software we generate, we must document our code to a level that will enable certification to be supported.

An adequate level of documentation in my company is based on the following set of rules.

1. Write the glossary entry for each word that is required and conceptually prove it is complete as a description before writing the code (this takes a lot of dialogue with the client).
2. Write out the required stack effects for the word to work properly and denote limiting factors. Test the adequacy of parameters passed (by rigorous inspection).
3. Provide a description of method (attached to the front certification form) if the word is high level and requires further factoring.
4. Write code commencing with the lowest level within your application design. Ensure that this level is a good fit to the underlying machine/operating system structures that already exist.
5. Test the resultant word for compliance with its description, and expected stack effects for correct function. Ensure that all logical paths are checked thoroughly.
6. Test the resultant word with a special emphasis on limit conditions. Again, check this for all logical paths.
7. If it passes all testing, issue the word and its associated documentation for release according to work procedures. Until the word is certified, it *must* not be made available for general reference by other teams.

Forth is eminently certifiable if it is produced according to rigourously applied design rules, is complete on a word-by-word basis, and enables support of a fully documented audit trail. The author uses the form of Figure Four to record the design intent, input and output stack effects expected, the source code of the word, and the checking and test signatures. This form provides evidence required for a full safety audit trail, and is similar to the issue of Certificates of Conformity for integrated circuits.

Forth will then have the embedded Safety Critical Systems market sewn up, with fully certifiable hardware and software.

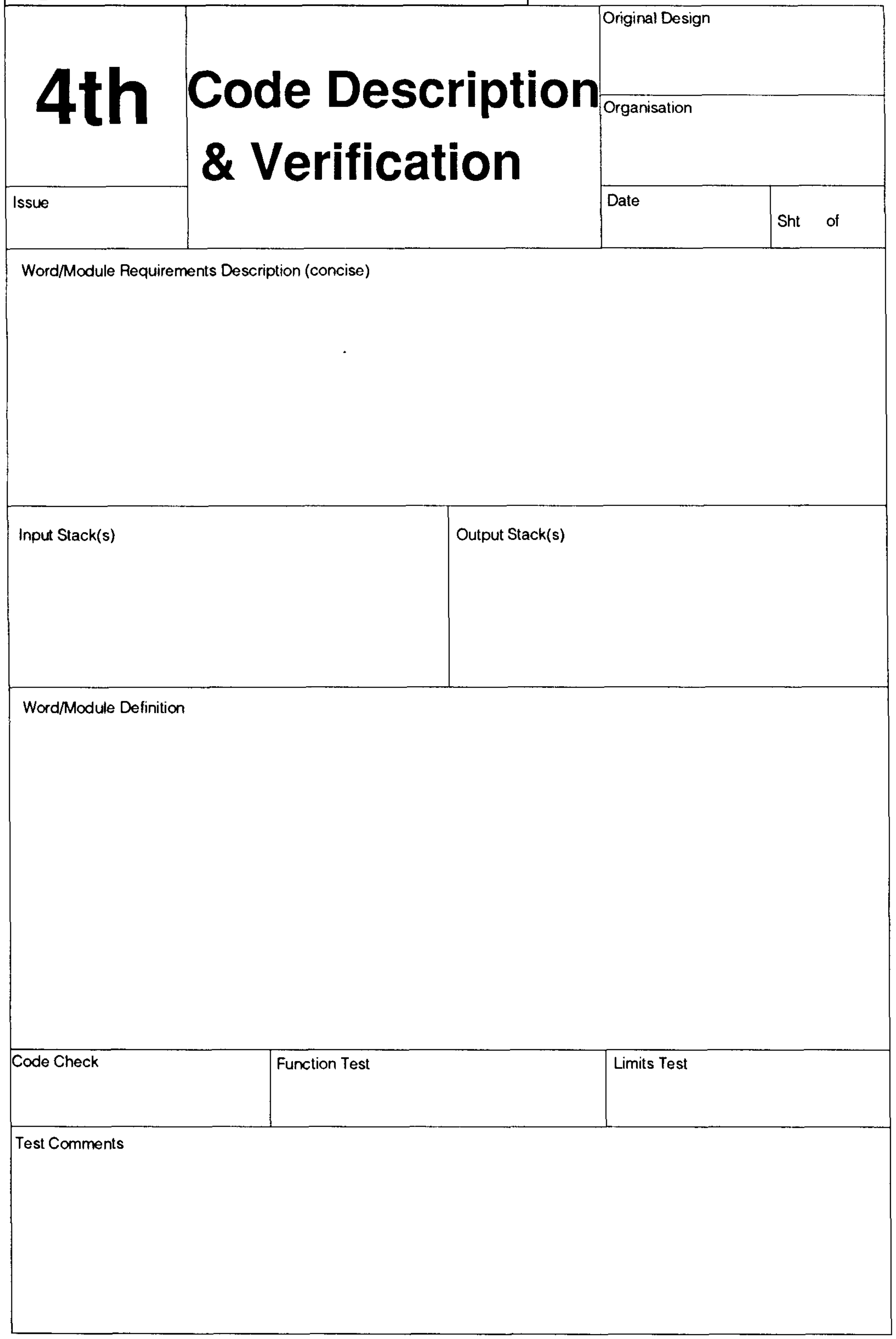
Forth and The Future of Safety Critical Systems

With a resilient and rigourously designed Forth kernel, additional wordsets, and the application code, new and more resilient hardware will be the next stage. It is the author's opinion that the stack computer is probably the best basis for such a platform. This requires the incorporation of integrity checking to the processor core. Using the stack computer would provide a sufficiently simple model on which to construct easily verified integrity checking logic. Forth will then have the embedded Safety Critical Systems market sewn up very thoroughly, with fully certifiable hardware and software.

Protect Yourself

I keep coming back to the importance of the audit trail. As designers or programmers, we have a responsibility to produce the best we can achieve, given the constraints of current technology and methods. We need to stay up to date with appropriate new methods and technology as it

Figure Four. Forth software certification form.



4th	**Code Description & Verification**	Original Design	
		Organisation	
Issue		Date	Sht of

Word/Module Requirements Description (concise)

Input Stack(s)	Output Stack(s)

Word/Module Definition

Code Check	Function Test	Limits Test

Test Comments

Paul E. Bennett (peb@transcontech.co.uk) is the Systems Engineering Director of Transport Control Technology Ltd., his own company, and has been involved in the design implementation, verification and validation, and commissioning of Safety Critical and Safety Related Systems since 1969. He has worked on Factory Automation Systems; Petroleum Production Well SCADA Systems; Nuclear Power Plant Irradiated Fuel Disposal Equipment; Specialist Robotic Cranes for Plutonium Handling; and Railway Control, Signalling, and Monitoring Systems.

Trained in electrical and electronic hardware design and construction, he picked up software through necessity of testing programmable systems that were beginning to appear in industry during the late sixties and early seventies. Paul has used Forth ever since he discovered its existence in 1982. In 1992, Paul became a member of the Safety Critical Systems Club and has been proactive in many of its events, and has also written for the Safety Systems Newsletter. He has published several papers which were given at EuroFORML and Software Quality Workshops, concentrating on Design for Safety Issues.

emerges. We need to update ourselves as new standards and legislation emerges.

We need to continuously train to improve ourselves, and we need to keep our own written log of what we did to achieve a safe system. Your log is valuable in that you have a record of useful techniques that may be re-used and you have the evidence of your correct actions if anything does go wrong and you are called to account.

Personal Risk Assessment

The risk assessment techniques mentioned in this article can also be used very successfully to assess the risks an individual engineer is exposed to. It is good practice for all engineering personnel to be acquainted with the methods.

The techniques can also be used on a business risk basis for the company as a whole. Currently, most businesses only look at the risk they expose themselves to in terms of the financial implications, but companies are also systems.

Part Two

More Than a Simple State Machine

Devin Wilson
Santa Cruz, California

In this project, we have created a clocked state machine designed to control a "traffic intersection." The traffic intersection is a circuit that Dr. Ting designed and built for the Forth Day programming contest organized by John Rible (see Figure One). It is plugged into the printer port of a computer (see Figure Two). It has four sets of four lights (red; yellow; green; and turn, also green) facing North, South, East, and West (relative to the board), and a switch for each set of lights, used to tell whether a car is coming from that direction.

We started the project last December, when John gave me the paper that said a little bit about the circuit and explained the rules. The first step was to go over the rules and make a state machine to show how the intersection would run. Although the concept of a state machine was new to me at the time, this step was relatively easy. Once we started the state machine, we found that we didn't need to treat heavy and light traffic differently, it would operate correctly either way. The end result was the state machine transition diagram shown in Figure Three. "EW" refers to cars going East and West, and "NS" refers to cars going North and South. Treating each pair of opposing directions as one simplifies the state machine amazingly and still allows for a relatively (if not completely) realistic and efficient intersection.

...this was a fun, interesting—and yes, educational—experience.

There is, however, one flaw in my state machine that remains uncorrected. When entering into the "EW Heavy" state, if there are still cars that want to turn, they will wait at the green light, causing the machine to move on to the "Turn 1" state. After ten seconds, the state will be "EW," and from there (if there is a car waiting at the North or South light) it will go immediately to "EW Caution." Notice that the cars wanting to go straight East or West only get about two seconds to go! The same thing can also happen when entering the "NS" state.

Next came the hard part—writing a Forth program that would control the circuit and run the state machine. In order to do this, I had to learn at least a little Forth. John gave me a book titled *Starting Forth*, which turned out to be interesting and understandable, and a small Forth system. After a few weeks, I had enough understanding of Forth to move on, but I found it very interesting and intend to go back and learn more in the near future.

The Forth program was done in two parts. John wrote the state machine "engine" (described in the last issue of *FD*), and together we wrote the state machine itself. I'm going to explain a bit about the state machine program, TRAFFIC.4TH (see Listing One). John wrote the Forth word `LIGHTS:` to define the words that turn the traffic lights on and off. Each bit of the binary number it uses turns one light on or off. With `LIGHTS:`, we could easily make words for only the combinations of lights we wanted. Similarly, `SWITCH:` is used to define words that sense the four switches that tell whether a car is waiting. The function of `DO-COUNTS` is a little less obvious—it is executed on each clock tick to keep track of how long each switch has been pressed. The counters are used in states "Turn 1" and "Turn 3," where we need to know whether a car wants to make a left turn. This particular part of the program took some thinking: we tried and discarded several ideas before we found one that worked. `WAIT&EXIT?` is the word used to clock the state machine. The variable `PERIOD` holds the number of milliseconds between clock ticks. The word `DONE?` does exactly what the name implies: press a key to pause, then press a different key to exit (or press the first key again to resume). Once, before `DONE?` was perfected, we went into the state machine, couldn't get back out again, and had to re-boot the computer!

The remainder of the program is the actual machine. When I first started to program the states, I named and defined the first five states, and ran headlong into a problem! The "EW," "Turn 1," and "Turn 2" states form a loop, so when defining their next-state logic equations, all of them need to use the name of the previous state. No matter what order they were put in, the first one the computer looks at will need a word that hasn't been defined yet. John solved this by changing MACHINE.4TH (the state machine engine he described in the last issue) so we could name the states without first needing to define

Figure One. Contest directions.

1995 Northern California Forth Day

Programming Challenge

adapted from Dr. Ting's "The Second Course"

A Simple Traffic Controller

The printer port has 12 output bits and 5 input bits. It is not quite enough to control the traffic lights at a busy city intersection, but about right for a rural intersection between two-lane roads. Let us use 4 input bits to sense cars entering the intersection and 8 output bits to control the traffic lights. To simplify matters, we assume that the lights towards opposing directions on the same road will be lit the same way, so that 4 bits are enough to control the green, yellow, red, and left-turn lamps in both directions.

The printer's data output port (usually 378, 278, or 3BC) drives the traffic lights:

Bit	Function	Bit	Function
0	Green, E-W proceed	4	Green, N-S proceed
1	Yellow, E-W caution	5	Yellow, N-S caution
2	Red, E-W stop	6	Red, N-S stop
3	Green, E-W left turn	7	Green, N-S left turn

The printer's status input port (usually 379, 279, or 3BD) reads the car sensor switches:

Bit	Function	Bit	Function
0	--	4	North lane sensor
1	--	5	South lane sensor
2	--	6	West lane sensor
3	--	7	East lane sensor (inverted)

A rough schematic of the board is shown in Figure Two. The N-S-E-W directions are marked on the board. The traffic controller program is to turn the lights on and off according to the following rules:

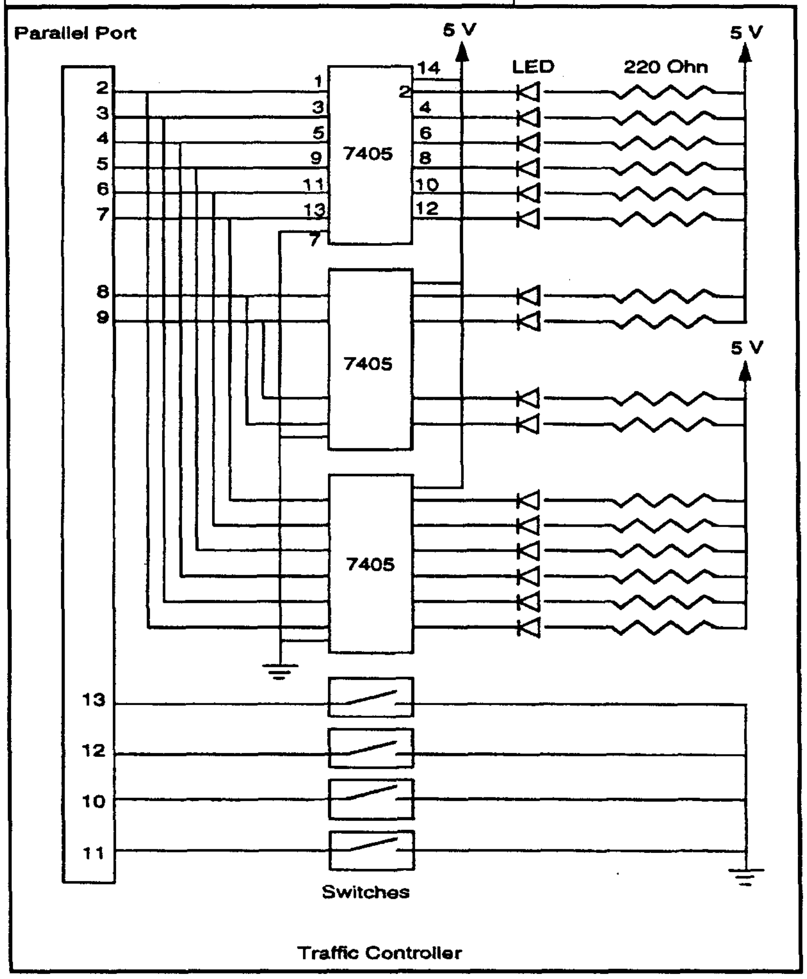
1. Turn on all stop lights upon power-up.
2. When there is no traffic, turn on E-W proceed lights and N-S stop lights.
3. When there is heavy traffic, repeat the following steps:
   a. Turn on E-W proceed and N-S stop for 10 seconds.
   b. Turn on E-W caution and N-S stop for 5 seconds.
   c. Turn on E-W stop and N-S left-turn for 5 seconds.
   d. Turn on E-W stop and N-S proceed for 10 seconds.
   e. Turn on E-W stop and N-S caution for 5 seconds.
   f. Turn on E-W left-turn and N-S stop for 5 seconds.
4. When there is light traffic, maintain the proceed light for the lane the last car drove through.
5. When a car is sensed at a stop light, do the following:
   a. Turn on caution light to the proceeding lanes for 5 seconds.
   b. Turn on stop light to the proceeding lanes and left-turn light to this lane for 5 seconds.
   c. Turn on stop light to the proceeding lanes and proceed light to this lane for 10 seconds.
6. When a car remains on a sensor for more than 2 seconds while the proceed light is on in that direction, do the following:
   a. Turn on caution light to the proceeding lanes for 5 seconds.
   b. Turn on stop light and left-turn light to the proceeding lanes for 5 seconds.

Heavy traffic means that all four sensing switches are triggering "frequently." Light traffic means that only switches in two opposing lanes are triggering and the switches in the other lanes are quiet.

This description does not describe all possible sequences. Make sure your program does not allow traffic conflicts (both directions green, for example). Other decisions not specified above should be documented (what happens when a car leaves a sensor while the light is red?).

Because of the limited number of systems, working in teams is encouraged. The decision of the judges will be based on the completeness and clarity (whatever those are) of all working programs (as defined by the judges). Bribes will *not* be a consideration! Most of all, have fun.

Figure Two. Contest circuit diagram.



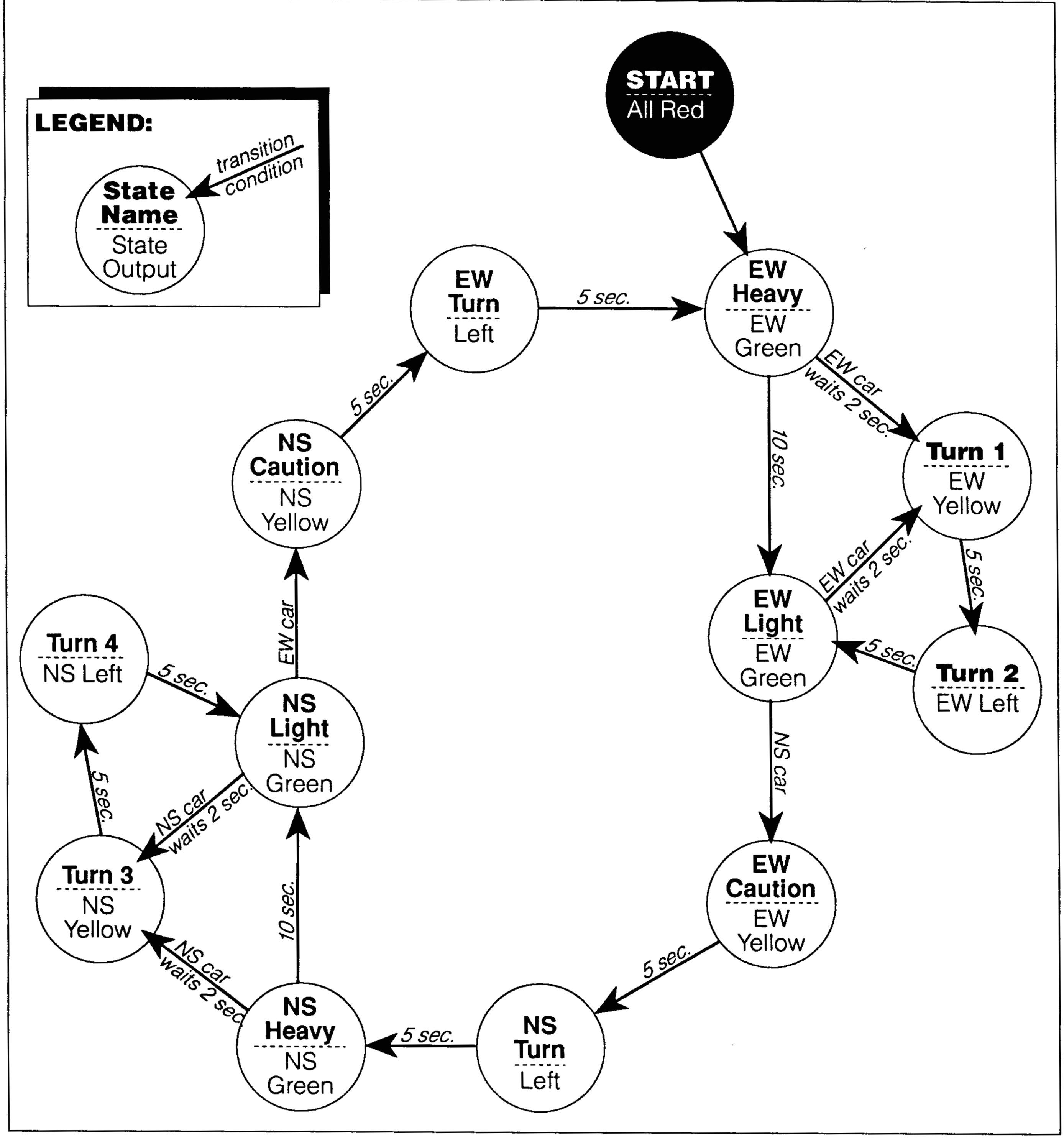
them. After he fixed that, the rest went smoothly. The first time the lights responded to a switch, we shouted. Writing the next-state logic, state outputs, and state assignments for the remaining states was definitely the most exciting part, because it was like I gave the state machine life: as I coded each part, I ran it, and watched it work!

On the whole, this was a fun, interesting (and yes, educational) experience. I will continue working in Forth and learning about the insides of a computer.

Devin Wilson is a fifteen-year-old homeschooler. He has been working with John Rible for over a year and a half, learning about the inner workings of a computer. Devin and John both can be contacted via e-mail at the jrible@quicksand.com address.

Code begins on next page...

Figure Three. The state diagram.



Listing. TRAFFIC.4TH using Dr. Ting's "traffic intersection."

```
\ This is an ANS Forth Program with Environmental Dependencies on:
\   \     from the Core extensions wordset
\   KEY?  from the Facilities wordset
\   MS    from the Facilities extension wordset
\   all the words included in the file MACHINE.4TH shown last issue
\   PC@ and PC! for byte access to IBM-PC compatible I/O space
\ A Standard System exists after this program is loaded.
\ Any operator's terminal facilities provided by the system are adequate.

\ IBM-PC compatible parallel port
HEX  378 CONSTANT PARALLEL-PORT     \ for LPT1; yours may be different
: PRN@  ( - char ) PARALLEL-PORT 1+ PC@ ;
: PRN!  ( char - ) PARALLEL-PORT    PC! ;

\ Traffic light I/O

: LIGHTS: ( pattern "name" - )
   CREATE ,
DOES> ( dfa - ) \ set specified light pattern
   @ PRN!
;

: SWITCH: ( mask "name" - )
   CREATE ,
DOES> ( dfa - on? )
   @  PRN@ 70 XOR  AND 0= 0=
;

2 BASE ! \ Traffic Light Pattern Outputs & Switch Inputs in binary

11000100 LIGHTS: NS-L      01001100 LIGHTS: EW-L
01000100 LIGHTS: NS-R      01000100 LIGHTS: EW-R
00100100 LIGHTS: NS-Y      01000010 LIGHTS: EW-Y
00010100 LIGHTS: NS-G      01000001 LIGHTS: EW-G
                           \ These are only for debugging
00010000 SWITCH: NC?        : NC.  ( - )  NC? . ;
00100000 SWITCH: SC?        : SC.  ( - )  SC? . ;
01000000 SWITCH: WC?        : WC.  ( - )  WC? . ;
10000000 SWITCH: EC?        : EC.  ( - )  EC? . ;

DECIMAL

\ Define switch counters

VARIABLE N#    VARIABLE S#    VARIABLE E#    VARIABLE W#

: DO-COUNTS  ( - ) \ increment if switch is down, otherwise set to zero
   NC? IF 1 N# +! ELSE 0 N# ! THEN
   SC? IF 1 S# +! ELSE 0 S# ! THEN
   EC? IF 1 E# +! ELSE 0 E# ! THEN
   WC? IF 1 W# +! ELSE 0 W# ! THEN
;

: CLEAR-COUNTS  ( - ) 0 N# !  0 S# !  0 E# !  0 W# ! ;

\ Define machine and state names

MACHINE: TRAFFIC-LIGHT
   STATE: START=     STATE: EW-HEAVY=   STATE: EW-LIGHT=   STATE: TURN1=
   STATE: TURN2=     STATE: EW-CAUTION= STATE: NS-TURN=    STATE: NS-HEAVY=
   STATE: NS-LIGHT=  STATE: TURN3=      STATE: TURN4=      STATE: NS-CAUTION=
   STATE: EW-TURN=
MACHINE;
```

```
\ Define next-state logic equations, all with stack ( - is-next? )

: >START      RESET= ;
: >EW-HEAVY   EW-TURN=    TICK# 10 = AND  START=    TICK#  4 = AND  OR ;
: >EW-LIGHT   EW-HEAVY=   TICK# 20 = AND  TURN2=    TICK# 10 = AND  OR ;
: >TURN1      EW-HEAVY=    EW-LIGHT= OR     E# @  W# @  MAX 4 >=  AND ;
: >TURN2      TURN1=      TICK# 10 = AND ;
: >EW-CAUTION EW-LIGHT=   NC? SC? OR AND ;
: >NS-TURN    EW-CAUTION= TICK# 10 = AND ;
: >NS-HEAVY   NS-TURN=    TICK# 10 = AND ;
: >NS-LIGHT   NS-HEAVY=   TICK# 20 = AND  TURN4=    TICK# 10 = AND  OR ;
: >TURN3      NS-HEAVY=    NS-LIGHT= OR     N# @  S# @  MAX 4 >=  AND ;
: >TURN4      TURN3=      TICK# 10 = AND ;
: >NS-CAUTION NS-LIGHT=   EC? WC? OR AND ;
: >EW-TURN    NS-CAUTION= TICK# 10 = AND ;

\ State outputs, all with stack ( - ); change counts only on entering state

: ?CLEAR-COUNTS ( - ) TICK# 0= IF CLEAR-COUNTS THEN ;

: START.      EW-R ;
: EW-HEAVY.   EW-G ?CLEAR-COUNTS ;
: EW-LIGHT.   EW-G ?CLEAR-COUNTS ;
: TURN1.      EW-Y ;
: TURN2.      EW-L ;
: EW-CAUTION. EW-Y ;
: NS-TURN.    NS-L ;
: NS-HEAVY.   NS-G ?CLEAR-COUNTS ;
: NS-LIGHT.   NS-G ?CLEAR-COUNTS ;
: TURN3.      NS-Y ;
: TURN4.      NS-L ;
: NS-CAUTION. NS-Y ;
: EW-TURN.    EW-L ;

\ Define the clock-tick word

: DONE? ( - stop? )   KEY? DUP IF DROP KEY KEY XOR THEN
; \ Tap a key to pause; tap same key again to resume, different key to stop.

VARIABLE PERIOD      500 PERIOD !      \ clocked twice per second

: WAIT&EXIT? ( #ticks - exit-machine? )
   DROP  PERIOD @ MS  DO-COUNTS  DONE?
;

\ Make machine and state assignments

\  clock-tick     <zero>       machine
   ' WAIT&EXIT?      0       ' TRAFFIC-LIGHT ASSIGN

\  to-state      output        state
   ' >START     ' START.      ' START=      ASSIGN
   ' >EW-HEAVY  ' EW-HEAVY.   ' EW-HEAVY=   ASSIGN
   ' >EW-LIGHT  ' EW-LIGHT.   ' EW-LIGHT=   ASSIGN
   ' >TURN1     ' TURN1.      ' TURN1=      ASSIGN
   ' >TURN2     ' TURN2.      ' TURN2=      ASSIGN
   ' >EW-CAUTION ' EW-CAUTION. ' EW-CAUTION= ASSIGN
   ' >NS-TURN   ' NS-TURN.    ' NS-TURN=    ASSIGN
   ' >NS-HEAVY  ' NS-HEAVY.   ' NS-HEAVY=   ASSIGN
   ' >NS-LIGHT  ' NS-LIGHT.   ' NS-LIGHT=   ASSIGN
   ' >TURN3     ' TURN3.      ' TURN3=      ASSIGN
   ' >TURN4     ' TURN4.      ' TURN4=      ASSIGN
   ' >NS-CAUTION ' NS-CAUTION. ' NS-CAUTION= ASSIGN
   ' >EW-TURN   ' EW-TURN.    ' EW-TURN=    ASSIGN

\ Now RUN the state machine
```

Forth on the Web

A CGI Shell for the Apple Macintosh

Ronald T. Kneusel
Milwaukee, Wisconsin

The World Wide Web (WWW) is a network of hypertext documents that exists as a layer on top of the Internet. Unlike static text, the text in a web document is active and when selected will move you from that document to another which may be on the same computer, or on a computer half-way across the globe. World Wide Web servers feed HTML (HyperText Mark-up Language) documents to web browsers (clients) which interpret the documents for display on the user's computer. The browser also goes the other way, interpreting the user's actions and sending the appropriate data to the server. Besides text, the WWW includes pictures, some of which are active like the text, as well as sound and movies. It should be noted that the WWW was started by physicists for exchanging technical data and has quickly evolved into the favorite time waster of technophile college students. That aside, the WWW has taken the Internet to the next level and is playing an increasingly important role in the evolution of cyberspace. "So?" you ask, "What's this got to do with Forth?" In a word, plenty. What follows is just a small example.

One of the best features of the WWW are CGI applications. A CGI (Common Gateway Interface) allows the computer to be linked into the web document. Basically, a CGI is a program that returns some sort of web document to the server which then passes it along to the client. The CGI allows for customization of the web. Most CGI applications receive their data from forms the user fills in and submits. Forms are displayed on the client's machine in a way that is familiar to users of GUI operating systems and include such standards as text fields, pop-up menus, radio buttons, and check boxes. That's the flashy side of the CGI; the interesting side is the program that accomplishes whatever it is you want done.

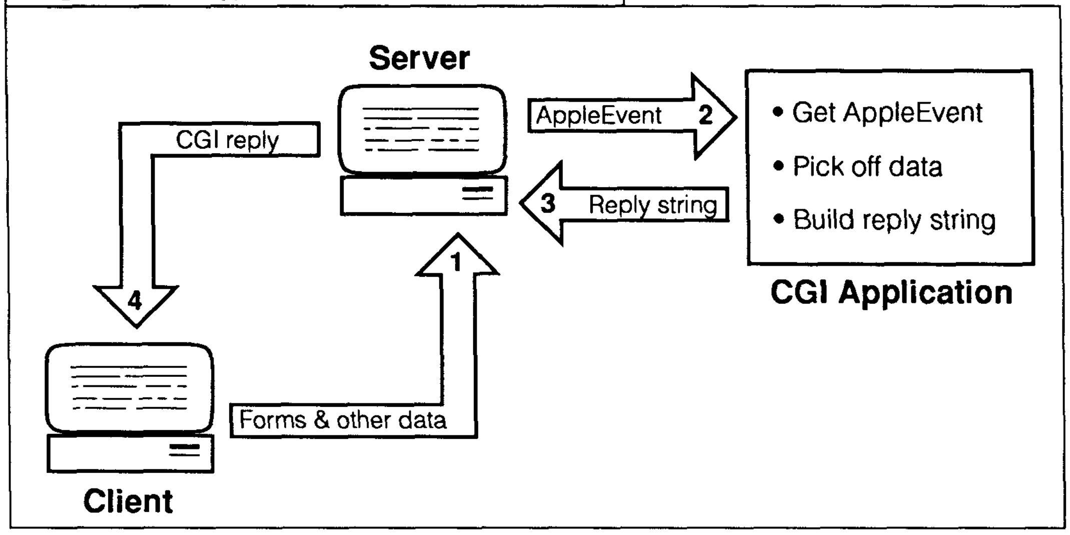
The CGI application can be written in any programming language that lets you communicate with the web server. The most common language for CGIs is the Unix shell itself. Big surprise, as the majority of web servers are running on Unix machines. Other popular languages include Perl, C, Tcl, and I've even seen Lisp and Fortran CGIs. There is a difficulty, though. A recent survey indicates that upwards of 20% of web servers in the world are running on Macintosh computers. This is interesting in its own right, as that figure is twice Apple's total market share, but makes it difficult for people running Mac-based servers. Why? Because on a Unix machine all the CGI needs to do is write its output to *stdout* and the server will get it. Macs are not so simple.

There are a handful of web servers for the Macintosh, with WebSTAR (or MacHTTP, its shareware cousin) being the most popular. WebSTAR uses Apple Events to communicate with its CGI applications. Apple Events are what Mac programs use to talk to each other. So, to write a CGI for the Mac requires a program that can receive and respond to the Apple Event sent out by WebSTAR. This is generally more difficult than the Unix method and has led to the creation of CGI "shells" which handle the ugly portions, leaving the programmer free to concentrate on more important things. Mac-based CGI shells can be found in C, Perl, Fortran, BASIC, Prograph, Frontier (a powerful scripting language), even HyperCard, but none in Forth—until now.

Ever heard of "Pocket Forth"? Probably not, though the Forth Interest Group sells a disk containing it. Pocket Forth, the freeware brain child of Chris Heilman, is a small (<17K) Forth for the Mac. It offers full control of the Macintosh Toolbox but often requires a bit of machine code programming to achieve it. It is limited to a 25K dictionary. It has no built-in file access, no integrated development tools, and has a limited name space (three characters plus length). However, besides its obvious lure as a minimalist's dream, it has two big pluses: Pocket Forth knows how to receive and respond to Apple Events and Pocket Forth knows how to use floating-point numbers. For this reason, it makes a good choice as a CGI platform and I've used it to develop a CGI shell for use with WebSTAR or MacHTTP.

The Apple Event sent by WebSTAR contains numerous fields providing the CGI with everything from the address of the client, to the type of browser they are using, to the forms data entered. The CGI shell receives the Apple Event, picks off several of the fields, and makes them available to the programmer. A typical exchange appears as in Figure One. The server delivers a document that contains a form of some kind to the client. The client fills in the form and submits it (step one). The server gathers the form data, plus some other data offered by the client, and constructs an Apple Event which activates the CGI application (step two). The application decodes the Apple Event with the shell code and uses the data to construct a reply string. This string is sent to the server (step three), which dutifully passes it down to the client (step four). The CGI shell comes into play between steps three and four above.

Figure One. Typical server-client interaction.



The CGI shell

The interesting thing about CGI programs in Pocket Forth is that they have no main word or startup word. When activated by the web server, the application opens as normal: the interpreter starts, prints the "ok" prompt, and waits for keypresses. What gets things going is the fact that while the interpreter is waiting for keypresses it is also listening for many different kinds of system events, including Apple Events. When the application receives the specific Apple Event sent out by the web server, it perks up and executes the handler associated with that event. It is precisely this handler that is defined when the CGI application is created.

One of the first tasks of any CGI application using the shell code is to evaluate the form data entered by the user. Each part of the received Apple Event consists of a string. In the case of the client's IP address or browser type, no further parsing is needed—the string can be used as is. Forms, on the other hand, do need parsing.

The data from the form is packed together as one string. Each field in the form (either a text field, radio button, or pop-up menu, etc.) is coded as the field name, an equal sign, and the value of the field. Fields are separated by ampersands. For example, if there are two fields, "one" and "two," with values of "42" and "chocolate cake," then they are passed along as:

```
one=42&two=chocolate+cake
```

Notice the "+"? Spaces are coded as "+" signs. Letters and numbers are as expected, but all other characters are coded as "%nn" where "nn" is the hexadecimal ASCII code for that character.

The CGI shell deals with this and will dutifully return the converted string for any field if given the name.

So CGI applications deal with forms data, field by field, and return a string containing the HTML code to be sent to the client. However, the real world deals with things like integers and floating-point numbers, not just strings. Does this mean the programmer is doomed to spending a lot of time converting items from numbers to strings and strings to numbers? Nope. The CGI shell introduces two useful and powerful concepts that make CGI programming significantly easier: objects and templates. No, these are not really objects in the sense of "object-oriented programming," they are more like smart record structures. Templates are just that, a template of HTML code containing tags indicating where to place the value of particular objects.

The shell uses two types of objects: generic objects and fields. A field is associated with a field in the HTML form that calls the CGI. Other than that, the two object types are identical. When the CGI starts, it automatically loads the data from the fields into the field objects. Generic objects are used when creating the reply string. They are associated with tags in the template file, as are field objects. Objects come in three flavors: integer, float, or string. The same group of access words are used with all three types. These words also handle converting values to and from strings.

An object is defined like this:

```
FP  25 " a_float"  #object 1st
INT 10 " an_int"   #object 2nd
STR 80 " a_string" #object 3rd
```

The word `#object` expects the type (`FP`, `INT`, or `STR`), size of the character representation (bytes), and a name identical to the name of the tag in the template file or the field name in the HTML form. Field objects are defined in the same way but use the word `#field` in place of `#object`. Upon definition, the name of the object is entered in an array for use when processing the template file. If the object is a field object, it is also entered in a separate array of fields to be initialized upon startup.

Within the CGI program, objects are used like other variables. The words `@val` and `!val`, analogous to `@` and `!`, are used to fetch and store values of the proper type. In the case of floats and ints, the number is returned (Pocket Forth puts floats on the stack as a five-byte beast). If the object is a string, the address of the string is returned, or the string whose address is on the stack is copied into the object. Contrary to normal Forth convention, the CGI shell uses null-terminated strings.

An Example

A complete example is in order. Following the grand tradition of simplistic first examples, we will construct a CGI to calculate the diameter, area, and circumference of a circle given its radius. Note that the code in the listings is Pocket Forth code and does not follow the ANS standard.

Many of the CGIs created with the shell consist of three files: a normal HTML file with the form where the user supplies the data (Listing One), a second file which is the CGI application (Listing Two), and a third file, the template file, with the HTML code to return plus the tags (Listing Three).

Listing One shows the HTML code for the page first brought up on the user's web browser. The form on this page has one field named `radius`. The user enters a number for the radius and clicks the Calculate button to submit the form. This causes the web server to look for the specified action (`circle.cgi`). The web server then sends the Apple Event containing the radius. At this point, the CGI application takes over.

Listing Two is the source code for the circle CGI. There is not much to it. First load the shell source code, then define the objects. After that comes the code for the particular task, which in this case is to calculate some stats for a circle of a given radius. Note the use of the words `!val` and `@val`. The Apple Event handler is the last thing defined. This is the code called when the Apple Event is received. Load the field data into the field object (`<getFields>`) and calculate the desired values (`find_stats`). Then load the template file and send the reply string back to the web server.

Listing Three is the template file used by the CGI application. It is an ordinary text file and contains, for the most part, HTML code. Of interest here are the tags set off by backquotes (`). These are the names of objects as defined in Listing Two. When the template file is read (`out fname NEW template`), these tags are replaced with the string representing the current value of that object.

Other Access Words

The CGI shell comes as three source files. All the files are merged into the shell.4th file used above. However, not every CGI needs the power of objects and templates. Therefore, it is possible to use the low-level words found in the file server.4th. This file contains all the code necessary to interact with the web server and to access parts of the Apple Event. Of particular use are the words `@Direct`, `@Addr`, and `@Browser`. These load a string with the direct argument (appended to the CGI name in the HTML form), the user's IP address, and the name of the user's web browser. Also available is the word `@Field` which loads a string with the data from a field. It does not convert strings to numbers. `@Browser` makes it possible to write CGIs that are "browser aware," so users can be sent to web pages designed to work best with their particular browser. (Useful, as not all browsers were created equal.)

Putting It All Together

The entire process, from entering data into a form to reading a template file and sending a reply, is separated by the shell into three independent sections. The first is the HTML file that initiates the CGI. The second is the program code that performs the desired function. The last is the template file which contains the reply. The program does not need to know that it is a CGI and that the values it is working with are destined for the Internet. As far as Forth is concerned, the values are normal floats, ints, or strings. Also, the HTML code is separated from the Forth code. There is no HTML code buried in the Forth application; therefore, making a simple change to the appearance of a page does not require rebuilding the CGI application. All of this works together to allow the programmer to create fast CGI applications with a minimum of time and effort.

To date, I've used the CGI shell in a number of places: as a conversion calculator for physical constants, to calculate least squares fits to linear data, as an on-line clinical calculator for physicians, and as a simple database search engine. A CGI capable of redirecting users to a randomly chosen URL is in use in Germany as part of an on-line psychology experiment in human learning. A similar CGI is used at East Carolina University by a physics professor to return randomly selected homework pages to his students. The possibilities are endless.

The example outlined above, the CGI shell source code, and other applications are available from my web site:
http://kreeft.intmed.mcw.edu
/circle.html (Circle example)
/physics.html (Conversions)
/cgishell.html (CGI shell code)
http://www.intmed.mcw.edu/clincalc.html (Clinical calculator)

Source code for the CGIs is available by sending some e-mail (rkneusel@post.its.mcw.edu) and asking for it.

Ron Kneusel works as a systems specialist for the Division of General Internal Medicine at the Medical College of Wisconsin (Milwaukee), where he also spends about 50% of his time doing data manipulation and analysis for a research group. He can be contacted by e-mail at rkneusel@post.its.mcw.edu.

Listing One. HTML code for the form.

```
<title>Circle Stats</title>
<h1>Circle Stats</h1><hr><p>
<form method=post action="circle.cgi">
Enter the radius: <input type=text name="radius"
size=10>
<p>
<input type=submit value="Calculate">
</form>
```

Listing Three. Template file.

```
<title>Circle Stats</title>
<h1>Circle Stats</h1><hr>
Previous:
<dl>
<dd> Radius = <b>`radius`</b>
<dd> Diameter = <b>`diameter`</b>
<dd> Area = <b>`area`</b>
<dd> Circumference = <b>`circ`</b>
</dl><p>
<form method=post action="circle.cgi">
Enter the radius: <input type=text name="radius"
value="`radius`" size=10>
<p>
<input type=submit value="Calculate">
</form>
```

Listing Two. Example.

```
\  Circle Stats - a simple example
\
--> shell.4th  ( load the shell source )

$[ fname circle.txt]             \ name of the template file, 'circle.txt'

1024 String>> out                \ put the reply string here

\  Define the objects.  One field object for the radius, three for the
\  values calculated.

FP 25 " radius"   #field rad     \ field object, radius entered by user
FP 25 " diameter" #object diam   \ diameter, used in by template file
FP 25 " area"     #object area   \ area
FP 25 " circ"     #object circ   \ circumference


\  Do the calculations
3.141592 fconstant pi

: diam_calc ( radius -- diameter )  2.0 f*  ;
: area_calc ( radius -- area )  fdup f*  pi f* ;
: circ_calc ( radius -- circ )  pi f*  2.0 f* ;

: find_stats ( -- )
  rad @val  diam_calc  diam !val   \ find diameter
  rad @val  area_calc  area !val   \ find area
  rad @val  circ_calc  circ !val   \ find circumference
;

\ Setup Apple Event handler
,s sdoc  ,s WWWΩ  ae:          \ AE: starts a handler for the type specified

  3 #digits !                  \ set three decimal places
  <getFields>                  \ get radius into the field object
  find_stats                   \ do the calculations

  out fname NEW template       \ put the HTML code in the template into the
                               \ string 'out' (NEW clears 'out' first).
  out REPLY                    \ send the reply back to the web server
  bye                          \ close the application
;ae                            \ end Apple Event handler
```

ADVERTISERS INDEX

The Computer Journal 8

FORML Conference 40

FORTH, Inc. 8

Forth Interest Group .... centerfold

Miller Microcomputer Services 21

Silicon Composers 2

MAKE YOUR SMALL COMPUTER
THINK BIG
(We've been doing it since 1977 for IBM PC, XT, AT, PS2, and TRS-80 models 1, 3, 4 & 4P.)

FOR THE OFFICE — Simplify and speed your work with our outstanding word processing, database handlers, and general ledger software. They are easy to use, powerful, with executive-look print-outs, reasonable site license costs and comfortable, reliable support. Ralph K. Andrist, author/historian, says: "**FORTHWRITE** lets me concentrate on my manuscript, not the computer." Stewart Johnson, Boston Mailing Co., says: "We use **DATAHANDLER-PLUS** because it's the best we've seen."

MMSFORTH System Disk	from $179.95

Modular pricing — Integrate with System Disk only what you need:

FORTHWRITE - Wordprocessor	$99.95
DATAHANDLER - Database	$59.95
DATAHANDLER-PLUS - Database	$99.95
FORTHCOM - for Communications	$49.95
GENERAL LEDGER - Accounting System	$250.00

MMSFORTH

MILLER MICROCOMPUTER SERVICES
61 Lake Shore Road, Natick, MA 01760
(508/653-6136, 9 am - 9 pm)

FOR PROGRAMMERS — Build programs FASTER and SMALLER with our "Intelligent" MMSFORTH System and applications modules, plus the famous MMSFORTH continuing support. Most modules include source code. Ferren MacIntyre, oceanographer, says: "Forth is the language that microcomputers were invented to run."

SOFTWARE MANUFACTURERS — Efficient software tools save time and money. MMSFORTH's flexibility, compactness and speed have resulted in better products in less time for a wide range of software developers including Ashton-Tate, Excalibur Technologies, Lindbergh Systems, Lockheed Missile and Space Division, and NASA-Goddard.

MMSFORTH V2.4 System Disk	from $179.95

Needs only 24K RAM compared to 100K for BASIC, C, Pascal and others. Convert your computer into a Forth virtual machine with sophisticated Forth editor and related tools. This can result in 4 to 10 times greater productivity.

Modular pricing — Integrate with System Disk only what you need:

EXPERT-2 - Expert System Development	$69.95
FORTHCOM - Flexible data transfer	$49.95
UTILITIES - Graphics, 8087 support and other facilities.	

and a little more!

THIRTY-DAY FREE OFFER — **Free MMSFORTH GAMES DISK** worth $39.95, with purchase of MMSFORTH System. CRYPTOQUOTE HELPER, OTHELLO, BREAKFORTH and others.

Call for free brochure, technical info or pricing details.

A PC Floppy Interface for non-DOS Hardware

Dwight Elvey
Santa Cruz, California

Many times you need a floppy drive on an embedded system that doesn't have an 80x86-type processor. This article is the second part in a two-part series to allow reading and writing of 360K DOS disks by non-DOS hardware. Most commercial systems have some provision for disk drives but a custom system will normally require a custom disk interface. This need for a custom interface can be avoided if one is willing to take advantage of the commercially available disk drive interface cards used in PCs. These interface cards all use the same standard for software, meaning that one is not restricted to one manufacturer. These PC cards often come with serial, printer, IDE, and floppy interfaces for less than $20. One couldn't buy the parts needed for these other functions and assemble them for anywhere near this price. This makes leveraging the mass market of the PC a powerful advantage.

In this article, I describe how I connected an XT floppy controller card to an NC4000 Forth computer. Much of the information is generic to any microprocessor one wishes to use. Many XT controllers came as only a floppy controller on one card; this was acceptable for my home project. The AT-type controller is only slightly different, and all the information here is relevant to it—I will describe the differences.

The floppy disk is divided into physical spaces that contain the data written by the controller. The first is the side, or head, used. Next is the cylinder—this is the radial distance in from the outside of the disk. The cylinder is also often called the track. Last is the sector. A sector is a piece of a track that holds a block of data. PCs use a sector size of 512 bytes. Heads are numbered 0 and 1. Tracks are numbered 0 through N. Sectors are numbered 1 through N. This makes calculating the head, track, and sector information a little tricky because one has to treat the sector number as zero-based until you need to read or write a command.

PCs use what is called a "soft-sectored" disk. This means the beginning of sectors is indicated by information written on the disk. In hard-sectored disks, sectors are marked by additional index holes. In the soft-sectored disks, the coding of sector starts is done by using data/clock sequences that are not legal for normal data but aren't so far off that the controller can't follow them.

The main types of coding for soft-sectored disks are FM, MFM, and M2FM. The controller used by the PC is only able to read FM and MFM disks. PCs use the coding scheme called MFM. MFM is able to record twice as much data as FM, but it also requires a better disk drive.

Each sector has additional information recorded besides the actual data. Various sync, check, and location information is also recorded, along with the data. The location information contains track, head, sector, and size information. This is compared with the information used in the various controller commands to locate the correct sectors. The check information is used to indicate that the data is correct. The controller used in PCs uses the CCITT 16-bit CRC for data checking. The controller only uses the CRC to verify correctness of the data. One could also use the CRC to correct a burst of bad bits in the data. CRCs like the CCITT can be used to fix corrupted data if there aren't too many bits wrong in a row. I don't recall the exact number, but it seems like up to seven bits can reliably be corrected with a 16-bit CRC like the CCITT, and can be less-reliably stretched a few bits more. CRC correction is based upon running the algorithm forwards until a number of zeros show up in the MSBs. When this happens, the LSBs are the error mask.

Before using the PC card, one has to have a basic understanding of the PC's bus. The PC uses a 62-pin connection of 31 pins on each side of the card. These connections are for the standard eight-bit cards. (The AT has an additional connector to make a 16-bit bus.) Most controller cards are still only eight bit. This makes connecting one of these cards easy, even on a 16-bit system, as in my case. One should find a good reference on the PC bus and a data sheet for the controller, as I won't go into great detail on much of the needed information. *The Indispensable PC Hardware Book* mentioned in the previous article is a great help.

Many of the available pins are not used by the floppy controller. These can usually be left open, since they have no function. Sometimes a quick look will determine if a signal has a circuit trace going to it. The floppy controller, unlike most other I/O devices, was intended to be used with DMA. If you have a processor system that requires other interrupts to always be available, you should consider using DMA to transfer sector information. This is because most floppy controllers have no buffers for data, unlike most hard disk interfaces. For the 360K disk, one has to maintain an average rate of one byte every 32 µs, and must provide the byte data within 27 µs of the request. This strict timing requires careful consideration. If you intend to do a polled system, like I have done, you'll probably need to turn the interrupts off for at least 200+ milliseconds for a sector read or write. This is because it is hard to predict what sector is under the head when a read or write operation is started. If these restrictions aren't a problem, the polled system I've described here will work for you.

Connections to your non-PC system may require a little creativity. The main ports for the floppy were intended to be at I/O address $3F0–$3F7 or, in case the card has secondary addressing, $370–$377. One may still use the board, even though one intends to use it as memory-mapped I/O or not even use the specified addresses. In my system, I wired A0–A2 to the system's A2–A4. I wired the board's A3–A8 to resistor pull-ups (i.e., using a resistor to cause a constant logical one on an input). I used A9 as a board select from an address comparator to determine the memory usage in my system. A9 has to be a select true, but otherwise can be treated just like a device select. The board's IOR (I/O Read) and IOW (I/O Write) lines were connected to my system's memory read and write strobes. Connecting D0–D7 to the system's data bus and providing +5 volts completed the system connections.

Most controller cards don't use the I/O CH RDY signal. Other signals may need to be provided for use by other parts of modern combination controller cards, such as ± 12 volts for the serial port. The floppy drives themselves require +5 and +12 volt power. Also, more addressing may need to be decoded for other functions. A PAL between the system's addresses and the card's addresses would be the simplest way to translate to a different address space.

Addresses used by the controller have the following map:

$3F0		Not used.
$3F1	R	Used by PS/2 systems for status.
$3F2	R/W	Digital Output Register (DOR) used for motor control, DMA select, reset, and drive select.
$3F3		Not used.
$3F4	R	Main Status Register. This port contains DIO and MRQ bits.
$3F5	R/W	Data Register. All commands and data are transferred through this port.
$3F6		Not used.
$3F7	R	DIR used by AT to indicate disk change status.
$3F7	W	Configuration Control Register (CCR). Used by AT to select clock reference. This controls data rates and timed functions such as step rate.

As can be seen, the AT has only extended the register set but not really altered the basic controller interface compared to the XT system. You need to be careful to remember that, when changing the CCR on an AT board, one also needs to change the various timing values selected by the Fix Drive command.

The values used by the Fix Drive command are determined by dividing the controller's clock frequency. The CCR changes the clock frequency of the controller for the different data speed requirements of different drive types. In order to maintain the same timing for things like step rates and delay times, you need to change the divider values used in the Fix Drive command when making changes to the CCR.

The CCR will have the following effects on the values in the Fix Drive command's data fields:

Speed selected	**Step rate**	**Head unload**	**Head load**
250 Kbps	2 ms/count	32 ms/count	4 ms/count
300 Kbps	1.67 ms/count	26.7 ms/count	3.3 ms/count
500 Kbps	1 ms/count	16 ms/count	2 ms/count
1 Mbps	0.5 ms/count	8 ms/count	1 ms/count

The fields for the Fix Drive command used in the `FSPECIFY` *word are:*

```
First parameter byte:  ssssuuuu
Second parameter byte: lllllld
    ssss = Step count (2's complement)
    uuuu = Head unload count
    lllllll = Head load count
    d = Not DMA
```

Values used in CCR:

bit 1	**bit 0**	
0	0	500 Kbps
0	1	300 Kbps
1	0	250 Kbps
1	1	1 Mbps

BPS speeds for CCR:

Disk	**Speed**
360K	250 Kbps
360K	300 Kbps (when in 1.2 Mb drive)
1.2 Mb	500 Kbps
720K	250 Kbps
1.44 Mb	500 Kbps
2.88 Mb	1 Mbps (may not work on all controllers/drives)

One can use a standard PC cable to connect the drive(s) to the controller board. Look at the ribbon cable used and, if the cable doesn't have some of the signal lines twisted between the first and second drive, you'll have to set the jumpers on the drive for different selects.

I used a small mini-tower PC-type power supply for my system. This came with the Molex connectors needed for the floppy drive's power. Many supplies that have both +5 and +12 can be used, but one must make sure that the 12 volts has enough current. Most floppies require about one ampere of 12 volts to run, and up to about three amperes of surge to start.

The floppy controller does much of the work of reading and writing the floppy for you, but it still helps to understand its interface to the bus. All data transfers must happen when the controller is ready. The controller synchronizes all transfers with two bits in the Main Status Register. One bit is MRQ (Master ReQuest). Data can only be transferred though the data port when this bit is true. The other bit is the DIO (Data Input/Output). This bit determines whether the controller wishes the system to read (bit true) or write (bit false).

The controller recognizes commands transferred to it as a sequence of bytes. Most major operations require eight bytes to be written in sequence, but some commands require less. Of course, sector writes will also require sector data. Many commands have an additional result phase that requires reading various status bytes. Almost all commands have a fixed number of bytes transferred—except the check-interrupt-status command. That command will return one or two bytes, depending on the values of the status in the first byte. Even though I'm not using interrupts in my setup, it is still necessary to use this command to determine when a recalibration or seek operation has been completed. Looking at the bytes used in the command, it seems that some information is redundant: the head selected is required twice. This is because the first bit is used to tell the drive which head to use and the second is used for comparison of the data read from the disk.

When reading a 360K disk on a 1.2 Mb drive, the tracks are spaced two steps apart. The Seek command is given the number of the cylinder that is twice the cylinder desired. When a command that will read or write the disk is given, the cylinder number must match the one recorded on the disk. It is not recommended that one use a 1.2 Mb drive to write to a 360K disk, as the width of the data written will not completely cover the data written with a 360K drive. If one only uses a 1.2 Mb drive to read/write the disk, this isn't a problem.

Here is a list of the commands the floppy controller recognizes:
Read Sector
Read Deleted Sector
Write Sector
Write Deleted Sector
Read Track
Format Track
Read Identification
Calibrate Drive
Check Interrupt Status
Fix Drive Data
Check Drive Status
Seek
Verify
Determine Controller Version
Seek Relative

Of these commands, I use only Read Sector, Write Sector, Calibrate Drive, Check Interrupt Status, Fix Drive Data, and Seek. These are the minimum needed to do floppy transfers. Since I'll be using the controller to read and write DOS disks, I'll leave disk formatting to my PC.

The Fix Drive Data command is done by the `FSPECIFY` word in my code. Part of this word is the step rate. Determining the best step rate is quite easy. You need to experiment with rates: as one steps faster, the drive will seem to quiet down and not make the clicking sound as it steps. This is usually the best step rate for your drive. One can then experiment with seeks and data reads to make sure the step rate truly works correctly. The maximum step rate for a drive changes very little over the drive's lifetime, and is dominated by the stepper motor's ability to move the fixed mass of the armature and head assembly.

Main operation in code:

1. `FRECAL`
   Power-up initializations or after error. This word will need the speed-crystal-select added if used with an AT controller.

2. `FSEEK`
   Locate track.

3. `FSEC@`
   Read a sector.

4. `FSEC!`
   Write a sector.

5. `SELDRV`
   Selects one of two floppy drives.

So, in conclusion, one can take advantage of the large volume of PC sales to provide a resource of I/O interface systems at a cost that couldn't be duplicated in a small embedded system. Using one of the combination cards would be a sound design decision to add resources to such a system.

Code Commentary

All standard PC floppy interfaces use the uPD765 as the base type controller. All of the standard uPD765 commands are usually supported. For this reason, one should get a spec sheet for this or a compatible controller (Intel's 8272 is one). Some of the newer chips, such as the 82077A, include a 16-byte FIFO and are more lax on timing requirements. Most standard controllers only have one level of buffering and, therefore, require that the data be handled quickly. For a 360K floppy that transfers one byte every 32 µs, you need to respond with each byte no later than 27 µs after the controller makes its request. This is why most people use the DMA feature of the controller. I am using a µP that doesn't have any interrupts and can be dedicated to controlling the floppy while doing transfers. Since the NC4000 is running at 4 MIPS, transferring data with this time constraint isn't a problem.

cmForth is different from many standard Forths, so one might want to look over the following "funny" NC4000 words:

`@+`	`( Addr Incr - Value Addr+Incr )` Machine-coded fetch and increment.
`!+`	`( Value Addr Incr - Addr+Incr )` Machine-coded store and increment.
`FOR`	`( n - )` The `FOR NEXT` loop is like `DO LOOP` except the loop counter `NEXT` is a down counter. The loop will execute n+1 times. When the value on the return stack = 0, `NEXT` will cause the loop to stop. You'll see places in my code where I do `R> DROP 0 >R` which is the same as `LEAVE` in some Forths.
`BSWAP`	`( HL - LH )` Not actually a cmForth word. I added special hardware to speed up byte swapping in 16-bit values.
`I`	`( - Value )` `I` in cmForth is like `R@`. It simply copies the top of the return stack. The return stack is also used for the `FOR NEXT` counter, so it makes more sense to call it `I`.

`DR0`	`( - )`
	Selects the offset for drive 0.
	DR0 = 360K Floppy A
	DR1 = 360K Floppy B
	DR2 = 5 Mb HD

`TIMES`	`( n - | Word )` Executes the following word n+1 or n+2 times. This is a confusing one because it works differently if the word is a code word than if the word is a nested word. Code words are executed n+2 times.
`2/MOD`	`( n - r q )`
`DOES`	`( - )` Similar to `DOES>` in regular Forth but doesn't return the address of the parameter field. To get the parameter field, it must be popped with `R>`. Since the carry bit may have been pushed into the high bit, it also needs `7FFF AND`. Essentially: `DOES>` equals `DOES R> 7FFF AND`

Normal Floppy PC addresses:
Write DOR Digital output register 3F2
D7 = MOTD
D6 = MOTC
D5 = MOTB
D4 = MOTA
D3 = DMAEN
D2 = REST\
D1 = DR1
D0 = DR0

Read Status register 3F4
D7 = MRQ Main request
D6 = DIO Data in/out
D5 = NDMA non-DMA
D4 = BUSY
D3 = ACTD
D2 = ACTC
D1 = ACTB
D0 = ACTA

Read/Write Data register: 3F5

Read Digital Input register (AT only): 3F7
D7 = Change Disk

Write Configuration control register (AT only): 3F7
D2 = RAT1
D1 = RAT0
D0 = HDRAT

RAT1,RAT0	00=500K 01=300K 10=250K 11=1000K
HDRAT	0 = 1000K OR 500K
	1 = 250K OR 300K

250K	360 5 1/4	360 DRIVE
	720 3 1/2	
300K	360 5 1/4	1.2M DRIVE
500K	1.2M 5 1/4	
	1.4M 3 1/2	

I have re-mapped the addresses for the controller. One should consider that the addresses used by the PC don't have to be the same as the addresses used in your system. One can simply put A9 through A4 to the correct level and use A3 of the controller as CS* or any other strange addressing. For this reason, I have used memory-mapped I/O (instead of Port I/O mapped) and changed the addressing. In my system, I've used DC00 as the base address, with controller A2 wire-mapped to A4, A1 wire-mapped to A3, and A0 wire-mapped to A2. This allowed more flexibility for my system, since hardware decoding was already there.

The bus in my system was more heavily loaded on D0–D7, so I have the controller's bytes D0–D7 mapped to my bus at D8–D15. Normally, one would map D0–D7 to D0–D7; this would be the fastest and simplest method. Accompanying this article is minimum code to talk to a 360K drive with an XT floppy controller card. I have not included any error control; for reliability, this should be added. I don't do any disk formatting, because I use my PC to do the original formatting. This allows this basic code to be used to read and write DOS sectors.

Code begins on next page.

Downloading: ELVEY1.fth and ELVEY2.fth are at ftp://ftp.Forth.org/pub/Forth/FD/1996

Dwight Elvey is a long-time member of FIG whose claim to fame is that he was the first to report getting an 8080 fig-Forth listing to work (which he purchased at the West Coast Computer Faire). He was also the winner of the *Forth Dimensions* Sort Contest. Dwight works as a test engineer for Hal Computer Systems and, over the years, has used Forth for many embedded systems and test setups. He can be e-mailed at elvey@hal.com. Current side interests are in digital signal processing, model slope gliders, and sailing.

Listing. ELVEY2.fth (ELVEY1.fth appeared in the preceding issue.)

```
HEX
( Floppy controller is memory mapped )

DC14 CONSTANT FDATA   \ Floppy data port

DC08 CONSTANT FREG    \ Floppy DOR
 0000 CONSTANT FSEL0  \ Floppy 0 select mask
 0100 CONSTANT FSEL1  \ Floppy 1 select mask
 0400 CONSTANT FRST-  \ Controller reset not
 1000 CONSTANT FMOT0  \ Motor 0 enable
 2000 CONSTANT FMOT1  \ Motor 1 enable
 F000 CONSTANT FMOT   \ Motor mask

DC10 CONSTANT FSTAT    \ Floppy status port
 4000 CONSTANT FDIR    \ T=RD F=WR
 8000 CONSTANT FREQ    \ Floppy REQUEST bit
 1000 CONSTANT FBUSY   \ Floppy busy bit
 C000 CONSTANT BUSYMSK \ busy and request
 FDIR FREQ OR CONSTANT FREQRD \ Request read

: BSWAP ( N - N' )  \ special hardware to swap bytes.
    -1 0 !+ @ ;     \ This only cost 4 cycles on my machine.
                    \ I but the byte swap at address $FFFF for simplicity.

: WFRQ ( - FStat ) \ Wait for FRequest.
   BEGIN
     FSTAT @           \ Check Status
     DUP FREQ AND 0=   \  for request.
   WHILE
     DROP
   REPEAT ;

: FSTAT& ( Mask - Result ) \ Masking for status
   FSTAT @ AND ;

: FRDAT ( - DATA )  \ Read Floppy Data port.
   WFRQ             \ Wait for request.
   FDIR AND         \ Should be a read request.
   IF
     FDATA @        \ Read floppy data
     BSWAP 0FF AND  \ Make byte
   ELSE
     ABORT" RD? "   \ Was expecting to read?
   THEN ;

: FWDAT ( Data - )
   BSWAP          \ Move byte to D8-D15.
   WFRQ           \ Wait for request.
   FDIR AND       \ Check for write?
   IF
     ABORT" WR? "  \ Was expecting to write?
   ELSE
     FDATA !       \ Write floppy data.
   THEN ;

DECIMAL

: FSPECIFY ( - ) \ Specify command
\ These values are different for AT controllers at different speeds.
\ I am not using DMA in my simple system -- I am using polled
\  I/O instead. This means that I have to dedicate the NC4000
\  to disk-only during transfers. If one wanted to, one could
\  use a DMA controller and use DMA. For 360K drives, one needs
\  to be able to supply a byte every 32 µs, and 16 µs for a 1.2 Mb.
\  This is no problem for a 4 MHz NC4000, but one should analyze
\  time used to do reads and writes.

  3 FWDAT                   \ Command Byte Specify
  [ 24 ( STEP) 2 / NEGATE 16 *     192 ( HUT) 32 / + ]
    LITERAL FWDAT           \ 24 ms step and 192 ms head unload
  [ 40 ( HLT) 4 / 2 *       1 ( NonDMA) OR ]
    LITERAL FWDAT ;         \ 40 ms head load time with no DMA

\ Most commands are built with 9 bytes
                     \ Command 1st byte.
VARIABLE FSEL        \ Selects, 2nd byte of a command.
VARIABLE FCYL        \ Cylinder, 3rd byte of a command.
VARIABLE FHD         \ Head, 4th byte of a command.
VARIABLE FSEC        \ Sector, 5th byte of a command.
                     \ Sector size, 6th byte of a command.
                     \ Max Sectors, 7th byte of a command.
1B CONSTANT GPL      \ Gap Length, 8th byte of a command.
                     \ Data Length, Always 0FF for non-zero Sector size above.

\ Result returns 7 bytes, first 3 are status.
VARIABLE FST0
VARIABLE FST1
VARIABLE FST2
\  last 4 are cyl,head,sect#,sect size.

DECIMAL

: CTSH ( DskAddr - ) \ Break a logical address into Cyl,HD,SEC.
  18 /MOD            \ 18 sectors per track.
  DUP 39 >           \ 40 track per disk.
  IF ABORT" <TRK>" THEN  \ too many tracks?
  FCYL !             \ track = cylinder.
  9 /MOD FLAG FHD ! \ determine head.
  1 + FSEC ! ;       \ Sector numbers start at 1, not zero.

HEX

: WNBSY ( - )
    BEGIN
     1F00 FSTAT& 0=   \ Wait for no activity of disk.
    UNTIL ;

: FWHD/SEL ( - ) \ Writes standard second command byte.
   FHD @ 4 AND       \ Head bit.
   FSEL @ OR FWDAT ; \ With drive bits.

: FCMD ( Command - ) \ Write a standard 9-byte command sequence.
   WNBSY         \ Wait not busy
   40 OR FWDAT   \ Mask in MFM bit to command.
   FWHD/SEL      \ Head drive select.
   FCYL @ FWDAT  \ Cylinder number.
   FHD @ 1 AND FWDAT  \ Head number.
   FSEC @ DUP FWDAT   \ Sector #.
   2 FWDAT            \ 2 = 512 Sector size.
```

(Code continues...)

```
   FWDAT               \ sector # or sectors per track.
   GPL FWDAT           \ Gap length.
   -1 FWDAT ;          \ Always OFF.

: FRSLT ( - )  \ Read Result Phase data.
     FRDAT FST0 !    \
     FRDAT FST1 !    \
     FRDAT FST2 !     \ Read 3 status bytes.
     3 FOR
      FRDAT DROP    \ discard other 4 bytes.
     NEXT
     100 CYCLES ;   \ a little delay for about 25 µs.

: FCLR ( - )     \ Clear any activity.
    BEGIN
      8 FWDAT    \ Status command.
      FRDAT 80 - \ No other status.
    WHILE
      FRDAT DROP \ discard cylinder.
    REPEAT ;

: WTEND ( - )    \ Wait for end of operation.
  0
  BEGIN
   DROP
   8 FWDAT         \ Status.
   FRDAT DUP 80 -  \ no activity.
  UNTIL
  FST0 !           \ save status.
  FRDAT FCYL ! ;   \ and current cylinder.

: SELDRV ( DRV# - )
   DUP
   IF
    FSEL1 FMOT1  \ controller mask and motor for drive1.
   ELSE
    FSEL0 FMOT0  \ controller mask and motor for drive2.
   THEN
   OR FRST- OR  \ clear reset and start motor.
   FREG !
   FSEL ! ;     \ and drive select for controller.

: ?FSEL ( DRV# -)      \ Make sure drive is selected and running
   FSEL @ OVER -        \ check selected
   IF
     SELDRV             \ if not then select drive.
   ELSE
     DROP               \ else do nothing.
   THEN ;

: FRECAL ( Drv# -) \ Recal drive.
   DUP SELDRV      \ Start drive.
   FSPECIFY        \ Load step, hlt and hut
   FCLR            \ force a clear of computers
   7 FWDAT         \ recal command
   FWDAT           \ write drive number
   WTEND ;         \ wait for the end

: FOFF ( - )      \ motor off
\ This should be called when it is desired to turn motor off.
\  This may be connected with some timer interrupt. I use
\  the KEY wait loop with a timer.
   FRST- FREG !    \ turn motors off
   -1 FSEL ! ;     \ invalid drive for ?FSEL

: FSEEK ( -)       \ Seek cylinder.
   WNBSY           \ wait not busy.
   FCLR            \ clear controller.
   0F FWDAT        \ seek cylinder command.
   FWHD/SEL        \ second command byte
   FCYL @ FWDAT    \ cylinder.
   WTEND ;         \ wait until done.

\ Addressing is 16-bit words
: FSEC! ( Addr -) \ Write a sector from address.
   5 FCMD         \ Write sector.
   1FF FOR
     1 @+         \ fetch 512 bytes.
     SWAP FWDAT   \ write to floppy.
   NEXT
   DROP
   FRSLT ;       \ read result.

: FSEC@ ( Addr - ) \ Read a sector to address.
    6 FCMD    \ Read a sector command.
    1FF FOR
     FRDAT     \ get 512 bytes.
     SWAP 1 !+ \ put them into buffer.
    NEXT
    DROP
    FRSLT ;   \ read result.

DECIMAL

: WRBUF ( Addr Blk# - ) \ Write a 1K buffer to disk block number.
   360 /MOD ?FSEL \ start drive.
   2*             \ disk physical address( 512/SEC ).
   DUP CTSH       \ calc cylinder, sector and head.
   FSEEK          \ move there.
   OVER FSEC!     \ write first half.
   1 +            \ next logical sector.
   CTSH FSEEK     \ move there.
   512 + FSEC! ;  \ write next bytes.

: RDBUF ( Addr Blk# - ) \ Read a 1K buffer from disk block number.
   360 /MOD ?FSEL \ start drive.
   2*             \ disk physical address( 512/SEC ).
   DUP CTSH       \ calc cylinder, sector and head.
   FSEEK          \ move there.
   OVER FSEC@     \ Read first sector.
   1 +            \ next logical sector.
   CTSH FSEEK     \ move there.
   DUP 512 + FSEC@ \ Read second sector.
   1023 FOR    \ Mask $4000 into each byte for cmForth.
    DUP @
    16384 OR
    SWAP 1 !+
   NEXT DROP ;
```

hForth: A Small, Portable ANS Forth

Wonyong Koh, Ph.D.
Taejon, Korea

Background History

I started a personal project two and a half years ago which has been in my mind for quite a long time: Widespread Forth in Korea. Postfix is natural to Korean people since a verb comes after an object in the Korean language. Also, Forth does not restrict a programmer to only alphanumeric characters; a Korean Forth programmer can easily express his idea in comfortable Korean words rather than be forced to think in English. As one might expect, there was an effort for Korean Forth. Dr. Chong-Hong Pyun and Mr. Jin-Mook Park built a Korean version of fig-Forth for the Apple II computer in the mid-eighties. Long-time *FD* readers may remember Dr. Pyun's letter in *FD* X/6. Unfortunately, the Korean computer community swiftly moved to the IBM PC while Dr. Pyun wrote articles about their work in popular programming and science magazines. It became somewhat obsolete before being known widely. Despite this and other efforts, Forth has been virtually unknown to most Koreans. Two and a half years ago, I decided to restart the effort and looked for a vehicle for this purpose. I found that there was no small ANS Forth system for the IBM PC. I decided to build one. In the course of ANSifying eForth, I have replaced every line of eForth source and felt that it deserved its own name. I knew there were Forth systems named bForth, cForth, eForth, gForth, iForth, Jforth, and KForth. I picked "h" since it apparently was not yet used by anyone; also, Han means Korean in the Korean language.

ROM Model Came First

eForth, which was written by Mr. Bill Muench and Dr. C.H. Ting in 1990, seemed to be a good place to start. I studied eForth source and Dr. Ting's article in *FD* XIII/1 and set the following goals:

- small machine-dependent kernel and portable high-level code
- strict compliance to ANS Forth
- extensive error handling through `CATCH`/`THROW`
- separated code and name space
- use of wordlists
- explicit consideration for separated RAM/ROM address space
- simple vectored input/output
- direct-threaded code
- easy upgrade path to optimize for specific CPUs

Most of these are adapted from eForth. I emphasize extensive error handling, since some well-known Forth systems cannot manage as simple a situation as divide-by-zero. In hForth almost all ambiguous conditions specified in the ANS Forth document issue `THROW` and are captured by `CATCH`, either by a user-defined word or by the hForth system.

The hForth ROM model is especially designed as a minimal development system for embedded applications which use non-volatile RAM or ROM emulators in place of ROM. The content of the ROM address space can be changed during development and is copied later to real ROM for the production system. The hForth ROM model checks whether or not the ROM address space is alterable when it starts. New definitions go into the ROM address space if it is alterable. Otherwise, they go into the RAM address space.

Alterable ROM address space	**Unalterable ROM address space**
	name space of new definitions
RAM address space	RAM address space
data space	data space / code space of new definitions
name space of old definitions	name space of old definitions
name space of new definitions	
ROM address space	ROM address space
data space / code space of new definitions	data space
code space of old definitions	code space of old definitions

Data space can be allocated either in ROM address space for tables of constants or in RAM address space for arrays of variables. `ROM` and `RAM`, recommended in the Appendix of the ANS Forth document, are used to switch between the RAM and ROM address spaces. Name space may be excluded in the final system if an application does not require the Forth text interpreter. The 8086 hForth ROM model occupies little more than 6 Kb of code space for all of the Core word set and requires at least 1 Kb of RAM address space for stacks and system variables.

The assembly source is arranged so that more implementation-dependent words come earlier. System-dependent words come first, CPU-dependent words come after, then come all the other high-level words. Colon definitions of all high-level words are given as comments in the assembly source. One needs to redefine only the system-dependent words to port the hForth ROM model to an 8086 single-board computer from the current one for MS-DOS machines, without changing any CPU-dependent words. Standard words come after essential non-Standard words in each system-dependent, CPU-dependent, and portable part. All Standard Core word set words are included to make hForth an ANS Forth-compliant system. High-level Standard words in the last part of the assembly source are not used for the implementation of hForth and can be omitted to make a minimal system. The current 8086 hForth ROM model for MS-DOS has 59 kernel words: 13 system-dependent words, 21 CPU-dependent non-Standard words, and 25 CPU-dependent Standard words. System-dependent words include input/output words and other words for file input through keyboard redirection of MS-DOS. For five of the kernel words, including `(search-wordlist)` and `ALIGNED`, CPU-dependent definitions are used instead of high-level definitions for faster execution.

System initialization and input/output operations are performed through the following execution vectors: `'boot`, `'init-i/o`, `'ekey?` , `'ekey`, `'emit?` , `'emit`, and `'prompt`. Appropriate actions can be taken by redirecting these execution vectors. `'init-i/o` is executed in `THROW` and when the system starts, while `'boot` is executed only once when the system starts. One has a better chance not to lose control by restoring I/O vectors through `'init-i/o` whenever an exception condition occurs. For example, serial communication links may not be broken by an accidental change of communication parameters. `'boot` may be redirected to an appropriate application word instead of the default word in a finished application. The traditional "`ok<end-of-line>`" prompt (which is actually not) may be replaced by redirecting `'prompt`.

Control-structure matching is rigorously checked for different control-flow stack items. Four 4-bit fields are used to check balancing of `orig`, `dest`, `do-sys`, and `of-sys`. A 2-bit field overlapped on a 4-bit field for `of-sys` is used for `case-sys`. Each field increases when an item is put on the control flow stack and decreases when it is consumed. All of these fields should be zero before the definition is added to the current wordlist. Otherwise the hForth compiler issues:

```
-22 THROW (control structure mismatch)
```

The number of words grows substantially as a Forth system is extended. Dictionary searches can be time consuming unless hashing or other means are employed. Currently hForth uses no special search mechanism, however, it maintains a reasonable compilation speed by keeping a shallow search depth in addition to using an optimized `(search-wordlist)`. Initially two wordlists are in the search-order stack: `FORTH-WORDLIST` and `NONSTANDARD-WORDLIST`. `FORTH-WORDLIST` contains all the Standard words and `NONSTANDARD-WORDLIST` contains all the other words. Upon extending hForth, optional Standard words will go in `FORTH-WORDLIST` and lower-level non-Standard words to implement them will be kept in separate wordlists which are usually not in the search-order stack. Only a small number of non-Standard words to be used by a user will be added in `NONSTANDARD-WORDLIST`.

RAM and EXE Models Came Later

The hForth package consists of three models: ROM, RAM, and EXE. The hForth RAM model is for RAM-only systems where name, code, and data spaces are combined. The hForth EXE model is for a system in which code space is completely separate from data space, and an execution token (xt) may not be a valid address in data space. The 8086 hForth EXE model uses two 64 Kb full-memory segments: one for code space and the other for name and data spaces. The EXE model might be extended for an embedded system where name space resides in the host computer and code and data space are in the target computer. A few kernel words are added to the ROM model to derive RAM and EXE models, and only several high-level words such as `HERE` and `CREATE` are redefined.

The ROM and RAM models are probably too slow for many practical applications. However, the 8086 hForth EXE model is more competitive. The high-level colon definitions of all frequently used words are replaced with 8086 assembly code definitions in the hForth EXE model. Comparison with other 8086 Forth systems can be found in Mr. Borasky's article "Forth in the HP100LX" (*FD* XVII/4).

The hForth models are highly extensible. Optional word sets, as well as an assembler, can be added on top of the basic hForth system. Complete Tools, Search Order, Search Order Ext word set words and other optional Standard words are defined in *OPTIONAL.F* included in the 8086 hForth package. An 8086 Forth assembler is provided in *ASM8086.F.* Much of the Core Ext word set is provided in *OPTIONAL.F,* and all the other Core Ext words except obsolescent ones and `[COMPILE]` (for which `POSTPONE` should be used) are provided in *COREEXT.F.* The complete Double and Double Ext word sets are provided in *DOUBLE.F.* The high-level definitions in these files should work in hForth for other CPUs. These files are loaded into 8086 hForth for MS-DOS machines through the keyboard-redirection function of MS-DOS. Complete Block, Block Ext, File, and File Ext word set words are provided in *MSDOS.F* using the MS-DOS file-handle functions. Other utilities are also included in 8086 hForth package. *LOG.F* is to capture screen output to an MS-DOS text file which is edited to make Forth text source. *DOSEXEC.F* is to call MS-DOS executables from within the hForth system. A user can call a familiar text editor, edit Forth text source, exit the editor, load the source, and debug without leaving hForth. This process can be repeated without saturating address spaces if a MARKER word is defined in the beginning of the Forth text source and called before reloading the source.

Multitasker

I had a chance to look at Mr. Muench's eForth 2.4.2 which has not yet been released by him. The multitasker is the most elegant one among those I have seen. It does task switching through two high-level words. I immediately adapted it to hForth. (Mr. Muench's multitasker is now included in P21Forth for the MuP21 processor.)

In the Forth multitasker, each task has its own context: parameter stack, return stack, and its own variables (traditionally called user variables). The contexts must be stored and restored properly when tasks are suspended and resumed. In Mr. Muench's multitasker, PAUSE saves the current task's context and wake restores the next task's context. PAUSE saves the return stack pointer on the parameter stack and the parameter stack pointer into a user variable stackTop, then jumps to the next task's status, which is held in the current task's user variable follower. It is defined as:

```
: PAUSE
  rp@ sp@ stackTop !
  follower @ >R ; COMPILE-ONLY
```

Advanced Forth users already know that >R EXIT causes a high-level jump for the traditional Forth virtual machine. Each task's user variable status holds wake and is immediately followed by the user variable follower. Initially, hForth has only one task, SystemTask. Its user variables status and follower hold:

SystemTask's	status	follower
	wake	absolute address of SystemTask's status

If FooTask is added, status and follower of the two tasks hold:

SystemTask's	status	follower
	wake	absolute address of FooTask's status

FooTask's	status	follower
	wake	absolute address of SystemTask's status

Effectively, the current task's PAUSE jumps to the next task's wake. At this point, user variables and stacks are not switched yet. wake assigns the return stack item (the next address of status, i.e., the address of follower) into the global variable userP which is used to calculate the absolute address of user variables. All user variables cluster in front of follower. Now user variables are switched. Then wake restores the parameter stack pointer stored in the user variable stackTop (now the parameter stack is switched) and restores the return stack pointer saved on top of the parameter stack (now the return stack is switched). wake is defined as:

```
: wake
  R> userP !
  stackTop @ sp!  rp!
  ; COMPILE-ONLY
```

What is clever here is that one item on return stack, left by PAUSE and consumed by wake, is used to transfer control as well as information for context switching. This multitasker is highly portable. Not a line of multitasker code was touched when the hForth 8086 RAM model was moved to the Z80 processor. I believe that it should be possible to port this multitasker to subroutine-threaded or native-code Forth by redefining them in machine codes.

I used this multitasker to update a graphics screen and to make the cursor blink in *HIOMULTI.F.* Console output is redirected to the graphics screen to display Korean and English characters for VGA and Hercules Graphics Adapters. EMIT fills characters into a buffer and a background task displays them on the graphics screen when hForth is waiting for user input. Scrolling text on a graphics screen is as fast as on a text screen. I also used the multitasker for serial communication in *SIO.F.* The main routine fetches characters from the input buffer and stores characters in the output buffer while the background task does the actual hardware control.

Jump Table Interpreter

I applied all the best ideas and tricks I know to hForth. Most of them came from other people, while I added a few of my own. I believe that some of them are worth mentioning.

The hForth text interpreter uses a vector table to determine what to do with a parsed string after searching for it in the Forth dictionary. The dictionary search results in the string and 0 (for an unknown word), xt and -1 (for non-immediate words), or xt and 1 (for immediate words) on the parameter stack. The hForth text interpreter chooses the next action with the following code:

```
1+ 2* STATE @
1+ + CELLS 'doWord + @
EXECUTE
```

The 'doWord table consists of six vectors.

compilation state	interpretation state (STATE returns -1)	(STATE returns 0)
non-immediate word (top-of-stack = -1)	optiCOMPILE,	EXECUTE
unknown word (top-of-stack = 0)	doubleAlso,	doubleAlso
immediate word (top-of-stack = 1)	EXECUTE	EXECUTE

The behavior of the hForth text interpreter can be interactively changed by replacing these vectors. For example, one can make the hForth interpreter accept only single-cell numbers by replacing doubleAlso, and doubleAlso with, respectively, singleOnly, and singleOnly. optiCOMPILE, does the same thing as the Standard word COMPILE, except that it removes one level of EXIT if possible. optiCOMPILE, does not compile null definition CHARS into the current definition. Also, it compiles 2* instead of CELLS if CELLS is defined as : CELLS 2* ;.

Special Compilation Action for Default Compilation Semantics

Compiling words created by CONSTANT, VARIABLE, and CREATE as literal values can increase execution speed, especially for native-code Forth compilers. A solution is implemented in the hForth EXE model to provide special compilation action for default compilation semantics. Words created by CONSTANT, VARIABLE, and CREATE have a special mark and xt for special compilation action. The hForth compiler executes the xt if it sees the mark. (POSTPONE must find this special compilation action also and compile it.) A new data structure with special compilation action can be built by CREATE and only two non-Standard words: implementation-dependent doCompiles> and implementation-independent compiles>. doCompiles> verifies whether the last definition is ready for the special compilation action, and takes an xt on the parameter stack and assigns it as the special compilation action of the last definition. compiles> is defined as:

```
: compiles>   ( xt -- )
    POSTPONE LITERAL
    POSTPONE doCompiles> ; IMMEDIATE
```

For example, 2CONSTANT can be defined as:

```
:NONAME
    EXECUTE POSTPONE 2LITERAL ;
: 2CONSTANT
    CREATE SWAP , , compiles>
    DOES> DUP @ SWAP CELL+ @ ;
```

It is the user's responsibility to match the special compilation action with the default compilation semantics. I believe that this solution is general enough to be applied to other Forth systems.

Turtle Graphics

I implemented LOGO's Turtle Graphics in hForth. The turtle moves on the VGA or Hercules graphics screen and follows the postfix Forth command 100 FORWARD instead of the prefix LOGO command FORWARD 100. No floating-point math is used at all. Integers are used to represent angles in degrees rather than in radians, and a look-up table is used to evaluate trigonometric functions. Only a few words are defined in machine code for line drawing and function evaluation. The turtle moves swiftly on a '286 machine. The English Forth source and MS-DOS executable, *ETURTLE.F* and *ETURTLE.EXE*, are in the 8086 hForth package as well as in Korean ones, wherein the turtle understands Korean commands.

Summary

hForth is a small ANS Forth system based on eForth. It is especially designed for small embedded systems. The basic ROM and RAM models are designed for portability; however, they can be easily optimized for a specific CPU to build a competitive system as shown in the 8086 EXE model. hForth has been ported to the H8 processor by Mr. Bernie Mentink. I hope hForth will be useful to many people.

hForth packages for 8086 and Z80 are on-line at:
http://taygeta.com/forth.html or
ftp://taygeta.com/pub/Forth/Reviewed/

Dr. Wonyong Koh (wykoh@pado.krict.re.kr) is a professional chemist and a sophisticated amateur programmer. He taught himself electronics and computer programming during his college years. He met *Starting Forth* in a bookstore ten years ago and has been fascinated with it ever since. He currently works on electronic thin films in the Korean Research Institute of Chemical Technology.

Using Forth to manipulate the real world

Forthware

Controlling DC Motors

Skip Carter
Monterey, California

Introduction

Last month's codeless column described the circuitry required to control electrical power. This time we will look at how we can use software to manipulate those circuits in order to control a DC motor.

DC Motors

Motors that are driven by direct current come in a vast assortment of physical sizes, with or without integral reduction gears, and with or without shaft-position encoders. Electrically, these motors come in three basic forms:

- The *series motor*, Figure One-a. These motors have the field windings in series with the armature. They have a high starting torque. When lightly loaded, they have a tendency to increase in speed with time to the point that they can actually be damaged, so they should not be run without a load or a speed control mechanism.
- The *shunt motor*, Figure One-b. These motors have the field windings in parallel with the armature. Shunt motors have less starting torque than a comparable series motor, but they are usually very good. These motors have the property that, once spun up, they approach a set speed and maintain a nearly constant speed over a wide range of loads.
- The *compound motor*, Figure One-c. A compound motorhas both series and parallel field windings. They have the high starting torque characteristics of series motors and the constant speed characteristics of shunt motors.

From the perspective of *controlling* the motor, all three of these motors are identical.

Controlling the Motor Speed

We can drive a DC motor with our transistor circuit for controlling DC power to inductive loads. Using a single digital output bit to control the transistor will turn the motor on (full speed) when the transistor is on, and will turn it off when the transistor is off. What about running the motor at different speeds? The speed at which a DC motor runs (for a given load) is proportional to the magnetic field strength in the coils. This field is proportional to the applied current, so it would appear that we need to control the current. (It is the way the equations for the coil current work out for series motors that make them tend to "run away" at no loads.) But focusing on the primary requirement of controlling the magnetic field strength suggest an approach that is much more convenient from a digital control perspective. Recall that those coils store current (causing us to be preoccupied with "fly-back" or "shunt" circuits in order to handle this current when it's released). Now we can use that current-storage property to our advantage. When we turn on the controlling transistor, we charge up the coil. If we then turn the transistor off, the voltage across the coil drops and the field collapses, *but this takes time.* So if we turn the transistor back on again before the field fully collapses, it starts to climb back up before it ever gets to zero. The shorter we make the off time, the higher the average field strength—up to the point where the on-time is continuous and we get a fully on motor coil. So, by controlling our transistor on and off rates properly, we can vary the field strength from full on to full off.

The typical way to do this is with a *pulse-width modulated* signal (PWM). With this type of signal, the cycle time is fixed (at, say, one kilohertz) and the *duty cycle* (the proportion of the cycle time during which the signal is on) is varied. Listing One, pwm.fth, provides a simple implementation of this type of control.

Changing Motor Direction—The H-Bridge

For stepper motors, we could change the direction of rotation by just reversing the order that we presented the coil patterns; for DC motors, we have to reverse the direction of the current through the coil. Conceptually, we need to have *two* transistor switches in order to do this. One provides a current path in one direction through the motor, the other switch controls current in the opposite direction. Probably the simplest way to achieve this is to use a relay driven by transistor switches. But if we are going to be using PWM speed control, relays are not a very attractive choice because the PWM will be cycling them too fast and they would quickly wear out even if they could switch at 1 KHz. We can build a transistor switch if we combine a high-side and low-side switch into a single circuit, like in Figure Two. With this circuit we apply a PWM signal on one transistor and turn the other off to get one direction of rotation, and turn the first one off and apply a PWM signal to the second to get the opposite direction of rotation. The two transistors should not be simultaneously switched on.

This simple switch has the disadvantage of requiring a dual polarity power supply. It would be much more convenient to have a circuit that gave us the ability to switch current directions without a dual-rail power supply. Such a circuit is known as an *H-bridge*. Unfortunately, the ***implementation*** of such a bi-directional current switch requires a combination of both low-side switching *and* high-side switching, as before. In addition, it turns out to be more flexible if we do this by using *n*-type devices for the low-side switches, and *p*-type devices for high-side switches. Now we have done two things that, last time, we concluded we wanted to avoid: i) the use of high-side switching, and ii) the use of *p*-type transistors.

The ultimate result is the relatively complicated circuit in Figure Three. (I am showing a bipolar transistor implementation which requires a voltage level shift on the control signals because of the above listed compromises; this is what the LM-324 op-amps are for. H-bridges are also frequently implemented using MOSFET transistors.) If we implemented the H-bridge with only *n*-type transistors, the control lines would connect the bases of 1 and 4 together and 2 and 3 together, instead of the way we implement it here. The strictly *n*-type H-bridge requires that the motor voltage be significantly higher than the control voltage (say 12 volts), whereas the kind with both types of transistors can work down to just slightly higher than the control voltage (like six volts). In any case, this circuit is messy enough that we have crossed a threshold of practicality: Unless you have a very exotic requirement, or are doing this for pedagogical reasons, it's simpler to use an integrated circuit implementation of an H-bridge (such as the Motorola MPC1710A or the SGS Thompson L293D). If you decide to use an IC H-bridge, just make sure it can handle the current loads you anticipate will be required—typical chips can drive the motor with currents in the range of .5 to three or four amps.

Listing Two, hbridge.fth, shows a typical H-bridge control program. Notice that we can do the typical Forth thing and build upon the PWM code we already have. To rotate in one direction, transistors 1 and 4 are on, running current through the motor one way. The other rotational direction is accomplished by switching on only transistors 2 and 3.

Note that IC H-bridges are usually controlled by a clock and direction pin, but the discrete component one shown here uses two clock lines, one for each direction (for a given direction of rotation, one side gets clocked and the other side is held low). So the driving code for an IC H-bridge will look somewhat different from the code shown here.

The use of the H-bridge also gives us the useful capability of *active* braking of the motor. Up until now, we stopped by turning off the controlling transistors and then coasting to a halt. But, for some implementations of the H-bridge, one can actually draw energy *out* of the coils when the motor is spinning, making the motor work against itself and thus achieving *dynamic* braking by switching on both transistors 1 and 2 simultaneously.

Figure One. The three basic types of DC motors.

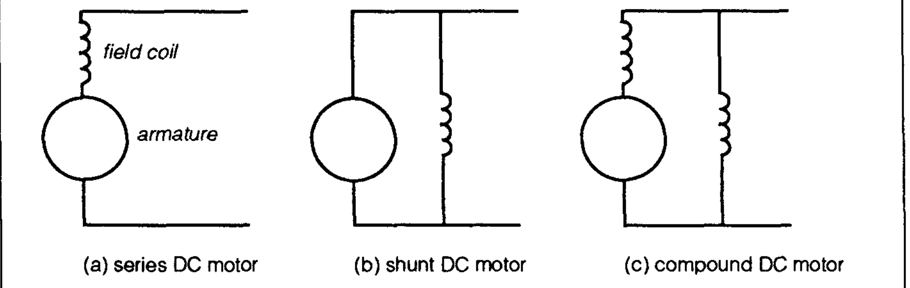
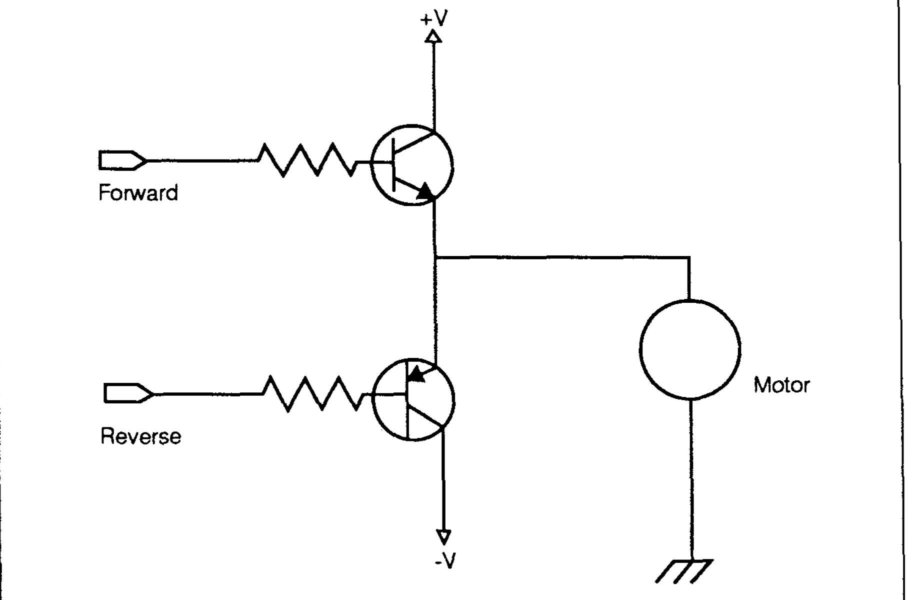


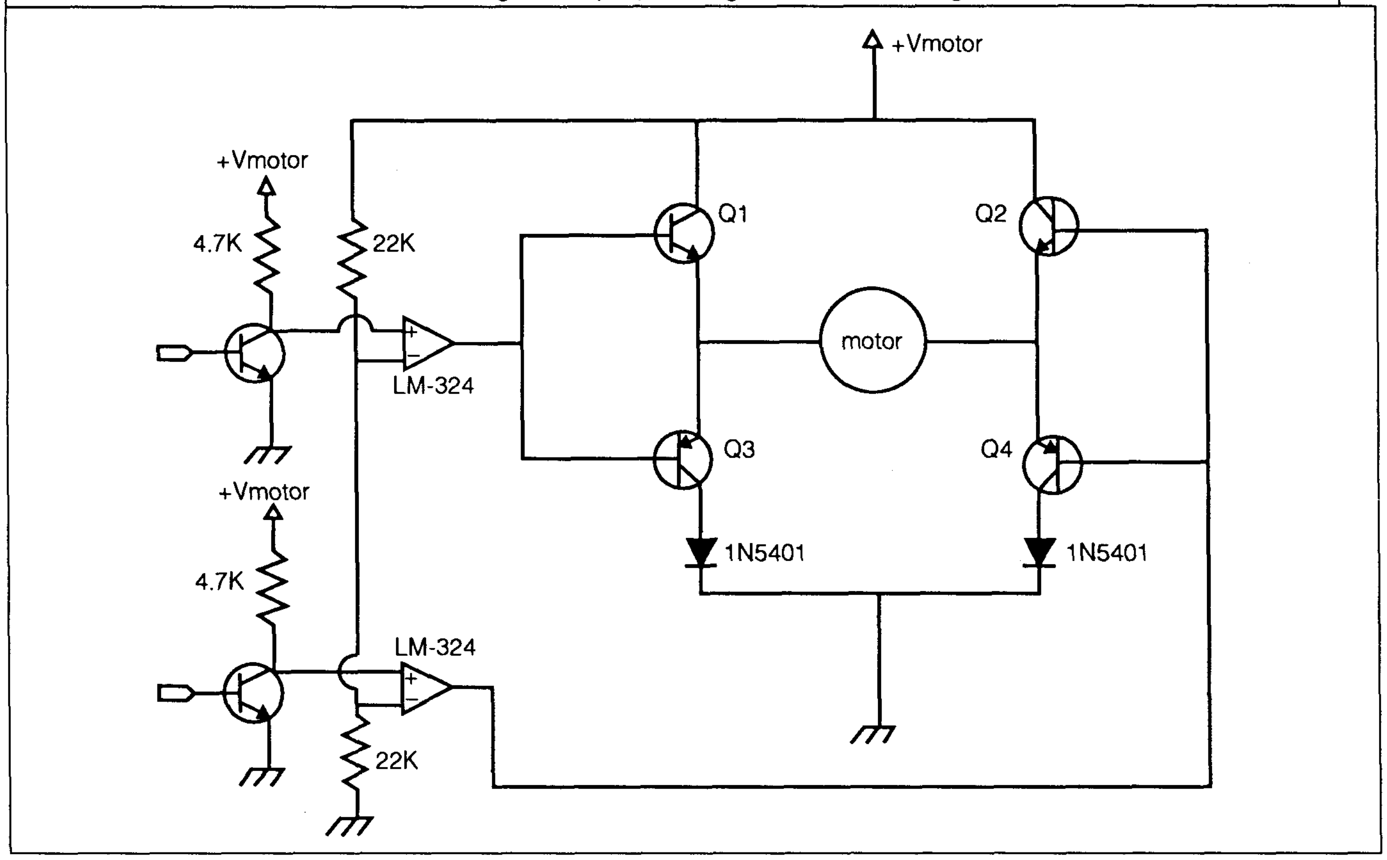
Figure Two. Switching the direction of a DC motor with two power transistors, for a dual polarity voltage supply.



Bipolar Stepper Motors

In our first column [*FD* XVII/5], I mentioned the four-wire *bipolar* stepper motors. These motors require proper sequencing of the direction of the current through two sets

Figure Three. An H-Bridge using NPN (e.g., TIP-121) and PNP (e.g., TIP-126) power Darlington transistors. The two unmarked NPN transistors are general-purpose signal transistors (e.g., 2N3904).



of coils. [See Table One.] We can do this if we have *two* H-bridges, one for each coil. Now we are using lots of transistors, eight of them, to control a single motor. Fortunately, many H-bridge ICs (such as the L293D) are dual H-bridge chips.

Table One. The bipolar sequence.

Step	C_{11}	C_{12}	C_{21}	C_{22}
1	-V	+V	-V	+V
2	-V	+V	+V	-V
3	+V	-V	+V	-V
4	+V	-V	-V	+V

So step one is the condition where coil 1 has a negative voltage on wire 1 and a positive voltage on wire 2, coil 2 has a negative voltage on wire 1 and a positive voltage on wire 2. Step two reverses the current in the second coil, leaving the first alone.

Listing Three, bipolar.fth, shows the code to drive a bipolar stepper motor through the above sequence.

Conclusion

With this installment, we have pushed very hard against the limits of what is reasonable to do with discrete components and high-level code. It might not look like the code presented here is particularly special, but it was designed to be as deterministic as possible for this real-time application. In this way, the motor's speed is as uniform as possible. By the way, here is a little experiment for you MS-DOS/Windows users: Take a look on an oscilloscope at the pulse timings when running the PWM code in a Windows shell and when running in native MS-DOS mode. If you do this experiment, you will never have to ask whether it is reasonable to run a true real-time application from Windows—you will know the answer.

In a typical microcontroller that might be used in an embedded application that needs to control a DC motor, you are likely to have an on-board timer or I/O coprocessor that can be used to generate the motor control signals. This type of feature should certainly be taken advantage of—if it is available, it potentially can help you to avoid the need to use a more powerful (and more expensive) processor for a given application. But if you follow what we did here, you will understand the principle of DC motor control regardless of the implementation details your application will dictate.

Now that we can turn things off and on and move things around, we need to focus our attention on measuring how the real-world is responding to our manipulations. Without *input*, we can't really control systems. Next time, we will begin looking at getting input from the environment.

Please send your comments, suggestions and criticisms to me through *Forth Dimensions* or via e-mail at skip@taygeta.com.

Skip Carter is a scientific and software consultant. He is the leader of the Forth Scientific Library project, and maintains the system taygeta on the Internet. He is also President of the Forth Interest Group.

Listing One. pwm.fth

```
\ pwm.fth           Simple PWM of a parallel port pin for controlling a DC motor.
\ $Author:   skip  $
\ $Workfile:   pwm.fth  $
\ $Revision:   1.1  $
\ $Date:   28 May 1996 02:38:24  $
\ This code is released to the public domain. Everett Carter, May 1996

\ code to set up the parallel port
S" /usr/local/lib/forth/fcontrol.fth" INCLUDED

\ ==============================================================
\ We will just assume that bit 0 is the motor,
\ but by playing games with the following masks we could
\ control multiple motors

0 VALUE off-mask
1 VALUE on-mask

\ ===================== Timing Control ==========================
VARIABLE hitime                \ 0..cycle-time, the motor will stall
                               \ for very small values (like 4 or 5),
                               \ depending upon the motor

VARIABLE cycle-time            \ try something like 100, depends upon
                               \ the motor and the CPU

\ The ANS MS is too coarse, we want the WHOLE CYCLE TIME to be
\ on the order of 1 ms, so we are falling back to the old standby
\ idle loop.  The values used here may need tuning for different
\ motors and CPUs

100 VALUE delay-time         \ cycle time granularity

: delay delay-time 0 DO LOOP ;

\ ==============================================================
: verify ( -- )        \ make sure we have useful settings

    hitime     @ 0 <  ABORT" illegal hitime (< 0)"
    cycle-time @ 1 <  ABORT" illegal cycle-time (< 1)"

    hitime @ cycle-time @ > ABORT" hitime > cycle-time"
;

: init ( -- )
    verify
    init-port TO #PORT
;

: run_motor ( -- )       \ run motor until keystroke received
    BEGIN
        hitime @ 0 > IF on-mask #PORT pc! THEN

        cycle-time @ 0 DO
           I hitime @ = IF off-mask #PORT pc! ELSE
                           KEY? IF KEY DROP UNLOOP EXIT THEN
                        THEN
           delay
        LOOP

     AGAIN
;

\ ==============================================================
: run ( -- )          \ initialize, run, and clean-up
    init
    run_motor
    off-mask #PORT pc!
    #PORT CLOSE-FILE DROP
;
```

Listing Two. hbridge.fth

```
\ hbridge.fth         Driving a DC motor with an H-bridge

\ $Author:   skip  $
\ $Workfile:   hbridge.fth  $
\ $Revision:   1.1  $
\ $Date:   28 May 1996 02:40:22  $

\ This code is released to the public domain. Everett Carter, May 1996

\ code to set up the parallel port
S" pwm.fth" INCLUDED

\ ==============================================================
\ We will just assume that bits 0 and 1 drive the motor,
\ but by playing games with the following masks we could
\ control multiple motors

: forward
    1 TO on-mask
;

: reverse
    2 TO on-mask
;

\ usage:  forward (or reverse) run
```

Listing Three. bipolar.fth

```
\ bipolar.fth         Driving a Bipolar stepper motor with a dual H-bridge

\ note: much of this code closely duplicates the code from the
\       Jan/Feb 1996 listing: steppers.seq.  The primary differences
\       are the values of the sequence array and that it is assumed
\       that the pins drive a dual H-bridge like Figure 3.

\ $Author:   skip  $
\ $Workfile:   bipolar.fth  $
\ $Revision:   1.1  $
\ $Date:   28 May 1996 02:42:46  $

\ This code is released to the public domain. Everett Carter, May 1996

\ code to set up the parallel port
S" /usr/local/lib/forth/fcontrol.fth" INCLUDED
S" /usr/local/lib/forth/fsl-util.fth" INCLUDED
S" /usr/local/lib/forth/structs.fth"  INCLUDED

\ ==============================================================
\ We will just assume that bits 0 and 1 drive the motor coil 1,
\ and bits 2 and 3 drive motor coil 2

structure: sequence
    integer: .n
    integer: .index
    4 integer array: .s{
;structure

sequence bipolar
```

(Listing Three continues...)

```
: init-seq ( -- )

    4 bipolar .n !
    0 bipolar .index !
    2 bipolar .s{ 0 } !
    4 bipolar .s{ 1 } !
    1 bipolar .s{ 2 } !
    8 bipolar .s{ 3 } !
;

-1 VALUE direction?

12 VALUE wtime

: idx++ ( seq_hdl -- idx )      \ increment the index, return old value
    2DUP .n @ >R
         .index DUP @
    DUP 1+ R> MOD
    ROT !
;


: idx-- ( seq_hdl -- idx )      \ decrement the index, return old value
    2DUP .n @ >R
         .index DUP @
    DUP 1- R> OVER
    0 < IF 1- SWAP THEN DROP
    ROT !
;

: fsteps ( seq_hdl n -- )
    0 DO wtime MS
       2DUP idx++ >R
       2DUP .s{ R> } @ #PORT pc!
    LOOP

    2DROP
;

: rsteps ( seq_hdl n -- )
    0 DO wtime MS
       2DUP idx-- >R
       2DUP .s{ R> } @ #PORT pc!
    LOOP

    2DROP
;


: reverse ( -- )                \ toggles rotation direction
    direction? IF 0 ELSE -1 THEN
       TO direction?
;

: steps ( n seq_hdl -- )
    ROT
    direction? IF fsteps ELSE rsteps THEN
;

\ ==========================================================
init-seq

\ typical usage:    12 bipolar steps
```

Call for Papers
FORML CONFERENCE

The original technical conference for professional Forth programmers and users.

18th annual FORML Forth Modification Laboratory Conference
Following Thanksgiving November 29 – December 1, 1996

Asilomar Conference Center
Monterey Peninsula overlooking the Pacific Ocean
Pacific Grove, California, USA

Theme: Experimenting with the ANS Forth Standard

The ANS Forth standard has been out for two years, and the review process will start in another two years. During the development of the standard, the lack of "common practice" led to many last-minute experiments. FORML, with its charter as Forth's "Modification Laboratory," is the appropriate place to let others know what your experiences have been as a developer or user while there's time for your ideas to spread.

Papers are sought that report on your experience writing ANS Forth programs and systems. That is, on your experiments. What worked, what didn't? How easy or difficult was it to...? Are ANS programs really portable? Where were the "gotchas" in writing the half-dozen or so public-domain ANS systems? How are you checking that your program or system really does comply? What has it been like to convert your customer base to ANS? Or is it worth doing at all?

Has documentation improved because of the ANS examples? Is it easier to read another's code? Have you seen any change in Forth's acceptance? What is needed for there to be a truly international standard?

By calling attention to the successes and the problems now, before the review process begins, we hope others will repeat your experiments, confirming or refuting your hypotheses. Can an alternative to DOES> really resolve syntactic problems and make meta-compilation easier? Can a tethered system be compliant and efficient? Would it make sense to have various common groups of environmental restrictions labeled "Forth models"?

Please, whether your ANS experiment was one line or a thousand, whether it succeeded or failed, or can be described in one page or ten, bring it to this year's FORML Conference to share with the world. As always, papers on any Forth-related topic are welcome.

Mail abstract(s) of approximately 100 words by October 1, 1996 to FORML, P.O. Box 2154, Oakland, California 94621 USA, or e-mail to FORML@ami.vip.best.com. Completed papers are due November 1, 1996.

The Asilomar Conference Center combines excellent meeting and comfortable living accommodations with secluded forests on a Pacific Ocean beach. Registration includes use of conference facilities, deluxe rooms, meals, and nightly wine and cheese parties.

John Rible, Conference Chairman **Robert Reiling, Conference Director**

Registration and membership information available by calling, fax, or writing to:
Forth Interest Group, P.O. Box 2154, Oakland, CA 94621
510-893-6784, fax 510-535-1295

Conference sponsored by the Forth Modification Laboratory, a Forth Interest Group activity.