$10 Volume XVIII, Number 1 May 1996 June

FORTH DIMENSIONS

Safety Critical Systems

Taming Variables and Pointers

LIFE That Knows When to Stop

OOF Doesn't Have to Be Slow

Intro to Power Control

More Than a State Machine

DOS Disk Access for non-PCs

Contents

Features

Departments

Editorial

Forth Dimensions
Volume XVIII, Number 1
May 1996 June

Published by the
Forth Interest Group

Editor
Marlin Ouverson

Circulation/Order Desk
Frank Hall

Forth in Cyberspace

Informal surveys would have us believe that only one-third to one-half of Forth Interest Group members are on line. And my impression is that a significant portion of on-line Forth activity is contributed and/or consumed by Forth users who are not FIG members. I wonder what would come of better cross-pollination between the two groups, and surmise that the interaction would benefit both camps.

I can't do much to persuade on-line professionals to join FIG, except publish a good magazine and refer users with questions to FIG resources, digital or otherwise. But there are ample reasons why every FIG member should have a modem and use it to establish an Internet connection. As one correspondent told me recently, he realized it is about time to join the 1990s.

Much of the popular hype about the World Wide Web emphasizes its graphical features, form over content. Communicating graphically usually means pushing lots of bits over the phone lines, resulting in a click-and-wait interface at remote locations like mine, where the local connection is still at a measly 2400 bps (though I can always dial long-distance or endure a surcharge for 14,400 or 28,800). But the wait is often worthwhile and, besides, the Internet still provides plenty of character-based resources for Forth folk.

E-mail is mostly taken for granted these days, but there's no denying its value. It's a matter of seconds to contact an author of a *Forth Dimensions* article to ask a question, propose an alternate solution, or request some code. Most Forth vendors are available for e-mail tech support and advice. There are even e-mail-based SIGs: you can join current discussions about Forth in robotics, Linux (see "Letters" in this issue), MOPS, Win32Forth, and safety critical systems, or you can propose a topic of your own and find people joining from around the world. Anything posted to one of these group addresses is received by all its members as e-mail, and recipients can reply to the group as a whole or to an individual. Working the on-line angle this way greatly compensates for the lack of local expertise or interest in a subject that is important to you.

I send lots of e-mail, especially during magazine production cycles. I sometimes begin an on-line session by mailing to various authors and, time zones notwithstanding, may receive replies from Russia, Korea, *et al.* before I sign off. Not bad, for the cost of a local telephone call and a couple minutes of connect time.

Your Internet service provider can explain how to access on-line newsgroups, including the one called "comp.lang.forth"—once there, you'll see a list of current topics that might include anything you've ever wondered, criticized, or praised about Forth, and more. It's a free-wheeling, unmoderated format, with not-quite-real-time questions (and answers) about Forth systems and techniques, project proposals, idea exchanges, code critiques... you name it. It is an excellent resource, and has helped many to find the right Forth system, to surmount a bump in Forth's learning curve, to interpret a nuance of ANS Forth, and generally to find Forth compatriots (neophytes and experts) and even collaborators. If you are new to newsgroups, look for a topic containing the word "FAQ" to help explain the on-line social conventions and to answer "frequently asked questions."

If you choose an Internet service provider that offers connection to the World Wide Web (hopefully at 14,400 bps or better), you'll find some of the newest and most exciting Internet-based Forth resources. Start at the Forth Interest Group "home page" (http://www.forth.org/fig.html) maintained by FIG President and *FD* columnist Skip Carter. Here are a few features you will find:

Forth Scientific Library Project—status updates, coding guidelines, reviewers' names and e-mail addresses, and the ability to download reviewed and accepted code contributions.

(Continues on page 6.)

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. Submissions may be edited for clarity and length.

A Forth History

Dear Editor,

Why has Forth not come into wider use and acceptance? Its status as a "cult" language has been a disappointment to many of us. We know Forth is a seminal, landmark contribution to computing, and it hurts us to see it "languish" unrecognized and unused (well, it's not entirely unused).

I have followed the fortunes of Forth since the early days of more than fourteen years ago, when Bill Ragsdale put together a valiant team of programmers to port Forth to various microprocessor environments. My first copy came from that group and ran on an Apple II.

At the time I was on staff at SRI, and one of my jobs was to watch and be knowledgeable about the state of and the advances in computing, especially the burgeoning microprocessor area. I watched the growth of Forth. I also followed other languages (e.g., C, Pascal, and Ada). I collected reams of data and experiential material for each processor and all languages. At first I believed the stuff from Bell Telephone about C and UNIX, and the stuff from UCSD about Pascal. I was skeptical about what I could see of Forth. It sounded too good. Like a few others, and unlike most of the computer community, I did an in-depth

I was skeptical about Forth. It sounded too good.

investigation and formed the opinion that most of the enthusiasm for C, UNIX, Pascal, etc. was truly cult hype. Forth survived my jaundiced cynicism and gained my respect. I took to using Forth myself.

Now, in those days, it came to pass that SRI won a contract to write a communications simulation program for the military. The group that had won the contract came to me and asked what sort of computer they should purchase. I asked the usual questions and, in so doing, found that they planned to write an astounding number of lines of code (compared to the value of the contract). I asked them to revisit their estimate of code volume. They did and replied to me that they really did believe in their original estimate (it later turned out that they were correct).

The team was chagrined to learn that, by their own estimate and using the lowest estimating factors then available, this valuable contract had been underbid by a factor of more than ten. They were going to put SRI (a not-for-profit organization) more than $3 million into debt on this project! I suggested that they should refuse the contract. This they, paradoxically, did not want to do.

By this time in my career, I already well knew that computer "science" is operating as science was in the Aristotelian era. That is to say, such "science" is done by committees arguing among themselves whilst sitting around tables. If any naive upstart should barge in and have the temerity to say, "Noble persons, I have gone out into the world, made an observation, and the results are..." the assembled bards would reply, "So much the worse for the facts." And there it will lie. This happened to me many times. Make no mistake—this attitude is even more deeply entrenched in so-called computer science today. Since I still make my living in the world of computers and am a practicing "expert" in C, C++, and X, even now I am somewhat reluctant to write this letter.

Thus, at the time, I hesitated to place my career on the line by recommending something heretical like Forth. But right is right, and sometimes you have to go with what's right. So, I swallowed hard and suggested that SRI might save the project if the team would use Forth as the language. I showed the relevant managers the data I had collected indicating that Forth was about ten times more productive than any other language (they were planning to use Pascal).

Predictably, they laughed. Or rather, they were embarrassed even to think about having to tell their customer that they would use what they and the customer considered a hobbyist's language (expletives deleted). They were, however, well and truly painted into a corner. And, in the end, they made me lay my reputation on the line again and brief their customer on the problem and my proposed solution. They made me suffer all the derision—and there was lots of it. After a while I convinced the customer that, unless they wanted to fork over another $3 or $4 million, they would have to let SRI use Forth. They finally agreed to go Forth.

SRI bought an H-P computer, and H-P graciously allowed one of their employees (a FIG member) to place Forth on their machine for the project for free. What was the result, you ask? Using programmers who were totally unfamiliar with Forth, SRI brought the project in on time, on budget, tested, and documented with very nearly the number of lines of code they first estimated. That should have been a stunning, earth-shattering vindication of Forth that caused everyone (at least at SRI) to rush into the Forth camp. As you can see, this brilliant, $3+ million dollar achievement had no appreciable effect on the community. Could this be a lesson that the computer community does not care about money or time?

That outcome further convinced me that my generally cynical view of the computing community (I won't call it a science or engineering) was solidly grounded. Neither engineering nor science is applied to computing. I have seen nothing in the years that followed to suggest I should change my view.

What's wrong with Forth? I don't know of a thing. It certainly is anathema when, in fact, Charles Moore should get the Nobel prize for his accomplishment.

I do, however, know what I personally want. Maybe there are others who want it, too. I want a PC that contains nothing but Forth—no DOS, certainly no UNIX, no C, no C++, no BASIC, Pascal, or Fortran. It should use the Xerox

PARC/Macintosh/NeXT Step/Windows graphical operator interface paradigm (please, absolutely no X), and the underlying operating system code should be in Forth. I don't want to co-exist with DOS/Windows/UNIX, I want command of the entire machine. What about application programs, you say? As far as I am concerned, "build it and they will come." Ideally, the processor in the PC would be a Forth engine, however, even the ghastly xx86 family architecture will do—i.e., give me a Forth machine, not just Forth add-in boards.

In my humble opinion, workstations, minicomputers, and mainframes are all dying and will, in the not very distant future, be replaced by networks of PCs. While this is not exactly the point of my letter, it could provide motivation for someone to offer a real alternative. Somewhere in the world, there must be a person or organization with enough money, vision, and guts to buck the raging current and who wants to do humanity a true favor like this.

Philip R. Monson
Kekaha, Hawaii

On Target

Just received the March-April 1996 issue—*wow!* It's one of your best yet!

Since I work with (and teach classes in) the C language, Frank Sergeant's article on integrating Pygmy and C was extremely interesting. And, having just installed Linux on a new Pentium (Bill Gates can't tell *me* to go to Win95 or else!) Skip Carter's tutorial on accessing the hardware in Linux was about as timely as you can get.

A couple of questions:

Have you considered getting someone to write a series of articles on Forth and Linux? There isn't much written documentation out there for Linux—we're starved for information on this really neat operating system. However, with Linux gaining in popularity every day, this would be a really valuable resource for Linux-based Forth-a-holics.

Have you ever considered making the code from each issue available on diskette or via FTP? It would sure be neat to be able to order a diskette or access a BBS, etc. for the code rather than make the 1,001 mistakes I typically make in typing in the listings!

Any chance that we (FIG) could obtain copies of the Linux Forth implementations mentioned in Skip's article? At the least, could you find out and publish their location on the WWW?

Thanks again for a super issue.

Tom Bohon
tomb@hecb.wa.com
Olympia, Washington

Thank you for your comments. Forth Dimensions *is very reader-oriented so, if we hear from more members wanting Linux (or other) materials, the content will adjust to reflect your interests. And yes, code for Linux Forth implementations is available on-line at FIG's site: check out ftp://taygeta.com/pub/Forth/Linux for current offerings. (Meantime, see András Zsótér's article and a note about a new Linux Forth in the conclusion to "Forthware," both in this issue.)*

We are trying to be more assiduous about putting the code we print in convenient on-line locations for downloading, provided that is all right with the respective authors. If you can't find what you are looking for at, for example, ftp://taygeta.com/pub/Forth/FD/1996 then send an e-mail note to me (editor@forth.org) asking if it can be made available. As to providing code on diskettes, I will refer your query to the Forth Interest Group's office (office@forth.org) to see if it is logistically feasible at this time.

Corrigendum

In paragraphs 8 and 9 on the right-hand column of page 21 of "Differential File Comparison," somehow "fail" *became* "happen"*. The two paragraphs should be:*

In twenty years of use this has hardly ever failed. In the very few times it has, the effect has been negligible. (You can tell that it has failed when an insertion appears just before a deletion.) It's at least seven years since I've seen it fail.

Of course you can force it to fail by using a poor hashing function. However the hashing function doesn't have to be sophisticated. The one used here works fine with 32-bit or 16-bit arithmetic.

(Editorial, from page 4)

Want ads—as of this writing, seven employers were seeking Forth programmers to fill jobs in a number of fields.

Resumes—FIG members can post their resumes and can include links to their own web home pages (ever wonder what happened to Leo Brodie?). FIG Chapters can do the same: the group in Maryland has taken advantage of this to coordinate aspects of their Forth OS project.

Forth Dimensions—An on-line sampler is included with a few articles, graphics, illustrations, and code from recent issues. Send your colleagues here to preview a little of what FIG membership will provide. You'll also find downloadable code from many of our recent issues at this site.

Forth conferences—You can link to sites about the euroForth and Rochester conferences, including details about upcoming and past conferences, and contents of their proceedings.

Forth bibliography—This interactive database of published Forth papers is rich with content, and will grow as its sponsors expand the range of publications it includes.

There is much more, like relevant IEEE docs, Open Firmware info, valuable ANS Forth documents, links to Forth vendor sites, and a section only for FIG members (have your membership number handy, it's on the *FD* mailing label). The features are growing, so, even if you've checked it out in the past, it's a good idea to stop in regularly to see what's new.

I'll look forward to meeting you on-line!

—Marlin Ouverson
editor@forth.org.

More Than a Simple State Machine

John Rible
Santa Cruz, California

Over the last year I've been working with a home-schooled teenager, helping him learn how computers work. Using my QS2 processor design and Clive Maxfield's *Bebop to the Boolean Boogie*, he's gone through Boolean algebra, combinatorial and sequential logic, and is now starting in on state machines. At last Fall's Northern California Forth Day, Dr. Ting talked me into organizing a programming contest based on some "traffic intersection" boards that he had made up for his programming course. They seemed ideal to illustrate state machines to my student also, but I didn't want to take too long a detour into teaching him Forth before we made some lights flash. The solution was obvious: I'd just write a simple state machine "mini-language" in Forth!

So while he was working out the state transition diagrams, I sat down to write some code. Since some of the contestants at Forth Day almost had the intersection working in an hour, I figured a day should be enough time to write just the state machine part. And I was only off by my usual factor of two: it took about two days to get the syntax pretty solid. But the code in this article represents another 3–4 days of exploring alternatives and filling in some gaping potholes in my original code. My student has now completed the project, and we hope to have a joint article describing his work ready for the next issue.

Start at the End

In our usual Forth style, I'm going to begin describing the lexicon at the end, with an example of its use, and then backtrack to fill in how the lexicon is coded. Table One shows the transition table of this fairly meaningless example. The goal of this "machine" is to repeatedly produce the sequence "1 2 3 " (that's a one followed by one space, a two followed by two spaces...) at the rate of one character per clock period, allow the operator to interrupt the machine at any time, and finally, to stop after a fixed number of periods. To accomplish this, the counter is cleared on entry to a new state and incremented otherwise. Both the operator halt and the timed exit are higher priority inputs than the count, and would require increasing the size of the table to properly describe their effects. (You're right, I did put the table together after I'd worked out the example.)

Because I'd started with a different concept than the usual software model of a state machine, one much closer to the hardware logic my student and I were working with, the lexicon doesn't look, or act, at all like the table-format lexicon Julian Noble shows in his book *Scientific Forth*. I was trying to highlight the difference between the combinatorial logic equations that generate flip-flop inputs and the clocked flip-flops themselves. This approach turns the normal method on its head, looking at how one enters the next state rather than how one leaves the current state. Think about that sentence while looking at Figure One, a typical "circles and arrows" state transition diagram, but focus on the arrowheads entering a state rather than the tails leaving it.

The result models the logic equations, clock, and flip-flops as a set of software equations that can be directly represented as a circuit diagram. A major drawback to its use in anything other than small, fairly slow problems, however, is that while the hardware "executes" the equations in parallel, the software ones are executed serially, on every clock tick. This can consume a substantial portion of the clock cycle for large machines with fast clocks.

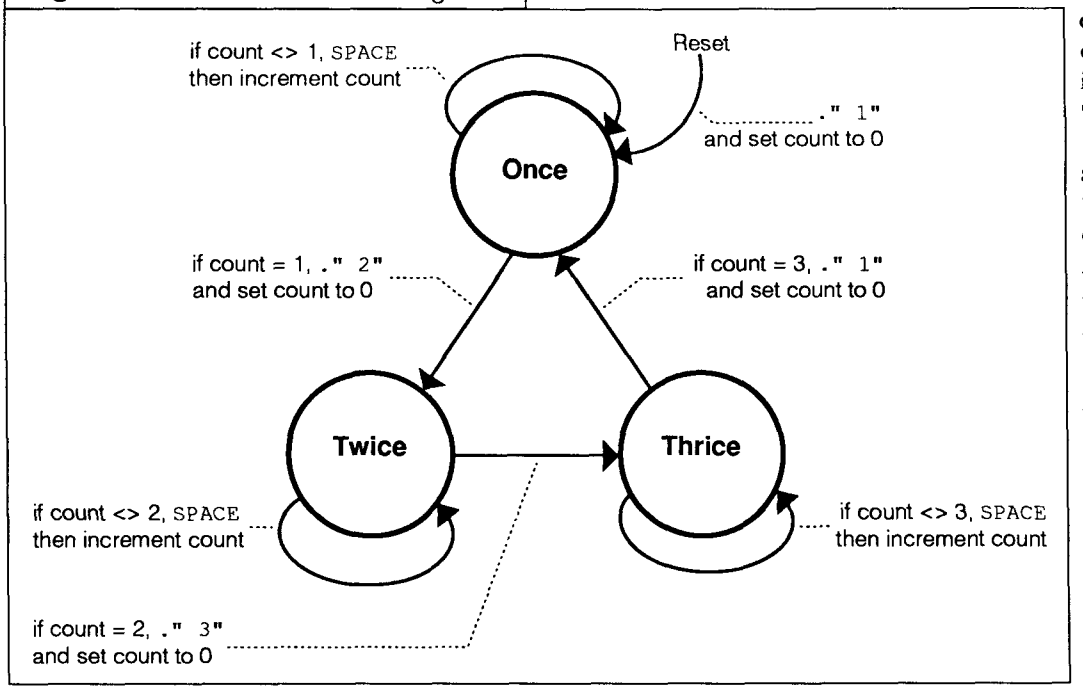
Step Through the Demo

Listing One shows the code for this example. The example describes a "one-hot" machine, characterized by

Table One. The transition table.

State	Count			
	0	1	2	3
Reset	count=0. >Once	error	error	error
Once	." 1" , +1. >Once	SPACE , =0. >Twice	error	error
Twice	." 2" , +1. >Twice	SPACE , +1. >Twice	SPACE , =0. >Thrice	error
Thrice	." 3" , +1. >Thrice	SPACE , +1. >Thrice	SPACE , +1. >Thrice	SPACE , =0. >Once

Figure One. State-transition diagram.



execution or starts the cycle over again, depending on the flag from TICK-TOCK.

Finally, the "next-state," "output," and "clock-period" behaviors defined above are ASSIGNed to the machine and the states, filling in the empty slots created when they were defined way back at the beginning.

To start the machine, just type RUN. Execution will end after 50 clock ticks if you have resisted the impulse to stop it sooner. Just re-enter RUN to pick up where you left off if you did stop it.

having a bit for each state, only one of which is on at a time. It is also a "Mealy" machine, whose outputs can change when the inputs change as well as on state changes. It is a "clocked" machine, changing states synchronously with an external clock pulse. There isn't room to give a tutorial on state machines here, so I'll just give a brief rundown of the DEMO machine and a few implementation notes on the code. Everything here is ANS Forth, amazingly enough, so you should have few problems getting it to run in your environment.

The name of the machine, DEMO, and its states ONCE=, TWICE=, and THRICE=, are defined first, so that they may be used in the next-state logic equations. The order in which the states are defined within the machine determines the order in which the next-state logic equations are evaluated, creating a priority ranking. The first state defined has the highest priority. Executing a state name returns TRUE when it is the current state, FALSE otherwise.

The next-state logic words, >ONCE?, >TWICE?, and >THRICE?, describe the state transition logic, returning TRUE when that state is to be the next "current state." They are checked, in order, on the "rising-edge" of the clock and the first one to return TRUE becomes the next current state. The machine also remains in the same state if none of the equations returns TRUE on a particular clock tick. Turning power on is simulated by the word RESET=, which only returns TRUE when no state is currently active.

The output words ONCE., TWICE., and THRICE., are executed just after the "rising edge" of each clock tick the specified state is active, followed by the clock-period word. In this example, the clock-period word TICK-TOCK waits the amount of time specified by PERIOD and then checks the keyboard for an operator interrupt or for having reached the elapsed-time limit, leaving TRUE on the stack if either is true. Although DEMO doesn't have any other inputs, you'd probably want to check them periodically in the output or clock-period words. After the clock-period word executes, the machine increments its tick counter and either stops

Under the Hood

Listing Two defines the state machine lexicon. I've purposely written the code in many styles, to illustrate the best (and the worst) of Forth documentation. I've grown fond of the style shown in STEP, where the stack is shown at the beginning of a line, only when it has changed from the line above. My primary goal, however, was to create just enough of a lexicon that my student could avoid the difficult parts of Forth, but still be coding in Forth. We'll see next time whether I was successful.

Both MACHINE: and STATE: compile structures with three data cells and one link cell which are filled in by ASSIGN. In order to nest multiple machines, the contents of current-machine must be pushed on the stack in the DOES> phrase of MACHINE: and popped back when exiting the machine in STEP. The guts of the machine are all in STEP, which I must leave as an exercise for the reader. Although it is a bit convoluted, it illustrates the use of the stack and control structures, and absorbed many hours getting it "right."

Notice that TICK#, which returns the number of time periods the machine has been in the current state, does a subtraction to get that value. There is just one counter being incremented (machine ticks) which is copied into the state being entered. Although this technique isn't talked about a lot, it can often make your code simpler, faster, and easier to maintain. (Some systems use a similar method to generate I.)

Yes, do-tick and do-output are more complicated than absolutely necessary, but they allow the machine to be RUN under all the error conditions that I could dream up and that my student stumbled into. The only debugging I've found necessary is to single-step the machine and display its contents with DEBUG. You'll soon catch on to the meaning of the various fields.

As I briefly mentioned in the code, this engine is more general than a finite state machine. It can model both Mealy and Moore (no, a different one) machines, but from the perspective of "how do I get here" rather than "where

do I go from here." Skip Carter gave a short talk about "Subsumption Machines" the month after I first presented this code, showing that my machine fits Brooks' model. I apologize for not being able to repeat his talk, but the generalization incorporates both an input-event hierarchy and the notion of time as an input.

John has been an apprentice Forth programmer since 1976, when he traded his Community College Data Processing Director's job for a microprocessor. In 1986, he helped design the processor Harris sold as the RTX2000, and has recently designed several more small, "Forth friendly," CPUs. He served as editor for both the ANS Forth and IEEE Open Firmware standards. The rest of his life involves children, community, gardening, and science fiction.

Listing One.

```
\ EXAMPLE.4TH  An example using the simple state machine lexicon
\ ================================================================================
\ This is an ANS Forth Program with Environmental Dependencies on:
\  KEY? from the Facilities wordset
\  MS   from the Facilities extension wordset
\  all the words included in the file MACHINE.4TH shown in Listing 2.
\ A Standard System exists after this program is loaded.
\ Any operator's terminal facilities provided by the system are adequate.
\ ================================================================================

\ This demo uses the clock-tick as an event, displaying a different digit
\  and number of spaces from each state.  The 'next-state' and 'output'
\  definitions could be defined with :NONAME to conserve name space.

\  define machine and state names ----------------------------------------

MACHINE: DEMO
   STATE: ONCE=      \ Stay in this state just one tick, then to TWICE=
   STATE: THRICE=    \ Stay in this state for three ticks, then to ONCE=
   STATE: TWICE=     \ Stay in this state for two ticks, then to THRICE=
MACHINE;

\  next-state logic equations, stack must be ( - flag ) --------------------

: >TWICE?  ( - is-next? )  TICK# 1 >     ONCE= AND ;
: >THRICE? ( - is-next? )  TICK# 2 >    TWICE= AND ;
: >ONCE?   ( - is-next? )  TICK# 3 >   THRICE= AND  RESET= OR ;

\  state output equations, stack must be ( - ) ----------------------------

: ONCE.   ( - ) TICK# IF SPACE  EXIT  THEN ." 1" ;      \ "1 "
: TWICE.  ( - ) TICK# IF SPACE  EXIT  THEN ." 2" ;      \ "2  "
: THRICE. ( - ) TICK# IF SPACE  EXIT  THEN ." 3" ;      \ "3   "

\  clock-period word, stack must be ( +n - flag ) -------------------------

: 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  250 PERIOD !   \ The period is 1/4 second in this demo

: TICK-TOCK? ( #ticks - exit-machine? )  PERIOD @ MS  48 >  DONE? OR
; \ The demo will end after 50 ticks

\  make machine and state assignments -------------------------------------

\  clock-tick    <zero>     machine
  ' TICK-TOCK?     0      ' DEMO      ASSIGN
```

```
\  enter-state   output      state
  ' >THRICE?    ' THRICE.   ' THRICE=   ASSIGN
  ' >TWICE?     ' TWICE.    ' TWICE=    ASSIGN
  ' >ONCE?      ' ONCE.     ' ONCE=     ASSIGN

\ run the state machine ---------------------------------------------------

\ DEMO          \ make it the current machine (only needed if more than one)
\ RUN           \ should show repeating pattern of "1 2  3   1 2  3   "
\ RESET         \ "power-off" the current machine: the state becomes zero
\ n PERIOD !    \ changes the repeat rate
\ S.            \ process one clock tick and display the internal variables
\ DEBUG         \ RUN with MACHINE. display after each tick
```

Listing Two.

```
\ MACHINE.4TH  A simple state machine lexicon
\ ============================================================================
\ This is an ANS Forth Program
\ A Standard System exists after this program is loaded.
\ Any operator's terminal facilities provided by the system are adequate.
\ ============================================================================

\ The machines that are defined by this lexicon are more general than what
\  are generally known as state machines. This was pointed out to me by Skip
\  Carter at a SVFIG meeting when I first showed this code. This code is not
\  very fast, however, since it is modelling in software (serially) what is
\  done in hardware (in parallel).

\ Only the uppercase words are needed to use this lexicon, the lowercase
\  ones are just used locally. A number of error checks are included, so not
\  all states need be ASSIGNed, or even named, before testing.

\ Defining Words ----------------------------------------------------------

VARIABLE current-machine ( - ptr.machine )

: MACHINE: ( "machine-name" - link )
   CREATE  HERE current-machine !
   HERE  0 ( link ) , 0 ( ticks ) , 0 ( current.state ) , 0 ( tick.xt ) ,
DOES> ( dfa - )
   current-machine !
;

: machine       ( - adr.link )   current-machine @ ;
: ticks         ( - ptr.state )  CELL+ ;
: current-state ( - adr.ticks )  machine CELL+ CELL+ ;
: do-tick       ( - exit-machine? )
   machine DUP IF
      CELL+ CELL+ CELL+ @ DUP IF machine ticks @ SWAP EXECUTE 0= THEN
   THEN 0=
;

: STATE: ( link "state-name" - link' )
   CREATE  HERE SWAP !  \ links this state to machine or previous state
   HERE  0 ( link ) , 0 ( ticks ) , 0 ( output.xt ) , 0 ( logic.xt ) ,
DOES> ( dfa - current-state? )
```

```
   current-state @ =
;

: do-output ( state - )           CELL+ CELL+ @ ?DUP IF EXECUTE THEN ;
: is-next?  ( - is-next? )
   DUP IF CELL+ CELL+ CELL+ @ DUP IF EXECUTE THEN THEN
;

: MACHINE;  ( link - )   DROP ; \ an alias that makes prettier syntax

: ASSIGN ( 'clock|'logic 0|'output 'machine|'state - )
   >BODY CELL+ CELL+ 2!
;

\ State Machine execution words ----------------------------------------

: RESET ( - )  0 machine ticks !  0 current-state ! ;

: RESET= ( - reset? )  current-state @ 0= ;

: first-state  ( - state )  machine @ ;
: checked-all? ( state|0 - state' FALSE | TRUE )   DUP IF @ ?DUP THEN 0= ;

: STEP ( - exit-machine? )
               current-state @  first-state
  ( cur link ) BEGIN  DUP is-next? 0= WHILE
               checked-all? UNTIL  \ nothing selected, so stay in same state
       ( cur )    DUP 0= IF  0= ( TRUE ) EXIT  THEN      \ unless it's reset
      ( next ) ELSE                \ there's a next state selected
  ( cur next )    SWAP OVER XOR IF \ it's different from the current state,
      ( next )       DUP current-state !           \ so make it current
                     machine ticks @ OVER ticks !  \ and initialize TICK#
                  THEN
               THEN
               do-output      \ done on each tick, not just entering state
          (  ) do-tick
     ( exit? ) 1 machine ticks +!
;

: TICK# ( - n ) machine ticks @  current-state @ DUP IF ticks @ THEN - ;

: RUN ( - )    BEGIN  STEP  UNTIL ;

\ Debugging words ------------------------------------------------------

: ?+ ( adr - adr+ ) DUP @ 8 .R CELL+ ;

: MACHINE. ( - )
               machine
       ( link ) BEGIN
                  CR DUP ?+ ?+ ?+ ?+ DROP \ show 4 machine or state values
               checked-all? UNTIL
;

: S. ( - )  STEP DROP MACHINE. ;                  \ single-step and display

: DEBUG ( - )  MACHINE. BEGIN CR STEP MACHINE. UNTIL ;
```

DOS-Compatible Disk Access for Any Target

Dwight Elvey
Santa Cruz, California

Using a PC as a development environment works fine, but leaves us with a recurring problem. Every time we change and recompile our application, we must transfer the new code to the target. Burning a PROM is one avenue for shuttling new versions of the code to the target. A more practical way is possible when the PC host and the target both support a serial connection.

A serial communications link does get the job done, but I have found a better solution by adding a minimal disk I/O subsystem to the target. Because this system conforms to a PC, in terms of the file structures written to and read from the disk, the host PC can continue to serve as the development system that generates the code. By moving the disk physically to the target system, the code becomes accessible to the target. A considerable added benefit of this approach is that the target can support intensive data logging functions, as long as its increased power and space needs are not prohibitive.

Why would one want to read and write a DOS disk from a platform other than an IBM-compatible PC? There are many reasons. One might be to transfer logged data from an embedded system to a PC in a format that can be stored or used when needed. My reason was to make a backup of the code I had written on a hobby project system, in case I messed up and didn't have a working PROM to recover from a mistake.

In this article, I describe DOS disk-file structures. Obtaining a DOS-compatible data format for the files written to disk turns out to be the bigger part of the solution, and required the most research. A follow-up installment will be about using a PC-type floppy controller on a non-80x86 embedded system. Although my system used 360K drives and an XT controller card, most of the information can be used for 1.2 Mb or 1.44 Mb drives and controllers.

Like many home projects, this one grew as I went along. It started with an NC4000 board from Silicon Composers, Inc. The board came with a cmForth in ROM and the source code. At first, all the code to run in RAM was downloaded through the RS-232 serial interface. I would write all my code on a PC and send it serially to the card. Gradually, I came to see the value of having a local disk attached to my single-board computer (called single board, not because it had just a single PC-board, but because it was mounted on a single plywood board).

First, I found, in my junk box, an old floppy controller for a SYM-2 board (SDK for Synertek 6502). I wrote a driver for this and used an old floppy disk drive. It was only 160 Kbytes, but it proved the effectiveness of the technique. Next, I found an old ST504 five megabyte hard-disk drive. Adding a PC ST-11 MFM interface card provided mass storage. Now I had, almost, a complete standalone development system. Adding a simple PROM programmer completed the system. However, it left me longing for a DOS-compatible drive to achieve interoperability with my PC. Also, I had worked myself into a problem: if I made a mistake in my new code that affected the ability to read/write disks or blow PROMs, I might not have a way of recovering the system.

I needed a backup of my target code that didn't require a working NC4000 system. I could have gone back to the serial interface but, wanting to enhance my system, I decided to go with a 360K DOS floppy system. (This disk format was chosen because I had several 360K drives from AT systems that were upgraded to 1.2M and getting an XT interface was easy.) Once the low-level controller code was working, I would write a DOS-style file-read and file-write command-line interface for the target. I would then have a DOS-compatible system that would allow me to save source code or binary images of the code in PROM as DOS files.

To keep the target code (image size) to a minimum, the low- and high-level routines would only support fundamental DOS-file-system-compatible read and write operations. For example, I offer no code to format the disk. The host PC can do any required formatting, on a 360K drive. One has to be careful when using 1.2 Mb drive with 360K disk. The 1.2 Mb drive writes a narrower track than the standard 360K drive.

In order to understand how the code conforms to PC standards, we need to understand how a DOS disk is organized. There are four sections to a DOS disk. First is the boot sector, then FATs, then the root directory and, finally, the files themselves.

The boot sector is always on the first track and sector of the disk. Even if the disk is not bootable, this sector must contain some executable code (the code that prints on the screen that it isn't a bootable disk). My NC4000 system ignores this code, but it is still required for the DOS

machine. This sector also contains information about the way the disk is formatted, any relevant information about revisions, number of FATs, etc. Since my system always uses 360K disks, I have chosen to ignore this also. If the disk system were intended to read different-sized disks, one would have to pay more attention to the data in this sector.

The FATs are in the next sectors of the disk. FAT stands for File Allocation Table, which is a map showing how the disk is used. It is a linked list that corresponds to the clusters on the disk. A cluster is the minimum allocatable disk space. In the case of a 360K disk, each cluster is two sectors, or 1024 bytes. The size of a cluster may vary for different-sized floppy or hard disks. There are two identical FATs on a standard 360K disk. This is so that, if one FAT is damaged, the user may be able to recover the disk's structure (wishful thinking). FAT entries are 12 bits each on a 360K disk—more about that later. Each entry's location in the table corresponds to a cluster on the disk. The table location will contain a link to the next cluster used by the file using this cluster, an end-of-cluster marker, an indication that the sector is unused, or the sector-is-bad indicator. Using this system, a file need not have consecutive clusters and, as long as the links are okay, one can read or write to the file.

360K disks use a 12-bit FAT entry. The newer 1.2 Mb and 1.44 Mb disks use a 16-bit entry. The 16-bit entry is easy because it is the typical byte-reversed Intel-format number. The 12-bit is more of a problem. Every three bytes in the FAT correspond to two FAT entries. If I use the nomenclature A0-A11 for the first entry and B0-B11 for the second entry, the first two FAT entries would look as follows:

FAT Bit Order for 12-bit FATs

Bits:	7	6	5	4	3	2	1	0
First Byte:	A7	A6	A5	A4	A3	A2	A1	A0
Second Byte:	B3	B2	B1	B0	A11	A10	A9	A8
Third Byte:	B11	B10	B9	B8	B7	B6	B5	B4

Other than the fact that this is messy, once one has the code to select an entry and the code to read and write an entry, one doesn't really have to worry much about this.

The next part of the disk is the root directory. This contains information about the filename, size of the file, attributes of the file and, most importantly, the first cluster number. This cluster number can then be used to index into the FAT and follow the chain of links through the entire file. The first directory may have links to other sub-directories on the disk. My implementation assumes there are no sub-directories, but this can be added easily: The easiest way to do this is to make a sub-directory with the PC and use the code I've supplied to look at how the root directory, sub-directory, and FAT are changed. Things usually make more sense when one experiments with them.

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.

The root directory is made up of seven sectors on a 360K disk. Each entry is a fixed-length record of 32 bytes. Each record entry is as follows:

Byte offset	Record Field
0 to 7	Filename
8 to 10	Filename Extension
11	File Attributes
12 to 21	Reserved
22 to 23	File Creation/Change Time
24 to 25	File Creation/Change Date
26 to 27	First Cluster Number
28 to 31	File Size in Bytes

In my read/write code, I ignore everything except the filename/extension, first cluster, and file size. One could add code to change time/dates and also checks to see if the file had attributes such as read-only. The time/date information would be especially useful if the file was used for data logging. (In any case, my system didn't have a real-time clock until I later added one of the smart-clock PROM sockets from Dallas Semi.)

Last comes the file data. The file always starts with the first cluster pointed to by the directory entry and follows the clusters allocated in the FAT. If the file doesn't use all of the last cluster, the unused space is wasted. Since I'll be taking 1024-byte blocks from my disk structures on the NC4000 system, I'll be using whole clusters for each `BLOCK`.

Overview of the Code

The NC4000 is a 16-bit machine. Since it is inconvenient to work in bytes, I have words that take two bytes and convert them into two leading, padded, 16-bit values. For the blocks used in cmForth, Charles Moore used 40 as the leading byte for reasons that never made clear sense to me, but I continued to use this. These pack and unpack words are `SMASH` and `UNSMASH`.

Next are some basic words to get the main parts of the DOS disk into memory buffers. Since I'll need these buffers a lot, I've made them separate from the `BLOCK` buffers used by the NC4000's disk I/O. The words `RDFAT`, `RDDIR`, `FAT@`, and `FAT!` should make sense. Words like `DIRFIND`, `CREATEFILE`, and `OPEN` should be okay, also.

`FTYPE` is used as a sanity check by displaying ASCII text files. `BLK>FILE` is the heart of what this is all about. This moves specified `BLOCK`s from the NC4000 file structure to the `OPEN`ed file on the DOS disk.

Well, that's it. All may use the code as they see fit—just don't blame me if it mangles your disk. In order to really understand all, it helps to have some books as reference. Here is a list of books I've found to be useful:

The Indispensable PC Hardware Book by Hans-Peter Messmer. Addison-Wesley (ISBN 0-201-87697-3).

The Programmer's PC SourceBook by Thom Hogan. Microsoft Press (ISBN 1-55615-321-X).

PC System Programming by Michael Tischer. Abacus (ISBN 1-55755-036-0).

Code begins on next page...

cmForth and Special NC4000 Words Used

`@+ ( Addr Incr - Value Addr+Incr )`
Machine coded fetch and increment.

`!+ ( Value Addr Incr - Addr+Incr )`
Machine coded store and increment.

`FOR ( n - )`
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. This 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 a 16-bit value.

`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 from R 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

```
DECIMAL
\ The NC4000 is a 16-bit, word-addressed machine. Because of this and the fact
\  that I wanted to use two bytes to an address space, I made a few words to assist.

: 2C@+ ( Addr - Addr+1 Low High )   \ two bytes from word location
  1 @+                    \ Fetch 16 bits and incr pointer
  SWAP DUP 255 AND        \ Low part
  SWAP BSWAP 255 AND ;    \ Hi part

HEX
\ SMASH and UNSMASH are like MOVE and CMOVE combined.

: SMASH ( From To - ) ( 512 Bytes )    \ Packs 2 bytes into a word
  2* 1FF FOR      \ Byte addresses are twice
     >R
     1 @+          \ Fetch from and incr
     SWAP 0FF AND \ Low byte
     I C!          \ Store in to
     R> 1 +        \ Incr to
  NEXT DROP DROP ;

: UNSMASH ( From To - ) ( 512 Bytes )   \ UnPacks a word to two bytes
  0FF FOR
     >R                \ Easier with one out of the way
     2C@+ 4000 OR I !  \ Same format as for disk
     4000 OR           \  used in CMForth: 40xxH
     R> 1 + 1 !+       \ Incr and store
  NEXT DROP DROP ;

: DOS0  FLUSH DR0 ; \ always use DR0 for DOS disk

CREATE FAT 200 ALLOT  ( 1024 BYTES ) \ Storage for a FAT from disk

\ FAT ( file allocation table ) contains linked list that describes the usage of the disk.
\  A DOS disk has two copies in the hopes that only one might be damaged, allowing
\  recovery of the data on the disk. On a 360K disk, the first FAT is at sector 3
\  ( zero-based ). The second is at sector 5. Reading BLOCK one gets part of each,
\  but they are out of order, hence the fun offsets below.
```

```
: RDFAT ( - ) ( read FAT )
   \ Easier to read last part of first FAT and first part
   \  of second with block = 1K
   DOS0 1 BLOCK                    \ used block I/O rather than direct read
   DUP 200 + FAT SMASH             \ do first 512 bytes from 2nd FAT
   [ FAT 100 + ] LITERAL SMASH ; \ do next 512 bytes from 1st FAT

CREATE DRCT    100 7 * ALLOT  \ Space for the root directory

\ I only provide usage of the root directory. Sub-directories could be easily added.

: RDDIR ( read DIR )
  \ Transfer a copy of the root directory to memory
     DOS0
     2 BLOCK 200 + DRCT SMASH
     3 BLOCK DUP DRCT 100 + SMASH
           200 + DRCT 200 + SMASH
     4 BLOCK DUP DRCT 300 + SMASH
           200 + DRCT 400 + SMASH
     5 BLOCK DUP DRCT 500 + SMASH
           200 + DRCT 500 + SMASH ;

  HEX
\ The FAT points to a cluster of sectors. On the 360K disk a cluster is 2 sectors. To make
\  things tougher, the entries in the 360K FAT are two 12-bit values packed into 3 bytes.
\  This requires a fancy fetch. The value fetched can be one of several values. 0 = unused,
\  0FF8 to 0FFF is an end marker, 0FF7 is a bad cluster, 0FF0 to 0FF6 are reserved, and
\  0002 thru 355 decimal are pointers to the next cluster.

: FAT@ ( Cluster - Cluster' )
   \ Given a cluster number, fetch the next link with a 12-bit FAT
    2/MOD 3 *       \ Calc bytes offset
    FAT 2* +        \ Make byte address in FAT
    >R IF           \ If odd cluster
      1 I + C@        \ Fetch bits 0-3
      2 TIMES 2/      \ Shift right by 4  ( 2/ is code word )
      R> 2 + C@       \ Fetch bits 4-11
      2 TIMES 2* OR   \ Shift left by 4 and combine ( 2* is code word )
    ELSE            \ If even cluster
      I C@            \ Bits 0-7
      R> 1 + C@       \ Bits 8-11 but in position 4-7
      BSWAP OR        \ So swap and combine
    THEN
    0FFF AND ;       \ Mask off unwanted part

: C@+ ( A - V A' )      DUP C@ SWAP 1 + ;

: CLSTR/MOD ( Low High - Rem Clstr# )
\  32-bit file size to cluster and part
    BSWAP 4 TIMES 2* >R  \ Shift left by 6
    BSWAP DUP 3FF AND    \ Remaining amount
    SWAP 8 TIMES 2/      \ Shift right by 10
    3F AND R> + ;        \ Make cluster number

    0E0 1 - CONSTANT #DIRS  \ number of directory entries less one

: DIR    \ display directory
  \ Name Ext Clusters RemBytes FisrtCluster Attribute
  \
  \ One could add printouts. I ignored the attributes and date since I had no real-time clock
  \  when I wrote this and I wasn't checking attributes. This assumes that the directory has
  \  already been read into RAM with RDDIR. If one wanted to, they could add RDDIR but one
  \  would have to be careful to write out any updates before using it or have a
  \  directory-updated flag.
     DRCT 2*          \ Byte address
     #DIRS FOR        \ 112 directory entries for 360K disk
      DUP C@ DUP 0E5 = NOT AND  \ 0 = empty, 0E5 = erased
```

```
    IF
      CR 7 FOR
        C@+ SWAP EMIT  \ Filename 8 characters
      NEXT SPACE
      2 FOR
        C@+ SWAP EMIT  \ Extension 3 characters
      NEXT
      C@+          \ Attribute
      0E + 2/      \ Back to 16-bit addressing
      1 @+         \ Start Cluster
      1 @+         \ Low word file length
      1 @+ >R      \ High word file length
      BASE @ >R HEX            \ Like my numbers in hex
      CLSTR/MOD 4 U.R  4 U.R   \ Clusters and bytes
      BSWAP 4 U.R              \ First cluster
      3 U.R                    \ Attributes
      R> BASE !
      R> 2*                    \ Back to byte addressing
    ELSE
      20 +   \ Next entry
    THEN
   NEXT DROP ;

: WRFAT ( - )  \ Write FATs back to disk
 \ Writing needs to build both FATs, so it is a little more work than reading.
   DOS0
   FAT 0 BLOCK             \ Get first 2 clusters
     200 + UNSMASH UPDATE  \ Put first part of first FAT
   1 BLOCK FAT OVER        \ Get next two clusters
     200 + UNSMASH         \ Remainder of first
   FAT 100 + SWAP UNSMASH UPDATE  \ First part of second FAT
   FAT 100 +
   2 BLOCK UNSMASH UPDATE  \ Remainder of second FAT
   FLUSH ;

CREATE CLUSTER 200 ALLOT ( 1024 Bytes ) \ Cluster buffer

: DOSBLOCK ( CLSTR# - ADDR )  \ Needs offset of 4K
     4 + BLOCK ;

: RDCLUSTER ( Clstr# - ) \ Read cluster from disk to CLUSTER buffer
   DOS0
   DOSBLOCK                    \ Read disk, one cluster 1K
   DUP CLUSTER SMASH           \ First half
   200 + CLUSTER 100 + SMASH ; \ Second half

: WRCLUSTER ( Clstr# - ) \ Complement of read is write
   DOS0
   DOSBLOCK CLUSTER OVER UNSMASH    \ First half
   CLUSTER 100 + SWAP 200 + UNSMASH \ Second half
   UPDATE FLUSH ;

: FAT! ( CLSTR# CLSTRADDR - )
\ Set FAT entry to point to cluster address. Notice the
\  similarity to FAT@. This is used to make the linked list of clusters.
   2/MOD 3 * 1 +      \ Get byte offset
   FAT 2* + >R        \ Make a byte address and save.
   IF                 \ Odd entry
     DUP 2 TIMES 2*   \ Left shift by 4
     0F0 AND          \ Mask unwanted parts
     I C@             \ Get what was in FAT to combine
     0F AND OR        \ Make one byte
     I C!             \ Save it back to FAT
     2 TIMES 2/       \ Shift right by 4 to do remaining 8 bits
     0FF AND          \ Mask any extra
```

```
    R> 1 + C!        \ Save it into FAT
   ELSE              \ Even entry
    DUP 0FF AND      \ Mask 8 bits of 12
    I 1 - C!         \ Store in FAT
    BSWAP 0F AND     \ Fast shift right by 8 with mask
    I C@             \ Fetch other 4 bits
    0F0 AND OR       \ Combine together
    R> C!            \ Save to FAT
   THEN ;

: FATEND ( CLSTR# - LSTCLSTR# )
\ Follow file clusters to end of file
   BEGIN
    DUP FAT@     \ Look at each cluster pointer
    DUP 0FF8 AND \ 0FF8 thru 0FFF
    0FF8 -
   WHILE         \ While not end mark
    SWAP DROP    \ Follow chain of clusters
   REPEAT DROP ;

: FATAVAIL ( - FirstAlailableFat )
\ Used to find a free FAT entry
  1             \ First two are always used
  BEGIN
    1 +         \ Next FAT
    DUP FAT@ 0= \ Unused?
  UNTIL ;

CREATE FILE0 0C ALLOT    \ First file handle
CREATE FILE1 0C ALLOT    \ Second file handle
CREATE CRNTFILE  FILE0 , \ Current file opened ( default is FILE0 )
\ Changing the current file handle with   FILE1 CRNTFILE !

: FLPART ( offset - | - addr )
\ Makes handle structure
    CREATE            \ Make name
     ,                \ Save offset value
    DOES R> 7FFF AND  \ Get parm field
     @ CRNTFILE @ + ; \ Make data field address

\ Following is the file handle data structure
 00 FLPART NAME      \ Filename
 05 FLPART EXT       \ Filename extension
 07 FLPART 0CLSTR    \ First cluster
 08 FLPART CLSTRS    \ Clusters per file
 09 FLPART REMBYTS   \ Bytes in last cluster
 0A FLPART LSTCLSTR  \ Last cluster
 0B FLPART DIR#      \ Directory offset count

: ?NAME ( - )
\ Get filename. Sorry this is a long one, but it had a lot to do. Parses
\  extension out and puts strings into the current file structure defined above.
   NAME 7 2020 FILL  \ BL's to name and extension
   BL WORD           \ Get string from input
   NAME 2* 0 OVER C! \ First byte set to 0 count
   0A + 0 SWAP C!    \ Same for extension
  2* 1 +
  7 FOR         \ Start looking for "." or BL
    DUP C@ DUP
    20 =
    SWAP 2E = OR
    IF              \ If either
      R> DROP 0 >R  \ Do a leave of FOR NEXT
    ELSE            \ Else put string into NAME
      DUP C@
      NAME 2*
```

```
      DUP C@          \ Fetch count
      1 + OVER C!     \ Increment count
      8 + I - C!      \ Store character ( notice trick to index backwards )
      1 +             \ Increment string pointer
    THEN
  NEXT
  DUP C@ 2E =         \ If "." then more for extension
  IF
   1 +
   2 FOR              \ Up to 3 bytes
     DUP C@ BL =      \ Look for end
     IF
       R> DROP 0 >R   \ If end of string leave loop
     ELSE
       DUP C@
       EXT 2*
       DUP C@ 1+ OVER C! \ Increment string count
       3 + I - C! 1 +    \ Save character
     THEN
   NEXT
  THEN DROP ;

\ Get a particular files part from directory table

: DIRENT ( Dir# Index - cAddr )
\ used to calc location of particular directory entry
   >R 3 TIMES 2*                  \ Shift by 5 or 020H times
   [ DRCT 2 * ] LITERAL + R> + ; \ Returns byte address

: DNAME ( Dir# - cAddr )  0 DIRENT ;  \ Name address
: DEXT ( Dir# - cAddr ) 8 DIRENT ;    \ Extension address
: DATT ( Dir# - cAddr ) 0B DIRENT ;   \ Attribute address
: DSTART ( Dir# - cAddr ) 1A DIRENT ;  \ Sector start address
: DSIZE ( Dir# - cAddr ) 1C DIRENT ;   \ File size address
: DSTART@ ( Dir# - Start ) DSTART 2/ @ BSWAP ; \ Cluster start
: DSIZE@ ( Dir# - #Clstr Rem )
   DSIZE 2/ 1 @+ @ CLSTR/MOD ;   \ Clusters and remainder bytes.
: DTIME ( Dir# - cAddr ) 16 DIRENT ; \ Time string
: DDATE ( Dir# - cAddr ) 18 DIRENT ; \ Date string

: $COMP ( cAddr1$ cAddr2$ Cnt - Flag )
\ String compare to match filenames with input strings: true = same    false = different
  DUP
  IF
    -1 >R           \ Flag on return stack
    1 - FOR         \ Number of bytes to compare.
      C@+
      SWAP ROT C@+
      SWAP ROT -    \ fetch and compare bytes ( faster than = on NC4000 )
      IF            \ Not the same?
        R> DROP     \ Drop loop count
        R> DROP     \ Drop flag
       0 >R  0 >R  \  and leave loop on NEXT
      THEN
    NEXT DROP DROP \ Discard string addresses
    R>              \ Return flag
  ELSE              \ no length so false
    DROP DROP DROP 0
  THEN ;

: DIRFIND ( - -1=no 0-#DIRS=yes )
\ Find a name in the DOS directory. Assumes a directory was read
\  by RDDIR with something like OPEN
  ?NAME
   -1
  #DIRS FOR             \ Look at one at a time
```

```
    NAME 2* 1 +          \ Counted string fetched by ?NAME
    #DIRS I - DNAME      \ Start from the bottom of the directory
    8 $COMP              \ Compare all 8 bytes
    IF
      EXT 2* 1 +    \ Extension fetched by ?NAME
      #DIRS I - DEXT    \ Again from bottom of directory
      3 $COMP           \ If both then done
      IF
       DROP #DIRS R> - \ Calc directory entry #
       0 >R            \ Set NEXT count to zero for leave
      THEN
    THEN
  NEXT ;

: OPEN  ( - | Filename )
\ open specified filename
   RDDIR       \ Read directory
   RDFAT       \ Read FAT
   DIRFIND     \ Find file
   DUP 1 +     \ Found?
   IF      \ found so read handle info
     DUP DSTART@ 0CLSTR !  \ First cluster
     DUP DSIZE@ CLSTRS !   \ Number of full clusters
     REMBYTS !             \ Remaining bytes in last cluster
     0CLSTR @ DUP          \ More than 0 bytes?
     IF
      FATEND               \ Then scan for last
     THEN
     LSTCLSTR !            \ Save last
     DIR# !                \ Save directory number
   ELSE    \ not found
     DROP ABORT" No file? "
   THEN ;

HEX
VARIABLE LINES  0 LINES !
: CR+   ( - ) \ Pause every 16 lines and check for quit by ESC key
   LINES @ 15 >
   IF KEY 1B =
     IF QUIT THEN
     0 LINES !
   ELSE
     1 LINES +!
   THEN CR ;

: EMITTEXT ( ADDR CNT - )
\ Simple text typer with pause
   FOR
     1 @+ SWAP DUP 400D =  \ Note: 4000 part of cmForth
     IF DROP CR+           \ Needs to look for CR
     ELSE DUP 400A =       \ Ignores LF's
       IF DROP
       ELSE EMIT           \ Emits characters
       THEN
     THEN
   NEXT DROP ;

: FTYPE ( - )
\ Type a specified file as text
  OPEN       \ Open file
  CR+        \ Do cr
  CLSTRS @ ?DUP  \ Check number of clusters
  IF
    0CLSTR @ SWAP 1 -
    FOR             \ For each cluster
```

```
      DUP DOSBLOCK    \ Read cluster
      3FF EMITTEXT    \  and type it
      FAT@            \ Get next cluster
    NEXT
  ELSE
    0CLSTR @          \ Just the first one
  THEN
    REMBYTS @         \ Bytes to type
    ?DUP
    IF
      SWAP DOSBLOCK SWAP 1 -  \ Type only bytes of file
      EMITTEXT                \  that remain
   THEN
   DROP
   LINES ! EMPTY-BUFFERS ;

DECIMAL
: CMOVE ( cFrom cTo Cnt - ) \ not the fastest, but works
   1 -
   FOR
     OVER I + C@
     OVER I + C!
   NEXT DROP DROP ;

: ADD-SIZE ( Bytes - )
\ Adds amount of bytes to file length to open file
  0CLSTR @ 0=
  IF          \ Check for 0 length file
    FATAVAIL DUP 0CLSTR ! \ Assign a cluster
    LSTCLSTR !            \ Also last
  THEN
  REMBYTS @ +     \ Add to remaining
  1024 /MOD ?DUP  \ Needs more clusters?
  IF 1 -
    FOR         \ Assign each cluster
      FATAVAIL
      DUP LSTCLSTR @ FAT!    \ Update FAT
      LSTCLSTR ! 1 CLSTRS +! \ Update end
    NEXT
    -1 LSTCLSTR @ FAT!       \ Mark last in FAT
  THEN
    REMBYTS ! ;              \ Update remaining bytes

( : SUB-SIZE ( BYTES - )
\ Reduce file size. Never checked out!
( REMBYTS @ SWAP - DUP 0<                          )
( IF NEGATE 400 /MOD ?DUP                          )
(  IF 1 - FOR CLSTRS @ IF 0CLSTR @ 0 OVER FAT! >R    )
(   0CLSTR @ BEGIN DUP FAT@ I - WHILE FAT@ REPEAT R>DROP )
(   LSTCLSTR ! -1 CLSTRS ! THEN NEXT 1024 SWAP - 1023 AND )
( THEN REMBYTS ! ;      )

: DIRAVAIL ( - Dir# )
\ Scan for a free directory entry to use. Doesn't check to see if none is
\  available. For safety, needs check added!
  0
  BEGIN
    DUP DNAME C@
    DUP E5 = NOT AND  \ E5 or 0 indicates free
  WHILE
    1 +
  REPEAT ;
```

```
HEX
: UPDATE-DIR ( - )
\ Puts new file directory info into directory structure. Doesn't do date and time. If you
\  have real-time clock, it would be nice to add but isn't truely needed for most things.
  DIR# @
  DUP >R DNAME 2/     \ Word address
  DUP 6 2020 FILL     \ File with blanks initially
  6 + 0A 0 FILL       \ Other inits
  NAME 2* DUP C@
  SWAP 1 + I DNAME
  ROT CMOVE           \ Put name into dir structure
  EXT 2* DUP C@
  SWAP 1 + I DEXT
  ROT CMOVE          \ Put extension into dir structure
  0CLSTR @ BSWAP I DSTART 2/ ! \ First cluster
  CLSTRS @ DUP 400 * REMBYTS @ + \ Any remaining
  BSWAP R> DSIZE 2/ 1 !+          \ Create length
  SWAP 40 / BSWAP SWAP ! ;        \  and store it

: WRDIR# ( Dir# - )
  DOS0
  10 / DUP 100 * DRCT +
  SWAP 5 + 2/MOD       \ Calculates the block containing Dir#
  BLOCK SWAP 200 * +
  UNSMASH              \ Transfers from RAM image to block
  UPDATE FLUSH ;       \ Writes back to disk

: CREATEFILE ( - | Filename )
\ Used to create a file
  RDDIR RDFAT          \ Get disk info
  ?NAME                \ Fetch a name
  DIRAVAIL >R          \ Find directory space
  0 CLSTRS !           \ Init clusters
  0 REMBYTS !          \  and remaining
  I DIR# !             \ What dir entry
  0 0CLSTR !           \ First cluster not assigned yet
  0 LSTCLSTR !         \ Same with last
  UPDATE-DIR           \ Put all this into RAM image of Dir
  R> WRDIR# ;          \ Save it to disk

: BLK>FILE ( Blk# - )
\ Appends block to opened file. Assumes block is on hard disk DR2 and DOS disk is
\  360K floppy DR0.  Also assumes DOS file is a block file.
  FLUSH             \ Empty any offending blocks
  400 ADD-SIZE      \ Increase file by 1024 bytes
  LSTCLSTR @        \ Find the last cluster
  DR2 SWAP BLOCK    \ Fetch from hard disk
  DROP DR0          \ Want to flush to DOS disk
  4 + 10 @ 12 + !   \ Changes block number to be correct for DOS disk. This is very
                    \  implementation-dependent. One could get the DOSBLOCK into
                    \  RAM and do a MOVE but this is faster.
  UPDATE            \ Mark block for return to disk
  UPDATE-DIR        \ Update directory info
  WRFAT             \ Write new FAT
  DIR# @ WRDIR# DR2 ; \ Write new directory info

\ Many other DOS functions could be added. The basic stuff
\  is all here. Things like COPY, RENAME, DELETE, etc.
```

Safety Critical Systems

Paul E. Bennett
Bristol, United Kingdom

A disaster caused by a failure of an engineered system has devastating effects on not only those who are killed or injured or suffer damage to their property but also on the company who operate the Engineered System concerned and those who designed and built it. With the propensity for litigation (almost at the drop of a hat), System Design Engineers, as well as manufacturers and operators, must look to protecting themselves against not only the horrendous levels of claims for compensation to the victims but also the prospect of imprisonment on criminal charges and the resultant destruction of their careers.

Whilst what has been written on Safety Critical Systems in engineering publications has concentrated mainly on the engineering aspects, the engineer is well advised to also pay attention to the legal aspects of his work and apply some risk analysis to his own position.

Introduction

Electronic and computer-based systems are increasingly being used in many application areas where they are likely to have an impact on the overall safety of the workers involved in a process or the public at large. Such systems need to be designed in a manner that gives the necessary levels of assurance that they are safe.

Safety is, in the normal sense of properties, an intangible factor that depends very much on the societal view of what corporations and others do that may affect people individually or as communities. It can be a very difficult issue to resolve, as everything in life carries an element of risk. Whether or not the risk is acceptable depends very much on the benefits a community may gain from accepting the risks involved. Some risks may even be too much to accept.

What are Safety Critical Systems?

Safety Critical Systems are those systems whose malfunction could potentially lead to death or damage. A malfunction in this case is faulty operation, late operation, non-required operation, or non-operation of the system, thus preventing the design intention being carried out. The term covers many system types (see Figure One) and includes pneumatic, hydraulic, electrical, mechanical, electronic, and programmable systems. They can be involved in industrial process plant, medical systems, or consumer goods. They may even be made from bricks and mortar or may be the manufacturing company you work for (see Figure Two).

The above may seem a wide ranging view, but it is applicable throughout all human endeavours. However, for the purposes of this article I shall restrict myself to discussion about programmable electronic systems, with a reminder that the reader considers the wider view for himself.

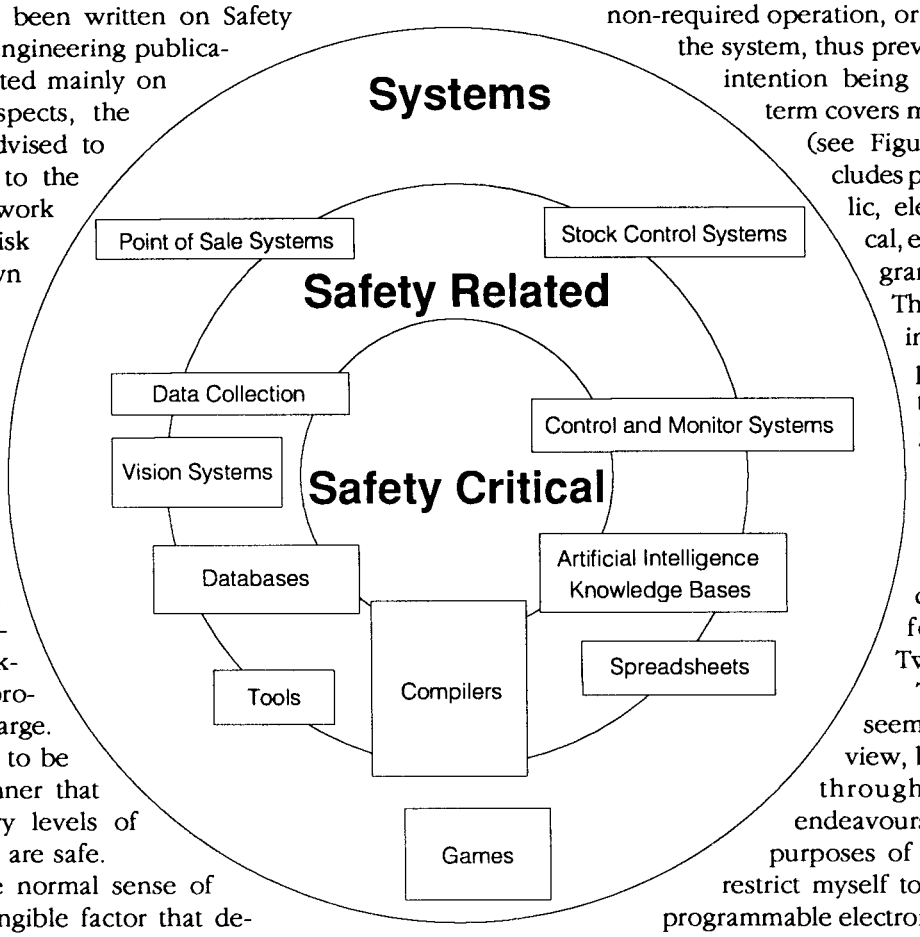
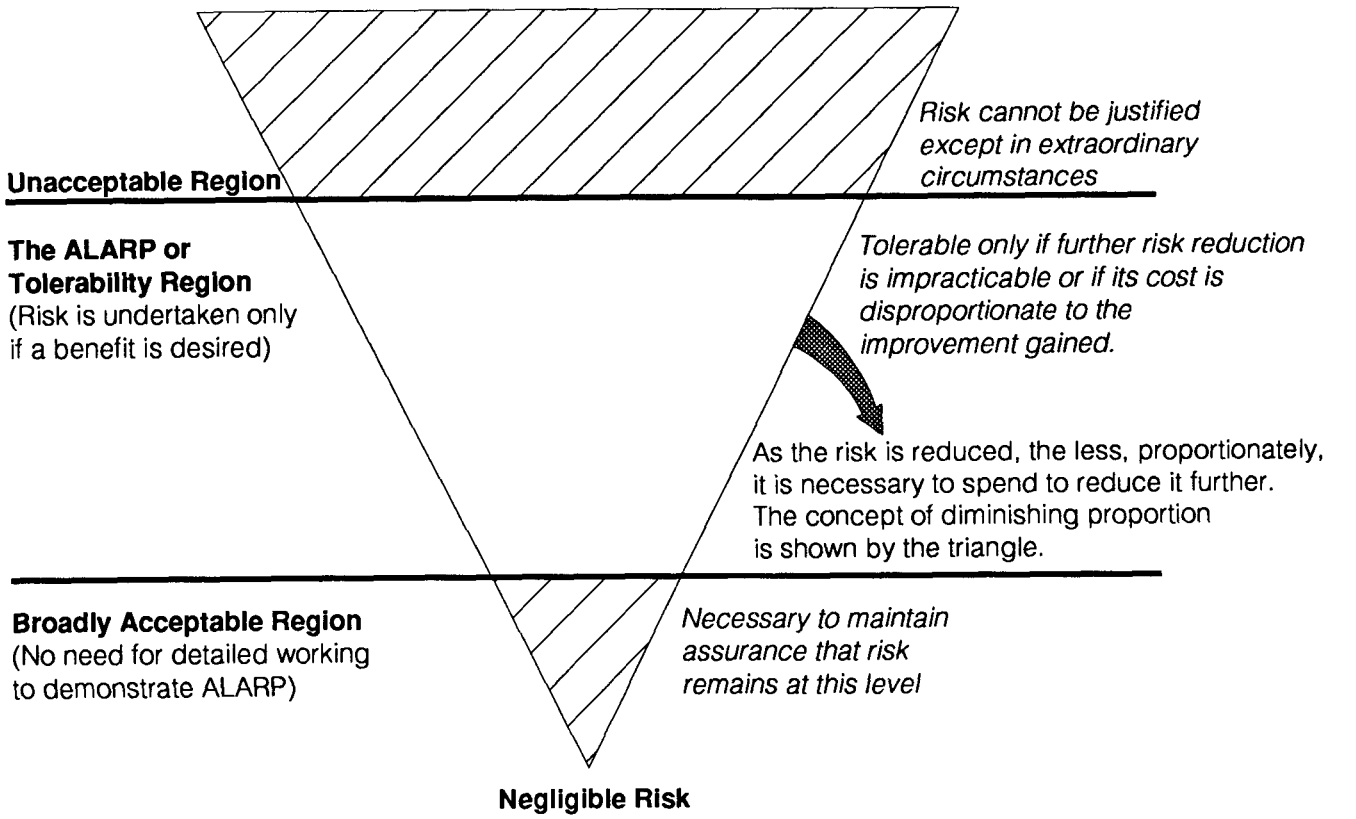


Figure One.
System-type definition.

Figure Two. ALARP Triangle.



Societal View of Hazards and Risk

Society in general accepts risks everyday for situations that do lead to death and destruction. This is evidenced by the existence of wars, car crashes, and other accidents reported in the news on a daily basis. People do get killed, property does get damaged. Some of it is preventable, some not.

Individuals also accept risks—climbing mountains, shooting rapids, leaping from tall buildings, etc. Some people do some of these things just for fun. Others do it as their job. Everyone, however, accepts some element of risk in their daily lives: crossing the road, living in a tall apartment where earthquakes occur, living close to their work at a chemical or nuclear plant.

Individually we would prefer no risk at all. We all expect that society in general is strong enough and caring enough to limit, by means of legislation and monitoring, the risks imposed on us by corporations or other individuals. The level of risk we finally do accept is not zero, nor could it ever be so. Risk, though, can always be reduced at a cost. The environment has become a concern as we have come to realise the aspects of environmental risk we have been exposing ourselves to and wish to reduce. The question is how much are we willing to pay for such a reduction in risk.

Recent research on what people will accept in terms of cost for reduction of risk to save just one life vary depending on which industry we are talking about. For car manufacturers and the road authorities, it seems like a life is worth approximate., £250,000. A similar analysis in the railway industries came up with the figure of £2 million, and in the nuclear industry, £3 million.*

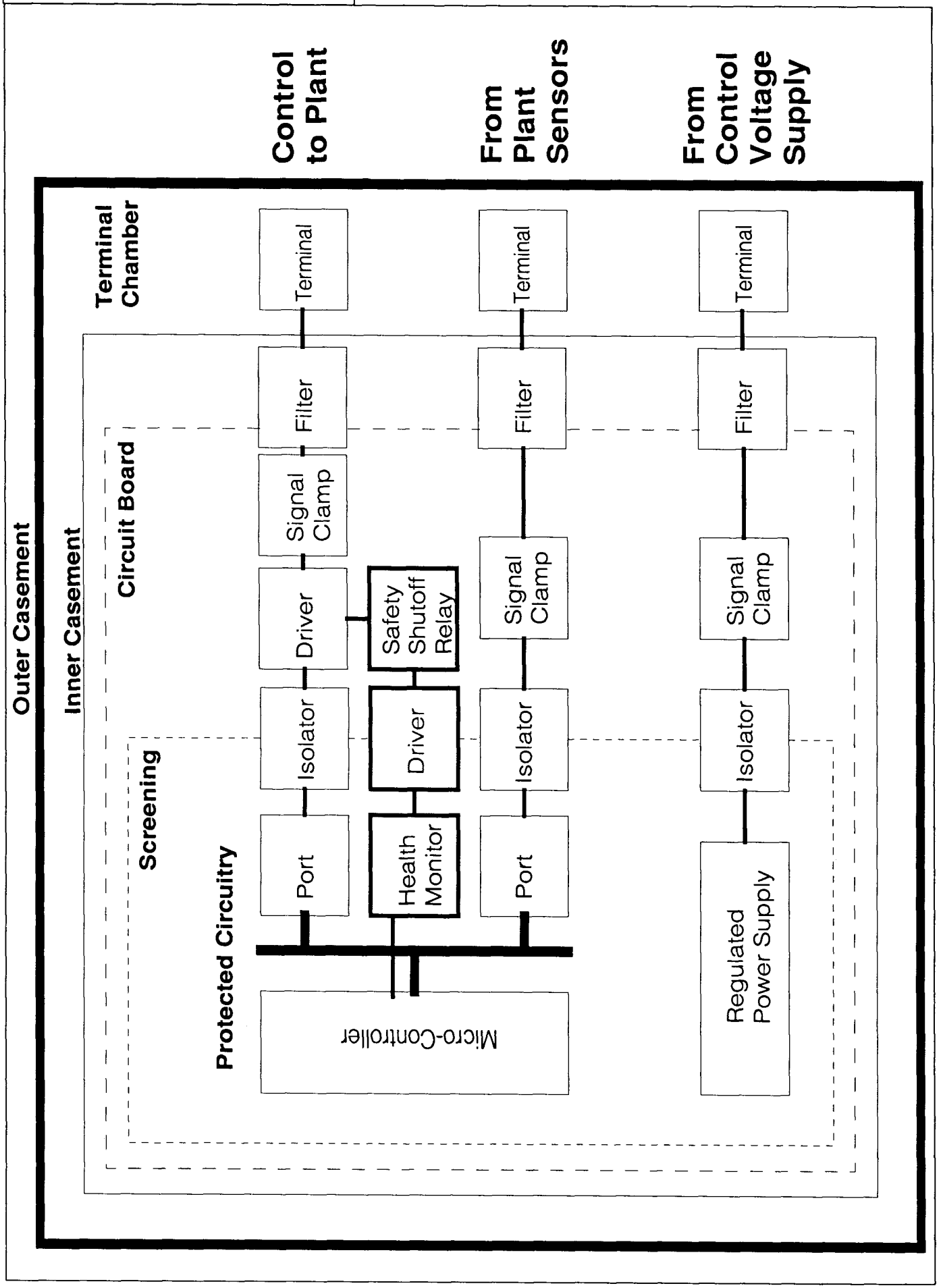
Whilst we can go on spending more and more money to reduce risk, there is a diminishing return on the investment the lower we are trying to drive the level of residual risk (see Figure Two). Therefore, it has been determined within the Safety Critical Systems community that the "Residual Risk Should Be As Low As Reasonably Practicable." Establishing what level of residual risk is acceptable for the application is what the risk analysis stage of the design process is all about.

Designed in Mitigation

In designing a system, it is worth bearing in mind the fact that 33% of rare failures are from small faults, and 33% of the most common failures are caused by very few faults. A common mode failure in something that is designed for the consumer goods market can have wider-ranging devastation than one major fault in a one-off project. Take an electronic control system for a washing machine. A fault in the design of the controller that can cause, in certain circumstances, the overheating of the motor may lead to the appliance catching fire. This fire may lead to death, injury, and damage of property for which, under the UK Consumer Protection Act, you as a designer of a component of that control system may be liable for damages and/

*Figures are from various sources of UK research, averaged out to provide a simple single figure).

Figure Three. Electronic containment.



or imprisonment. You would certainly be liable if it was your component that failed and you were not using the best available practice.

This all seems very daunting for a systems designer or manufacturer to contemplate. For reasons of self protection and preservation, you will need to adopt and work to well-structured design procedures and methods; you will also need to create a clear audit trail for all design work and testing and validation performance.

Inherent Safety

A process that cannot, through any stimulus or environmental deterioration, degrade to a hazardous state is inherently safe. Such systems are preferable from the design position. They require no further protective measures to remain safe. Processes that, if a failure occurs, shut down by themselves to a known safe state are examples of inherently safe systems.

Boundaries and Containment

The provision of well-defined boundaries and the ability to contain hazardous materials or situations within those boundaries is the next best approach to designing an inherently safe system. Such containment or boundaries should be strong enough to resist a breach until the process can be made safe by other means (see Figure Three).

Protective Measures

Protection systems are demand-driven by a failure in the monitored process. They are only active during the existence of a hazardous condition in the process. They must be ultra-reliable and ultra-dependable. Protection systems are usually the most difficult to prove in terms of plant safety, and should only be used when inherent safety and boundaries and containments are not enough to achieve a risk level that is ALARP.

How Good is Good Enough

Not all systems or sub-systems need to be designed to the same Safety Integrity Level. The design of the system components should be appropriate to the Safety Integrity Level required to maintain the assurance of safety for the overall system and evidentially supported in the Safety Case.

Safety Cases

Many of the larger industrial organisations are compelled to comply with rigorous licensing requirements before they may begin operation of new plant, and they have to re-prove their systems at regular intervals. It is the plant operator's responsibility to ensure that the plant he operates is safe. In the design of such large industrial installations, a Safety Case is generated for presentation to a licensing authority.

Safety Cases begin in draft form as very humble documents. Throughout the design lifecycle they grow as changes and adaptations to the design are made and the risks are assessed for the latest situation. Prior to setting to work, Safety Cases will have to provide sufficient supportive evidence of the soundness of the design, the adequacy of all operating procedures, and that emergency conditions have been sufficiently considered in the design of the system under consideration to satisfy any relevant inspecting authority and, probably, the insurance company providing cover.

A Safety Case covers a wide range of evidence, of which only a part will deal with the control system and how it is used in the various modes the plant may experience. It will also cover the intended maintenance of the plant or equipment and, maybe, also its eventual de-commissioning.

Organisations who are mass producing an inexpensive item of consumer equipment may not see the need for making a Safety Case. Their product may seem to use such a simple device that they do not see the full consequences of its failure. It is, however, worth the effort to do a full risk analysis and record the results of the analysis in the Safety Case.

Legislation

In the United Kingdom, the main legislation is "The Health and Safety at Work Act" and "The Consumer Protection Act." However, the various European Directives have added to the weight of legislation applicable to all systems. Examples of such legislation are "The Machinery Directive," "The EMC Directive," "The Low Voltage Directive," and "The Lifts Directive." There are others. Sometimes the legislation may contradict other laws. The engineer is faced with some difficult choices and legal advice is well worth seeking in such cases. However, in the end it is your engineering judgment you rely on. It is important to support that judgment as best you can.

Standards

A standard is defined as something by which all else is measured. Standards exist to ensure that work conforms to a certain level of expectation. Standards bodies, like IEC, CCITT, ISO, and ANS, form committees to discuss issues of standardisation in a wide variety of topics. The latest standard that is expected out of the IEC stable is IEC1508 "Functional Safety: Safety Related Systems." This document is in seven parts. Parts one, two, and three are normative—dealing with the general requirements; electrical, electronic, and programmable electronic systems; and software requirements, respectively—and are due for final publication during early 1996. Part four defines the terms and abbreviations used within the standard and is also expected out early in 1996. Parts five and six are guidelines on the application of parts one, two, and three and are expected to be published during the last quarter of 1996. Part seven contains a bibliography of techniques and is expected out early in 1996. IEC1508 is a generic standard from which each industry sector is expected to derive its own sector-specific standards.

IEC1508 is expected to have a major impact on the design of all systems. This arises from its denoting four levels of Safety Integrity (SIL one to four). The only problem left to the engineer is how does he know if he is designing a system with no Safety Integrity Level requirement (let's call this a

Table Two. Safety Integrity Levels; Target Failure Measures.

Safety Integrity Level	Demand Mode of Operation (Probability of failure to perform its design function on demand)	Continuous/ High Demand Mode of (Probability of a dangerous failure per year)
4	>=10E-5 to <10E-4	>=10E-5 to <10E-4
3	>=10E-4 to <10E-3	>=10E-4 to <10E-3
2	>=10E-3 to <10E-2	>=10E-3 to <10E-2
1	>=10E-2 to <10E-1	>=10E-2 to <10E-1
This implies that:		
SIL 0	<10E-1	<10E-1

SIL0 system). I will return to this question later.

The Safety Integrity Levels are denoted in Table Two (part one of IEC1508).

Codes of Practice

Codes of practice are issued by the institutions and societies to which practitioners hold membership. Some such of these bodies are the Institute of Electrical Engineers (IEE), British Computer Society (BCS), and the Engineering Council. These codes give guiding rules on what is considered best practice in terms that generally comply with the legislative and standards requirements. One or two of these codes of practice have now become standards in their own right.

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.

Taming Variables and Pointers

Chris Jakeman
Peterborough, United Kingdom

Forth programmers enjoy unlimited access to their code and computer, but there are times when so much freedom is counter-productive. When I had trouble recently debugging some code with lots of pointers, I gave up and instead added a new word to Forth, `LimitVar`, to find my mistakes for me.

`LimitVar` is an alternative to `VARIABLE` which takes two integer limits and, by redefining ! and +!, warns if the variable is going to stray outside those limits. For example, to generate a warning, first define some variables such as:

```
S" limitvar.seq" INCLUDED
         \ Load LimitVar code from file
         \ and use it ...
0   7 LimitVar DayOfWeek
1  13 LimitVar MonthOfYear
0 100 LimitVar YearOfCentury
```

and then try to set an impossible value with !. This generates a warning such as:

```
13 MonthOfYear !
WARNING: Trying to set MonthOfYear to
13 ( 13 ) beyond limits 1 - 12
To continue, press Return
```

For convenience, this warning repeats the value in brackets as a signed integer. Note that the upper limit given is the maximum acceptable value + 1, the same format as required by `WITHIN`, which is used to implement the check.

A Stack Example

The check is even more useful for code which works with pointers, as this sort of code is harder to build correctly. For example, a stack of five cells can be built as:

Reserve space for the stack and note the limits which a stack pointer should respect:

```
HERE ALIGN
5 CELLS ALLOT
HERE CONSTANT StackTop
     CONSTANT StackBase
```

Then create a stack pointer `SP` which does respects these limits:

```
StackBase StackTop LimitVar SP
```

Now we can add three simple stack operators:

```
: EmptyStack ( -- )
  StackBase SP !
;

: Push ( n -- )
  SP @ !
  [ 1 CELLS ] LITERAL SP +!
;

: Pop ( -- n )
  [ -1 CELLS ] LITERAL SP +!
  SP @ @
;
```

and a test for overflow:

```
: PushTooFar ( -- )
  EmptyStack
  0 BEGIN
    1+
    CR ." Push " DUP .
    DUP Push
  AGAIN
;
```

That's enough—now we can test for overflow with `PushTooFar`:

```
PushTooFar
Push 1
Push 2
Push 3
Push 4
Push 5
WARNING: Trying to set SP to 32323 (
32323 ) beyond limits 32313 - 32322
To continue, press Return
```

Since 0 is useful to indicate a null pointer, we make it a special case and arrange for LimitVar to accept 0 as a legal value.

After Testing

Once the need for checking has passed, leave the definitions and limits unchanged. They will provide valuable documentation, and more checking might be needed at a later date. To disable LimitVar, comment out the include line and add a dummy definition:

```
\ S" limitvar.seq" INCLUDED
: LimitVar 2DROP VARIABLE ; \ Dummy def
```

How It Works

LimitVar is simple and surprisingly efficient. My first attempt was neither of these, so I appealed for help on the Internet newsgroup comp.lang.forth. My thanks are due to several contributors, but especially to Jonah (JET) Thomas who came up with the following idea:

- Reserve an area of memory and access it as an array.
- Build each LimitVar variable as a CONSTANT pointing into that array.
- Redefine ! and +! to check for addresses within that area. (This can be done quickly using WITHIN and makes for a fast implementation.)
- For each LimitVar in the array, store the current value and the minimum and maximum limits.

I've extended this idea here to store two additional values:

- The execution token, so that redundant entries can later be found in the array and removed.
- The address of a copy of the LimitVar's name, so that we can print a helpful warning message.

The code given here is an ANS Forth program with an environmental dependency (detailed in the text) and requires:

- WITHIN from the Core Extension word set
- FORGET from the Tools Extension word set

The data structures look like Figure One.

Checks are carried out by ! and +! using code such as:

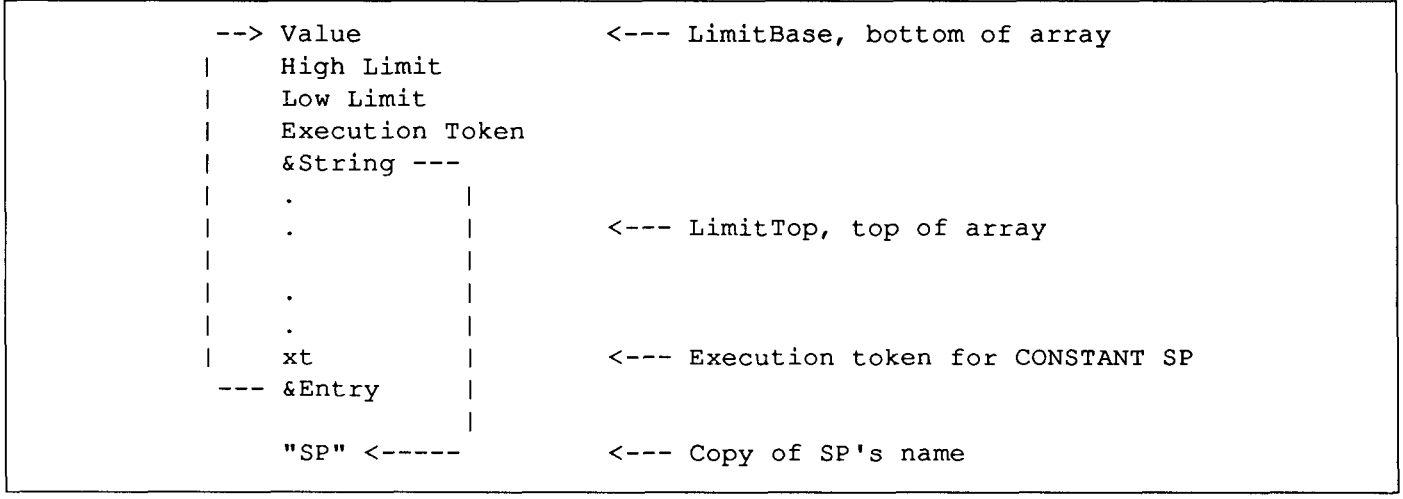
```
: !  ( x & -- )
  DUP LimitVar? IF CheckLimits THEN
  !
;
```

The basic code is given here:

```
\ Build an array at LimitBase to
\ hold data and limits for each
\ LimitVar.

\ Size it to hold 10 entries for now.
10 CONSTANT MaxLimitVars

\ The size of each entry is:
5 CELLS CONSTANT >LimitVar<

\ which will be used as:
\ Offset      Use
\ ======      ==============
\ 0 CELLS for Data
\ 1 CELLS for High Limit
\ 2 CELLS for Low Limit
\ 3 CELLS for Execution Token
\ 4 CELLS for &Name

\ Reserve space for the array:
HERE ALIGN
MaxLimitVars >LimitVar< * ALLOT

\ Note address of array for LimitVar?
\ to use:
HERE CONSTANT LimitTop
                CONSTANT LimitBase

\ An index into the array:
VARIABLE LimitVars  0 LimitVars !
```

An ANS Forth program cannot access the name of a

Figure One. The data structures.



variable, so LimitVar saves the name using CompileString which re-reads it with BL WORD, allots space, and then copies it into the space:

```
: CompileString  ( -- &Data )
\ Adds a counted string delimited by BL
\ to the data space.
  HERE  BL WORD
  2DUP C@
  1+              \ Allow for count char.
  CHARS           \ Convert to address
                  \ units.
  DUP ALLOT       \ Add to data space.
  MOVE            \ Move string into
;                 \ place.

: LVField  ( #Field -- &Field )
\ Calculates & of a field of the
\ current entry in LimitVar array.
  CELLS           \ Get offset of field.
  LimitVars @     \ Get offset of entry.
  >LimitVar< *
  LimitBase + + \ Calculate address.
;

: LimitVar ( LowLimit HighLimit ++ -- )
           ( -- &Data )    \ At run-time
\ Creates a LimitVar
  LimitVars @ 1+  MaxLimitVars >
   ABORT" No room to create a LimitVar"
  >IN @ >R        \ Save to rewind input
                  \ later.
  0 LVField       \ Compute address of
                  \ field 0.
  CONSTANT        \ Create a constant
                  \ pointing there.
  0 0 LVField ! \ Store 0 in field 0 as
                  \ the initial value.
  1 LVField       \ Store the limits in
  2!              \ fields 1 & 2.
  R@ >IN !        \ Rewind input
  '               \ and get xt.
  3 LVField  !  \ Store it in field 3.
  R> >IN !        \ Rewind input and get
  CompileString \ &NameString.
  4 LVField  !  \ Store it in field 4.
  1 LimitVars +!
;
```

That is all for creating LimitVars; now we need some words to test against the limit:

```
: LimitVar?  ( & -- Flag )
\ True if & lies within LimitVar array.
  LimitBase LimitTop WITHIN
;
```

```
: Return  ( -- )
\ A useful I/O word
  ." , press Return"
  KEY 13 = 0= ABORT" Aborted"
;

: Continue  ( -- )
\ Another one
  CR ." To continue" Return
;

: CheckLimits  ( X &Data -- X &Data )
\ Prompts with a warning if X is
\ outside limits for the LimitVar.
\ X=0 is not checked, which makes this
\ version suitable for pointers.
  OVER IF          \ If X<>0 ...
    2DUP CELL+ 2@ \ Get the limits.
    WITHIN 0= IF
      CR ." WARNING: Trying to set "
      DUP 4 CELLS + @
      COUNT TYPE SPACE  \ Variable name
      ." to " OVER U.
      ." ( " OVER .
      ." ) beyond limits "
      DUP CELL+ 2@ SWAP
      U. ." - " 1- U.
      Continue
    THEN
  THEN
;
```

CheckLimits does not consume its arguments, as this makes it more efficient.

```
: !  ( x & -- )
\ ! redefined to check LimitVars.
  DUP LimitVar? IF CheckLimits THEN
  !
;

: +!  ( x & -- )
\ +! redefined using the new !.

  DUP >R          \ Save &.
  @ +             \ Compute result.
  R> !            \ Store it.
;
```

Within And Without

WITHIN is defined in ANS in such a way that

```
5 0 10 WITHIN
```

returns true and so does

```
15 10 0 WITHIN
```

I.e., by reversing the arguments, WITHIN can be used to test for "without"! Since CheckLimits uses WITHIN, it can also be used to exclude a variable/pointer from a range.

Tidying Up

LimitVar checks that the array is not already full before adding to it, but we also need a way to remove redundant entries from the array. In the code below, we extend FORGET to do just this. At this point, conforming to ANS becomes overly complex, so instead we declare an "environmental dependency" and assume that as each word is compiled, it is given an execution token greater than the previous one.

Using this assumption, we can remove redundant entries by scanning through the array until the first redundant one is found and then leaving LimitVars to overwrite all entries from there onwards:

```
: ForgetLVs   ( xtToForget -- )
\ Recovers space for LimitVars.
\ Used by FORGET.
  0 LVField >R      \ Save & of 1st
                    \ unused entry as a
                    \ limit.
  0 LimitVars !     \ Use as a loop index
  BEGIN             \ For each LimitVar..
    0 LVField       \ Get &Entry.
  R@ U< WHILE       \ While entry in use
    3 LVField       \ Get xt for the
    @               \ the LimitVar.
  OVER U< WHILE     \ While < xtToForget
    1 LimitVars +!
  REPEAT            \ ... from 1st WHILE
  THEN              \ ... from 2nd WHILE
  DROP  R> DROP
;

: FORGET ( ++ -- )
\ Redefined to manage LimitVars
  >IN @ >R      \ Save the input stream
  ' ForgetLVs
  R> >IN !      \ Rewind the input stream
  FORGET
;
```

Enhancements

LimitVar becomes even more useful if it is enhanced with an option to show the calling tree—the sequence of words executed to reach the current warning. But that, as they say, is another story.

Chris Jakeman is an engineer sadly promoted out of hands-on software work some years ago. He retains his enthusiasm for software and its potential by using Forth to explore the boundaries. His writings often appear in the FIG-UK magazine *Forthwrite*. Send e-mail to cjakeman@apvpeter.demon.co.uk.

Does Late Binding Have To Be Slow?

András Zsótér
Hong Kong

Introduction

Today's software tool is object-oriented programming, as many programmers will agree. It is no surprise that many Forth dialects already have some sort of OOP support either built into the kernel [2, 3] or as an add-on feature [1]. On the other hand many from the Forth community will argue that the overhead involved in virtual method calls and field accesses is unacceptable in time-critical applications. This paper is directed towards them, and its main goal is to make object-oriented techniques more attractive for those who like the "close to silicon" approach.

In this paper the implementation details of my object-oriented Forth model[1] [3] will be presented. In this model the overhead involved in virtual method calls and field accesses is very low. Actually, on the Intel486 microprocessor there is *no* overhead involved in virtual method calls and field accesses. The only overhead is in object instance access. In other words, once an object has become active all virtual method calls take exactly the same time as any other call, and field accesses take as much time as an ordinary variable access. Because the usefulness of this implementation technique is not limited to Forth, most of the following ideas have already been published [4]; here the emphasis will be on our language-specific advantages.

An Object as a "Mini Universe"

I was reading an old article in *BYTE* [5] which complained about the instuctions and clock cycles spent on ordinary housekeeping in any program. The trouble is that function parameters are passed on the stack, so the caller has to push them onto the stack and the callee has to fish them out[2]. In a well-factored program this parameter passing can be significant. Of course, for Forth programmers the painful part is not parameter passing (because we use the stack for our temporaries anyway) but the need to rearrange the stack. A piece of code using only global variables would be much cleaner and faster because it would not have to fiddle around with the parameters. The problem is that such a routine would be extremly inflexible and not at all reusable.

Considering all these, I immediatly thought about OOP. An object can have its own world where variables (the fields of the object) are all in some way at a fixed address so they do not have to be passed as parameters. A routine (a method in this case) only has to know who[3] is the object calling it. Explicit parameter passing is required only in communication with the outside world.

In object-oriented languages objects and their methods—either virtual or static—constitute an alternative way of factoring. Not only common instruction sequences[4] can be factored out, but data structures and pieces of code handling them; which can lead to much cleaner program design[5].

Methods or Messages?

I have to explain at this point why I prefer the term "virtual method" to "message." The latter implies that one object communicates with another and only as a special case can the sender and the receiver of a message be the same object. The word "method" only indicates that the routine is somehow special in the sense that it operates on an object instance and assumes certain properties of that object (e.g., its memory layout). In other words, a method is a piece of code which implements a particular kind of behavior of a class of objects. The more specific term "virtual method" means a method whose identity (the actual piece of code) is not known when the program is compiled[6] so it must be determined at run time. Because, in my OOF, methods are often used for factoring out reusable functionality *inside an object,* I consider the term "message" to be rather misleading.

1. To avoid confusion I refer to this model as "my OOF" because other object-oriented Forth implementations [2] are also called OOF.
2. Of course, on CPUs with huge register files and with register windows, stack accesses can usually be avoided.
3. We consider objects as animate entities.
4. Or, in more Forthish parlance, word sequences.
5. Here I have to do what many Forthists would regard as heresy and recommend a C++ textbook [9]. The usefulness of the design principles suggested in that book are far beyond the scope of one particular programming language.
6. If the type of an object is known at compile time, a smart enough compiler can do the binding statically. If a method whose identity is known at compilation time is small enough, a compiler can even in-line it [9].

Listing One.

```
Objects DEFINITIONS              \ Objects is the base of the Class hierarchy.
3 Class Points                   \ A new class with 3 new virtual methods.
Points <|                        \ The detailed description of the new class.
Field X                          \ The X coordinate of the Point.
Field Y                          \ The Y coordinate of the Point.
Method Show   ( -- )             \ How to show a Point.
Method Hide   ( -- )             \ How to hide a Point.
Method Move   ( dY dX -- )       \ How to move a Point a given distance.
        |>

As Init use: ( Y X -- ) X ! Y ! ;M  \ Initialization of a new object instance.

: GotoXY ( -- ) X @ Y @ AT-XY ;     \ This is a static method.

As Show use: GotoXY [CHAR] * EMIT ;M
As Hide use: GotoXY SPACE ;M
As Move use: Hide X +! Y +! Show ;M

10 40 Obj P                          \ Create a new object instance.

: Test ( dY dX -- )  P { Move } ;    \ A sample word using our object.
```

An "Object-Oriented Architecture"

The above ideas might look good on paper but to learn more about them we need an implementation. When I started to experiment with my OOF I wanted to comply with the following conditions:

- The system must have efficient support for both field accesses and virtual method calls.
- The compiler has to be simple, so the efficiency of the OOP support must not depend on optimization.
- The details of OOP implementation must be expressed in the language so that the user of the system can tailor the high-level layer to his or her individual taste[7].

As a working hypothesis let us consider a microprocessor which has built-in support for the above-mentioned kind of object-oriented behavior. We need one register to contain the base address of the active object; all field addresses will be relative to this base address. Another register is needed to hold the address of the *Virtual Method Table* (VMT), which contains the addresses of the virtual method instances[8]. Our theoretical microprocessor needs a couple of instructions to load these registers, and to read their contents and save them on a stack.

So the basic ideas are clear but the next problem is that the whole thing has to be implemented somehow in hardware. Making an object-oriented microprocessor is far beyond our facilities, so I had to emulate the above behavior on an existing one. Our PCs are based on the Intel486 chip, which does not have very many registers but enough to spare two for OOP support. This microprocessor also has quite flexible addressing modes, so once the base address of an object is in a register, one instruction is enough to read or write one of its fields. The same can be said about the VMT pointer and virtual method calls. In my Dynamic Object-Oriented Forth, or DOOF (the implementation of my OOF model for the Linux operating system), I used the following register assignment:

eax	: TOS	=	The Top Of Stack
edi	: PSP	=	The Parameter Stack Pointer[9]
esp	: RSP	=	The Return Stack Pointer
ebp	: LSP	=	The Locals' Stack Pointer
ebx	: OBJ	=	The OBJect Pointer
esi	: VMT	=	The Virtual Method Table Pointer

Examples

Listing One[10] shows a sample object which is just a point on the screen (actually a character position on a character screen). This simple object has two fields which are its X and Y coordinates in a Cartesian coordinate system. A *Point* can show itself and hide (delete) itself. These are the most basic virtual methods or types of behavior a *Point* can manifest. One example of a more complicated kind of behavior is that a *Point* can move itself a specified distance in the X and Y direction. A good example for field accesses is `GotoXY` which is a static method. As we will see later there would be no performance penalty if we implement it as a virtual method. DOOF generated the sequence of Intel486 instructions[11] shown in Figure One-a for `GotoXY`. Not highly efficient code but so far no optimization has been used[12].

7. I used Forth as the assembly language of a hypothetical machine with object-oriented architecture.
8. I will call an abstract entity corresponding to a "kind of behavior" or a "kind of message the objects of a class answer to" a *virtual method*. The actual routines corresponding to the actual behavior of the objects of a given class will be referred to as *virtual method instances* in this paper.
9. Unlike in the MS-DOS version of the program, the parameter stack is not the same as the machine stack. This makes the system slower because the only efficient stack-handling instructions of the x86 family of microprocessors use the (e)sp register. So any operation which changes the number of items on the stack has to increment or decrement the parameter-stack pointer explicitly, using an extra instruction. On the other hand linking with existing C and C++ code as well as implementing most of the system in C++ is much easier in this way.
10. Because the OOP-support words are non-standard, the glossary at the end of this paper explains them.
11. DOOF, just like OOF, does not use threading. It generates machine code.
12. Because no optimization has yet been implemented in DOOF.

Figure One-a.

```
sub     edi,4           ; Make room on the stack.
mov     [edi],eax       ; Save top of stack.
lea     eax,[ebx].X     ; Get the address of the first field.
mov     eax,[eax]       ; @
sub     edi,4           ; Make room on the stack.
mov     [edi],eax       ; Save top of stack.
lea     eax,[ebx].Y     ; The address of the next field.
mov     eax,[eax]       ; @
call    AtXY            ; An ordinary call of a Forth word.
ret                     ; Return.
```

Figure One-b.

```
sub     edi,8           ; Make room for two items on the stack.
mov     [edi+4],eax     ; Save top of stack.
mov     eax,[ebx].X     ; Get the value of the first field.
mov     [edi],eax       ; Save the value on the stack.
mov     eax,[ebx].Y     ; The value of the next field.
call    AtXY            ; An ordinary call of a Forth word.
ret                     ; Return.
```

Figure Two.

```
call    [esi].Hide      ; Call Hide.
sub     edi,4           ; Make room on the stack.
mov     [edi],eax       ; Save top of stack.
lea     eax,[ebx].X     ; Get the address of the first field.
mov     edx,[edi]       ; Move value to a register.
add     [eax],edx       ; Add to the cell at the address.
mov     eax,[edi+4]     ; Move next top item to tos register.
add     edi,8           ; Adjust stack pointer.
sub     edi,4           ; Make room on the stack.
mov     [edi],eax       ; Save top of stack.
lea     eax,[ebx].Y     ; Get the address of the next field.
mov     edx,[edi]       ; Move value to a register.
add     [eax],edx       ; Add to the cell at the address.
mov     eax,[edi+4]     ; Move next top item to tos register.
add     edi,8           ; Adjust stack pointer.
call    [esi].Show      ; Call Show.
ret                     ; Return.
```

A more intelligent compiler would output something like the code in Figure One-b. This code is concise and efficient, but the problem is that the compiler has to be smart to produce this kind of output. With present compiler technology this optimization is possible [6, 8][13]. Nevertheless, the first version of the code can be produced by any dumb compiler.

If we take a closer look at either version of the code we can see that the number of instructions to access a field of the object is the same as the number of instructions needed to access a global variable (with the same smartness of the compiler)[14]. So far so good, but OOP is not only about field accesses.

The quality of an OOP implementation depends on the efficiency of virtual method calls. The only virtual method instance in our example which calls other virtual methods is `Move`, which has been compiled to the following sequence of instructions by DOOF shown in Figure Two.

Again this is rough code generated by a dumb compiler. On the other hand, what we have been interested in is clearly visible in the above list: the virtual method calls are single instructions[15]. In other words, a virtual method call is exactly as expensive as any other call as long as our program does not change the active object instance.

Changing the Active Object Instance

The word `Test` in our example changes the active object and then calls one of its virtual methods. The machine code generated by DOOF is given in Figure Three-a.

13. Of course it will not help us if we have to make a Forth implementation which runs on a machine with very limited resources.
14. A variable access usually means pushing the address of the variable onto the stack and then executing a @. In either version of the compiled program the instruction sequence would be similar to the above except that instead of [ebx+offset] a simple constant address would be used.
15. Unfortunately, on architectures with less-flexible addressing modes this is not always the case. Depending on the flexibility of indirect calls the overhead can be very small (zero instructions as on the Intel architecture, one instruction on CPUs which do not have indirect calls but can use a register as target address, and it can take even longer on very simple microprocessors where indirection is not quite supported).

Figure Three-a.

```
sub    edi,4                ; Make room on the stack.
mov    [edi],eax            ; Save top of stack.
mov    eax,offset P ; Push the address of the object onto the stack.
push   esi                  ; Save the VMT Pointer.
push   ebx                  ; Save the Object Pointer.
mov    ebx,eax              ; Load Object Pointer from the stack.
mov    esi,[ebx-4]          ; Load VMT Pointer with the new objects VMT.[16]
mov    eax,[edi]            ; Reload top of stack.
add    edi,4                ; Adjust stack pointer.
call   [esi].Move           ; Call virtual method.
pop    ebx                  ; Restore Object Pointer.
pop    esi                  ; Restore VMT Pointer.
ret                         ; Return.
```

Figure Three-b.

```
push   esi                  ; Save the VMT Pointer.
push   ebx                  ; Save the Object Pointer.
mov    ebx,offset P ; Load Object Pointer from the stack.
mov    esi,[ebx-4]          ; Load VMT Pointer with the new objects VMT.
call   [esi].Move           ; Call virtual method.
pop    ebx                  ; Restore Object Pointer.
pop    esi                  ; Restore VMT Pointer.
ret                         ; Return.
```

Of course the fiddling with the stack makes our code somehow obscure again. A smarter compiler could emit something like that shown in Figure Three-b.

From the second version of the code which can be generated from `Test` we can clearly see that the overhead in changing the active object is significant (six housekeeping instruction for only one useful virtual method call). On the other hand, if the virtual method calls other virtual methods (as in our case) or accesses a couple of fields in the active object before the program switches to another object, the code can be significantly smaller and faster than in implementations where each and every method receives the address of the object it has to operate on as a parameter on the stack[17] [4].

Multiple Inheritance and Other Bells and Whistles

My OOF has always been intended to be a *low-level* language which provides the advantages of late binding with no, or only a minor, performance hit. In such a low-level concept more complicated constructs like multiple inheritance and operator overloading have no place. Of course most of them can be implemented on top of the features provided by OOF. For example, multiple inheritance can be provided by aggregating several objects together, and for every method the compiler can present the appropriate part of the object [10]. By making the Class "Classes" (the special vocabulary type representing classes of objects [3]) a part of the class hierarchy, DOOF provides a way to derive more sophisticated tools to create new types[18].

Operator overloading is more troublesome. To meaningfully provide operators which can do arithmetics on different types of numbers we have to provide a way for the operator to check the type of its operands. Traditionally, items on Forth's stack are typeless, so this checking cannot be done without substantially modifying the internals of our traditional model of a Forth engine. The resulting language is still Forth but the overhead of the run-time type checking can make it extremly unattractive for the designers of time-critical applications.

In arguments about Forth, the complete lack of predefined data structures is listed as one of the major weaknesses of the language. After reading some C++ literature [9, 10] and seeing the complaints about C's array concept and its incompatibility with C++ objects[19], I have started to think otherwise: Our "greatest weakness" can be our biggest stength[20] because we do not have to take away anything from the language to adapt OOP. Plain old-fashioned Forth is the best starting point to make some kind of Forth++[21]. We do not have array concepts that would be incompatible with the more carefully designed data aggregates needed for an object-oriented system. Forth—very luckily—has remained a low-level tool, a kind of portable assembler, for most of its users. My OOF model has been designed to live peacefully with this approach.

16. The VMT pointer is stored in the cell immediately *before* the object. In this way the body of the object is not interrupted with an extra field and the address of the object can be passed to routines written in other languages than Forth as ordinary data structures.
17. Of course if everything is an object, and even ordinary integer arithmetics is implemented via method calls, my argument will not hold true. On the other hand such a system should only be implemented for good reasons, e.g., in certain AI programs. The presented technique is intended to improve efficiency, not to replace sanity in program design.
18. I have trouble putting it into proper words. As traditional Forth is an untyped language, we have not developed a terminology for concepts such as "the type of a type."
19. For further details see the notes at the end of x8.2.4 in reference [10].
20. This statement sounds very Taoist—no wonder many outside the Forth community consider Forth a religion rather than a programming language.
21. The term Forth++ appears more and more often in Forth

Conclusions

In this paper a fast and effective way of implementing OOP has been presented. The benefits of the technique are most useful in applications where speed is critical. The suggested technique provides the benefits of object-oriented design without the need of a sophisticated compiler. Although the examples are Intel-specific, the presented argument will hold true on most modern microprocessors.

The usefulness of this approach is not limited to one particular programming language; nevertheless, languages which encourage extensive factoring will benefit most from it. In Forth, where the average definition is only a couple of words long[22], the implicit parameters passed from method to method with no extra cost can make a program based on OOP even faster than its counterpart implemented by using more conservative techniques[23].

References

[1] Dick Pountain, *Object-Oriented Forth*. Academic Press Ltd. (London), 1987.

[2] M. Dahm, "Object-oriented Forth," *Forth Dimensions*, Vol. XIV, No. 1, 16–22 (1992 May).

[3] A. Zsoter, "An Assembly Programmer's Approach to Object-Oriented Forth," *Forth Dimensions*, Vol. XVI No. 6, 11-17 (1995).

[4] A. Zsoter, "Implementation of Object-oriented programming via register-based pointer," *Journal of Microcomputer Applications* (1995), 18, 279–285.

[5] P. Wilson, "The CPU Wars. An overview of the microprocessor battlefield, and how it got that way," *BYTE*, Vol. 13, May 1988, 213–234.

[6] D. Watson, *High-Level Languages and Their Compilers*. Addison-Wesley Publishers Ltd., 1989.

[7] P.A. Steenkiste. "Advanced Register Allocation," in *Topics in Advanced Language Implementation*, edited by P. Lee. MIT Press, 1991.

[8] T. Pittman, J. Peters. *The Art of Compiler Design, Theory and Practice*. Prentice-Hall, Inc., 1992.

[9] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, 1994.

[10] M. A. Ellis, B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Publishing Company, 1995.

Trademarks mentioned:
Intel486 is a trademark of Intel Corporation.

discussion but there is no implementation I am aware of which would actually have the name Forth++.

22. Or at least we should strive for adopting a programming style with short definitions.
23. This issue has yet to be examined. Of course I am talking about equal functionality. If a program needs a certain level of indirection (e.g., I/O primitives have been vectored in many Forth implementations), OOP is definitly a good alternative.
24. In both existing implementations of the model described in this paper, classes are represented as specialized vocabularies, each of which has a VMT.
25. A more detailed description of the "fake virtual method" Init can be found in [3].

Glossary

`Objects` (--)
The base class of all classes.

`Class` (<Name> Methods --)
Creates a new class[24] with *Methods* new slots for virtual methods in its VMT.

`<|` (-- End)
Performs the function of `DEFINITIONS` followed by pushing the offset of the end of an object instance (which is the same as the offset of the first of its new fields to be) the stack.

`|>` (End --)
Makes *End* (the first unused offset) the new default size of the class.

`Field` (<Name> o1 -- o2)
Creates a word *Name*. If *Name* is executed later it pushes the address of the cell which is *o1* away from the base address of the active object.

`Method` (<Name> --)
Creates a word *Name*. This word is the application programmer's interface to a new virtual method. If this word is executed later it will call the corresponding virtual method in the active object's VMT. The method indexes are assigned automatically as long as there are enough free slots in the class's VMT.

`As` (<Name> -- Index)
Pushes the index of the virtual method identified by *Name* onto the stack.

`use:` (Index -- use-sys)
Starts a headerless definition which will be the body of a virtual method instance of the class.

`;M` (use-sys --)
Finishes a definition started by `use:`.

`{` (Obj --)
Pushes the active object onto the return stack and then selects *Obj* as the active object.

`}` (--)
Reactivates the object which was previously pushed onto the return stack by `{`.

`Init` (args* --)
This is the name of a "virtual method"[25] which initializes the active object with *args**. The latter can be any number of arguments required by the type of the object.

`Obj` (<Name> args* --)
Creates an object with the Forth name *Name* and initializes it via Init.

András Zsótér's object-oriented Forth for Linux can be downloaded via FTP:
ftp://taygeta.com/pub/Forth/Linux/doof.README
ftp://taygeta.com/pub/Forth/Linux/doof-0.0.1.tgz

LIFE That Knows When to Stop

JOHN HORTON CONWAY's Game of Life has been a popular computer pastime since MARTIN GARDINER wrote about it in his *Scientific American* column in 1970 and after. Implementing it is a favorite assignment or project in programming classes.

The implementation given here has two features that are unusual.

1. It does not run until terminated by the operator, but stops by itself when it detects the inevitable cycle.
2. It checks and updates 32 life-cells at a time.

If you are already familiar with the game, skip ahead until section "Smart Life."

Theoretically, the universe of the Game of Life is an infinite square board. Every computer representation must necessarily have a finite universe. This means that the game must eventually end in a cycle. The cycle may be a steady-state, that is, a cycle of length one.

The elements of universe are cells, or as I call them here, *life-cells*. A life-cell can be *dead* or *live*, represented by a bit.

The rules for the Game of Life are:

- A dead life-cell becomes live in the next generation if it has exactly three neighbors.
- A live life-cell remains live in the next generation if it has exactly two or three neighbors.
- A live life-cell becomes dead in the next generation if it has fewer than two or more than three neighbors.

You specify which life-cells are live in the first generation; the rules determine what happens to them.

The neighbors of a life-cell are the adjacent eight life-cells.

Exhibit One-a gives the first eight generations of *Five in a Row*. After that, the last two repeat forever as in Exhibit One-b.

Three-in-row, which alternates with Three-in-a-column, occurs often enough to have its own name, *blinker*. Four blinkers like the above are called a *traffic light*. The generation of blinkers or traffic lights is the most common cause of a cycle of two.

Exhibit One-a.

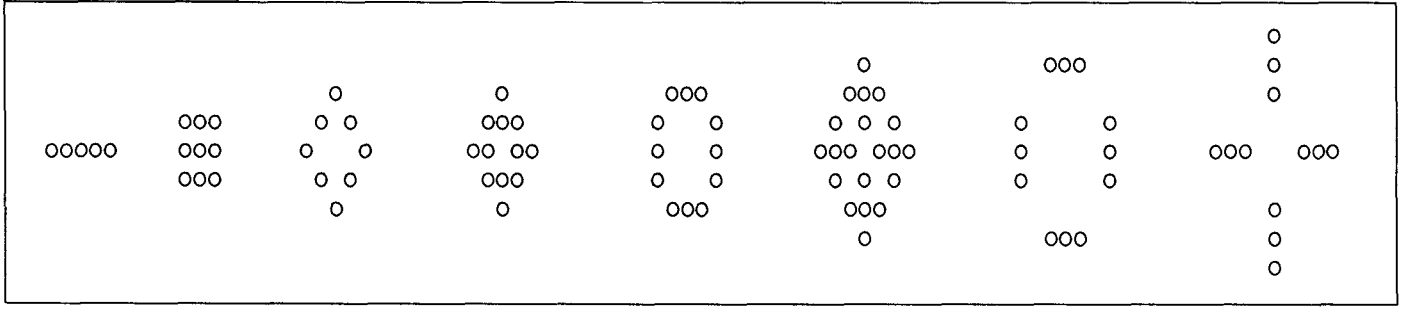


Exhibit One-b.

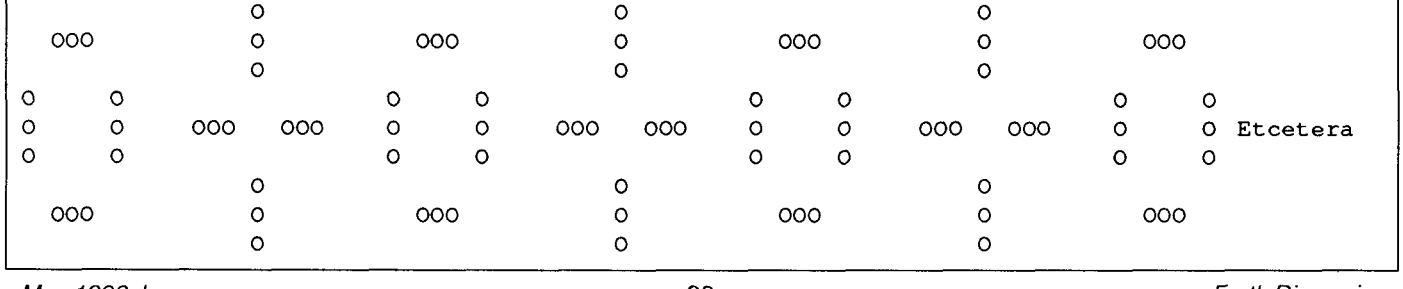


Exhibit Two. Glider.

```
 O
  O
OOO
```

Exhibit Three-a. The r-pentomino.

```
 OO
OO
 O
```

Smart Life

In this implementation the topology of the universe is a torus, that is, the universe wraps around top to bottom and left to right.

You can change the size of the universe by changing the definitions of `columns` and `rows`. The values used here are 64 × 25.

As examples of cycle detection see Exhibit Two and Exhibits Three-a through Three-c.

The glider in any of its eight rotations and reflections will reproduce itself every four generations, moving horizontally and vertically one life-cell. Unless it encounters an obstruction, in a 64 column by 25 row universe it will return to its starting position in 6400 generations, recognizing a cycle of 6400 generations. Adding something out of harm's way that vanishes early, for instance, six in a row or 15 in a row, will cause the cycle to start later, but it will still be detected at generation 6400. In an infinite universe it will travel forever unless it runs into something.

The r-pentomino in a 64 × 25 universe will become ten still-lifes and a traffic light in a cycle of two after 1222 generations. It will create several gliders which will smash into something and destruct. The glider was discovered by Conway and his friends while they were watching the r-pentomino.

Changing the values of rows and columns will change how configurations develop.

On a 48K Apple II in the early 1980s I accidentally had a configuration which took six days to go 3.4 million generations before entering a short cycle. I have not been able to rediscover it.

Cycle detection is easy to implement. Copy the starting configuration to a "scratch" universe. Then for every generation in the "real" universe advance two generations in the scratch universe. When the "real" universe and the "scratch" universe are identical, the cycle has been found. The cycle will be detected somewhere in its first occurrence in the real universe.

The other special feature of this implementation is the updating of life-cells `CELL-BITS` at a time, where `CELL-BITS` is the number of bits in a Forth cell.

This is done by using `AND`, `OR`, `XOR`, and `INVERT` to imitate logic gates for two-bit adders and three-bit adders. These are not used on single bits but on Forth cells of `CELL-BITS` bits.

As the simplest example, consider adding the bits of two Forth cells in parallel.

`XOR` gives every bit 0 of the `CELL-BITS` answers. `AND` gives every carry bit of the `CELL-BITS` answers.

Exhibit Three-b. The r-pentomino after 100 generations in a 64 × 25 universe. The starting position was at the center of the universe. Find a glider in an intermediate stage. Hint: it is made up of five cells.

```
                                                  OO
                                                 O  O
                                        OO       O  O
                                        OO         OO
                          OO
 O                         OO       OO            OO O
OOO                        O        OO            OO OO
  OO                                                                    O
O OO                                                 OOO OO            O
O                                      OOO           OO  OO           OO
O O                              OO      O O OO      OOOOO             O
O  O                             OO      O    O O   O OO
                                       O  O O    O   O O
O  O                                   O   OO O O   O
 OO                                    O      OOO
O OO                                                                    O
O
     O       OO                                                        OO
     O       OO                                                        OO
                                                                       O
    O                                                                  O
 O                                                                     O
OO                                                                      O
```

`sum-the-row` adds the bits of three Forth cells in parallel. For example,

```
left:       00001111
middle:     00110011
right:      01010101
=
sum-1       00010111
sum-0       01101001

            01121223
```

`add-to-pair` adds in parallel the bits of a Forth cell to the bits of a pair of Forth cells, setting the bits of Forth cell `carry`.

```
x           00001111
+
sum-1(or2)  00110011
sum-0(or1)  01010101
=
carry       00000001
sum-1(or2)  00110110
sum-0(or1)  01011010

            01231234
```

Combining these in `add-to-sum-210`, we add the bits of nine Forth cells in parallel into `sum-2`, `sum-1`, and `sum-0`. Luckily, the results eight and nine are treated the same as the results zero and one.

The program has been written to avoid hardware dependencies. If your system permits, you should re-write `show-universe` to refresh the display rather than scroll. Standard Forth `AT-XY` can be used for this. You may have to change the value of `rows`. Even better is to increase the values of `columns` and `rows` and use a graphic bit display. The value of `columns` will be effectively rounded down to a multiple of `CELL-BITS`.

For additional credit, what does the following Forth word do?

```
: X                             ( ??? -- ??? )
      BEGIN   ?DUP
      WHILE   2DUP AND 2* >R    XOR R>
      REPEAT
;
```

This classic Forth puzzle was my inspiration for this implementation of the Game of Life.

Menopause

The most important change is to display the universe without scrolling so life-cells can seem to move. Also try changing the values of `rows` and `columns`. Use `VALUE` instead of `CONSTANT` so you can change interactively.

See how big a universe you can have and be happy with.

You can investigate what happens when the topology is a rectangle instead of a torus.

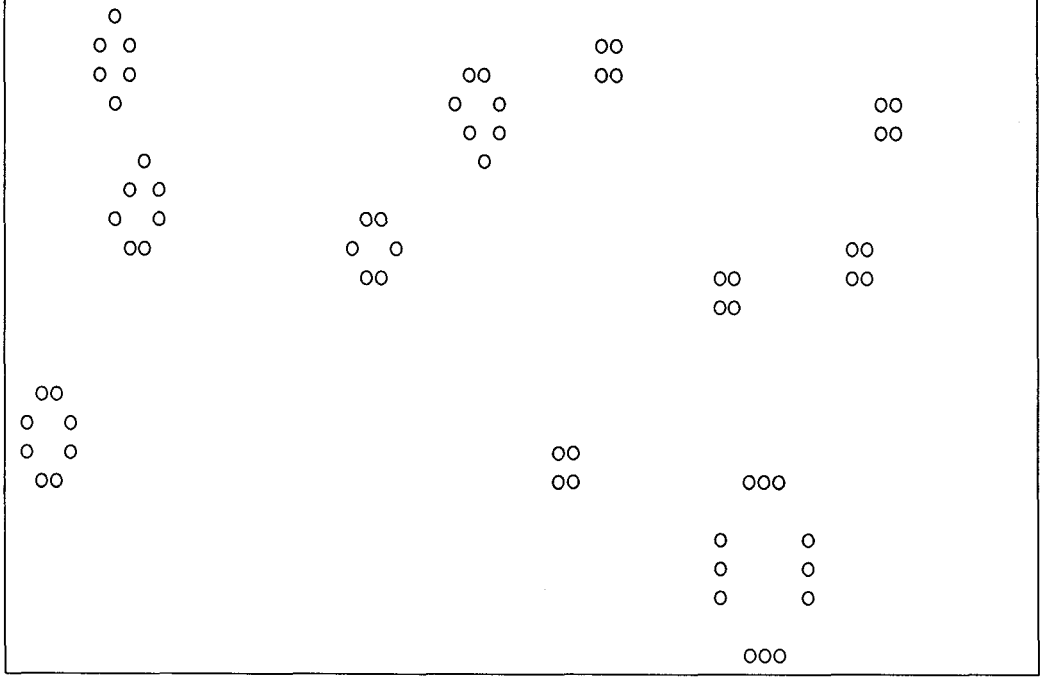
When looking for long-running configurations, put `TRUE TO TESTING` to avoid massive printing.

The Truth

Real life is not a game. It's a toy.

A game has a known purpose. Real life does not.

Exhibit Three-c. The r-pentomino after 1222 generations in a 64 × 25 universe.



Code begins on next page...

Wil Baden is a professional programmer with an interest in Forth. A text-only edition of this article is available for the asking by e-mail to:
wilbaden@netcom.com

Listing. Life That Knows When to Stop.

```
( Life That Knows When To Stop       Wil Baden  197?-1996 )
( J. H. Conway's "Game of Life", with cycle detection.    )
( Stops when the inevitable cycle is detected.            )
( Computes new generation CELL-BITS life-cells at a time. )
( Used:
            fileid LIFE
  where
      fileid is for a text file with the starting generation.
      "O" for live; " " for dead, i.e. empty.
      Note: Any text file can be used for examples.
      It would be easy to convert to BLOCK files.
)

( Environmental Values )

8                              CONSTANT ADDRESS-UNIT-BITS
ADDRESS-UNIT-BITS CELLS        CONSTANT CELL-BITS

( The following are the key parameters.  They determine the size
( of the universe.  Change them to suit yourself.                )

64                             CONSTANT columns
25                             CONSTANT rows

columns  CELL-BITS /           CONSTANT cols

1  CELL-BITS 1-  LSHIFT        CONSTANT mask-bit

( This code presumes that the bit-map goes left-to-right from
( high-bit to low-bit.  If the bit-map goes the other way
( define `1 CONSTANT mask-bit' and exchange LSHIFT and RSHIFT
( in `SET-BIT', `sum-the-row', and `show-universe'.
( The change is needed only if you are doing a bit-display.   )

FALSE                          VALUE    TESTING

( Working Variables )

VARIABLE left         VARIABLE middle      VARIABLE right

VARIABLE center

VARIABLE top-1        VARIABLE top-0
VARIABLE mid-1        VARIABLE mid-0
VARIABLE bottom-1     VARIABLE bottom-0
VARIABLE sum-2        VARIABLE sum-1       VARIABLE sum-0

VARIABLE carry

( Ancillary Operations )

: XOR!       DUP >R @ XOR R> ! ;           ( x a -- )
: AND!       DUP >R @ AND R> ! ;
: OR!        DUP >R @ OR  R> ! ;

: SET-BIT                                  ( offset addr -- )
      >R                                        ( offset)( R: addr)
            CELL-BITS /MOD CELLS R> + >R   ( bit#)
            mask-bit SWAP RSHIFT           ( mask)
      R> OR!                               ( )( R: )
;

( Add CELL-BITS 1-bit-numbers to CELL-BITS 2-bit-numberpairs. )

: add-to-pair                              ( x a b -- )
```

```
66        ( Sets: carry )
67        ROT                                        ( a b x)
68        DUP carry !
69        OVER XOR!                                  ( a b)
70        @ INVERT carry AND!                        ( a)
71        carry @ SWAP XOR!                          ( )
72 ;

74 ( Add CELL-BITS 2-bit-numberpairs to the 3-bit-numbersums. )

76 : add-to-sum-210                          ( x y -- )
77        ( Sets: sum-2 sum-1 sum-0 )
78        >R                                         ( x)( R: y)
79              sum-2  sum-1  add-to-pair            ( )
80        R> sum-1 sum-0 add-to-pair                     ( R: )
81        sum-1 @ INVERT  carry @ AND  sum-2  XOR!
82 ;

84 ( The sum of three 1-bit numbers can be expressed with 2 bits.
85 ( That is what `sum-the-row' does, CELL-BITS at a time.        )

87        ( If bit-map left-to-right is high-bit to low-bit:       )

89        (       left      middle            right
90        (     | . . . L | 31 30 . . . 1 0 | R . . . |            )

92 : sum-the-row                          ( left middle right -- )
93        ( Sets: sum-1 sum-0 )
94        0 sum-1 !
95        OVER  ( = | 31 30 . . . 2 1 0 | ) sum-0 !
96        CELL-BITS 1- RSHIFT  OVER  1 LSHIFT
97              ( = | 30 29 . . . 1 0 R | ) OR
98        sum-1 sum-0 add-to-pair                    ( left middle)
99        1 RSHIFT  SWAP  CELL-BITS 1- LSHIFT
100             ( = | L 31 30 . . . 2 1 | ) OR       ( left)
101       sum-1 sum-0 add-to-pair                    ( )
102 ;

104 ( The Rules of the Game of Life )

106 ( sum-2 sum-1 sum-0 center Stack  Result )
107 ( 0     0     0     x      0 0 x  0  0 or 8 total is dead. )
108 ( 0     0     1     x      0 1 1  0  1 or 9 total is dead. )
109 ( 0     1     0     x      1 0 x  0  2 total is dead. )
110 ( 0     1     1     x      1 1 1  1  3 is survive or birth. )
111 ( 1     0     0     x      1 1 x  x  4 total is survive. )
112 ( 1     0     1     x      1 0 1  0  5 total is dead. )
113 ( 1     1     0     x      0 1 x  0  6 total is dead. )
114 ( 1     1     1     x      0 0 1  0  7 total is dead. )

116 : live-or-dead                          ( -- x )
117       sum-2 @  sum-1 @  XOR
118             ( 2 3 4 or 5 )
119       sum-2 @  sum-0 @  XOR
120             ( 1 3 4 or 6 )
121       AND
122       ( 3 or 4 )
123       center @ @  sum-0 @ OR AND          ( x)
124 ;

126 ( `the-universe @' is a bit map of the Game of Life's world. )
127 ( `new-universe @' and `alt-universe @' are scratch universes. )

129 CREATE the-universe    HERE CELL+ ,   cols rows * CELLS ALLOT
130 CREATE new-universe    HERE CELL+ ,   cols rows * CELLS ALLOT
```

```
131 CREATE alt-universe    HERE CELL+ ,   cols rows * CELLS ALLOT

133 ( This is a typical definition to initialize the universe. )

135       : checked   ABORT" (File-Access Error) " ;

137       \ The definition of `NOT' is up to you.

139 : purge-universe                    ( -- )
140       the-universe @  cols rows * CELLS  ERASE
141       new-universe @  cols rows * CELLS  ERASE
142 ;

144 : load-universe                      ( fileid -- )
145       purge-universe
146       rows 0 DO
147             PAD cols CELL-BITS * 2 PICK READ-LINE checked
148                   0= IF   DROP LEAVE   THEN
149             0 ?DO
150                   I CHARS PAD + C@ [CHAR] O < NOT IF
151                         I  J cols CELLS * the-universe @ + SET-BIT
152                   THEN
153             LOOP
154       LOOP                                DROP
155 ;

157 ( Fill the universe with the next generation. )

159 ( The topology of the universe is a torus: the top and
160 ( bottom rows are next to each other; the leftmost and
161 ( rightmost columns are next to each other.            )

163 : spawn-a-generation                          ( Universe -- )
164       ( For CELL-BITS columns at a time. )
165       cols 0 DO
166             ( Get left, right, and middle cell positions. )
167             I 1- cols + cols MOD CELLS OVER @ +  left !
168             I 1+ cols MOD CELLS OVER @ +  right !
169             I CELLS OVER @ +  middle !

171             ( Get sums for mid row. )
172             left @ @  middle @ @  right @ @  sum-the-row
173             sum-1 @  mid-1 !    sum-0 @  mid-0 !

175             ( Get sums for wrap-around "top" row. )
176             left @ rows 1- cols * CELLS + @
177                   middle @ rows 1- cols * CELLS + @
178                         right @ rows 1- cols * CELLS + @
179                               sum-the-row
180             sum-1 @  top-1 !    sum-0 @  top-0 !

182             rows 1-  0  DO                              ( .)
183                   ( Remember the center life-cell. )
184                   middle @  center !

186                   ( Get new bottom row. )
187                   cols CELLS  left +!
188                   cols CELLS  middle +!
189                   cols CELLS  right +!

191                   ( Get sums for bottom row. )
192                   left @ @  middle @ @  right @ @  sum-the-row
193                   sum-1 @  bottom-1 !    sum-0 @  bottom-0 !

195                   ( Add sums for mid and top rows. )
196                   0 sum-2 !
```

```
197                   mid-1 @  mid-0 @  add-to-sum-210
198                   top-1 @  top-0 @  add-to-sum-210

200                   ( Judge. )
201                   live-or-dead
202                         J I cols * + CELLS new-universe @ + !

204                   ( Save new sums for mid and top rows. )
205                   mid-1 @  top-1 !      mid-0 @  top-0 !
206                   bottom-1 @  mid-1 !   bottom-0 @  mid-0 !
207             LOOP                                        ( Universe)
208             ( Remember the center life-cell. )
209             middle @  center !

211             ( Get the wrap-around "bottom" row. )
212             I 1- cols + cols MOD CELLS OVER @ +  left !
213             I 1+ cols MOD CELLS OVER @ +  right !
214             I CELLS OVER @ +  middle !

216             ( Get sums for bottom row. )
217             left @ @  middle @ @  right @ @  sum-the-row

219             ( Add sums for mid and top rows. )
220             0 sum-2 !
221             mid-1 @  mid-0 @  add-to-sum-210
222             top-1 @  top-0 @  add-to-sum-210

224             ( Judge. )
225             live-or-dead
226                   I rows 1- cols * + CELLS new-universe @ + !

228       LOOP
229       ( Exchange the-universe and new-universe. )
230       DUP @ new-universe @ ROT ! new-universe !       ( )
231 ;

233 ( This displays the bit-map of the universe.  It should be
234 ( replaced by high-performance system-specific graphic routine. )

236       CREATE linebuffer   columns CHARS ALLOT
237       VARIABLE #out

239 : show-universe                              ( -- )
240        rows 0 DO
241             0 #out !
242             cols 0 DO
243                   I J cols * + CELLS the-universe @ + @ ( cell)
244                   CELL-BITS 0 DO                ( cell)
245                         DUP mask-bit AND IF
246                               [CHAR] O
247                         ELSE
248                               BL
249                         THEN                        ( . char)
250                               linebuffer #out @ CHARS +  C! ( cell)
251                         1 #out +!
252                         1 LSHIFT
253                   LOOP                              DROP
254             LOOP
255             linebuffer #out @ -TRAILING TYPE   CR
256       LOOP
257 ;

259 ( The inevitable cycle is found by advancing the alternate universe
260 ( two generations for every generation of the reference universe.
261 ( When the two universes are equal the cycle has been detected.    )
```

```
263       : BOUNDS        OVER + SWAP ;        ( a n -- a+n a )

265 : detect-cycle                          ( -- flag )
266       the-universe @                      ( addr)
267       alt-universe @  cols rows * CELLS  BOUNDS DO
268             DUP @ I @ = NOT
269                   IF   DROP FALSE UNLOOP EXIT   THEN
270             CELL+
271       1 CELLS +LOOP                        DROP
272       TRUE
273 ;

275 ( Spawn new generations until the cycle is detected. )

277       VARIABLE                generation

279 : show-life                             ( -- )
280       the-universe @  alt-universe @  cols rows *  CELLS MOVE
281       0                                   ( generation#)
282              BEGIN
283                   the-universe spawn-a-generation
284                   1+
285                   TESTING IF
286                         DUP  100 MOD 0=  IF
287                               DUP .  CR
288                         THEN
289                   ELSE
290                         ." Generation " DUP .  CR
291                         show-universe
292                   THEN
293                   alt-universe spawn-a-generation
294                   alt-universe spawn-a-generation
295                   detect-cycle
296             UNTIL
297        generation !                              ( )
298 ;

300 ( After the cycle has been found, show it.  The cycle may
301 ( be detected at any place within its first occurrence.   )

303 : show-cycle                            ( -- )
304       0                                     ( cycle#)
305             BEGIN
306                   the-universe spawn-a-generation
307                   1+
308                   ." Generation " DUP . ." in cycle " CR
309                   show-universe
310                   detect-cycle
311              UNTIL
312        ." Cycle of " .                      ( )
313              ." after " generation @ . ." generations "
314        CR
315 ;

317 : evolve                                  ( -- )
318       show-universe
319       show-life
320       show-cycle
321 ;

323 : life                                    ( file -- )
324       load-universe                           ( )
325       evolve
326 ;

328 \ Procedamus in pace.   Wil Baden    Costa Mesa, California
```

16th Annual Rochester Forth Conference
"Open Systems"

June 19–22, 1996
Ryerson Polytechnic University
Toronto, Ontario, Canada

featuring invited speakers

MITCH BRADLEY
Firmworks Inc.
creator of "Open Firmware"

CHARLES MOORE
Computer Cowboys
inventor of the Forth language

The Institute for Applied Forth Research is pleased to present the 16th Annual Rochester Forth Conference, on Open Systems. Since 1981, the Rochester Forth Conference has been a popular forum for researchers, developers, users, and vendors of the Forth language. This year we extend our welcome to all developers of Open Systems for an exchange of knowledge and opinion!

Papers • Working Groups • Poster Sessions • Vendor Exhibits • Tutorials

Papers will be presented on all aspects of Forth technology. Special topics include Open Firmware, Plug and Play Systems, Scripting Languages, SGML and HTML, Java, Distributed Computing. There will also be papers and working groups on Forth programming standards (including ANS Forth), embedded and real-time systems, scientific/engineering applications, and education. Tutorials will include: Introduction to Forth, Advanced Forth, Forth Under Windows, Metacompilation, "Open Boot" Firmware, HTML, Java, and TCP/IP. Many of the major vendors of Forth and related technologies will be on hand to demonstrate their products.

Conference Location

This year, for the first time, the Rochester Forth Conference moves to Toronto, Ontario—the city Peter Ustinov called "New York run by the Swiss." Toronto is a center of high technology, finance, and culture, and is one of the world's most popular tourist destinations. Our conference venue at Ryerson Polytechnic University is in the heart of downtown Toronto, within walking distance of many attractions. Toronto is ideally located for U.S., Canadian, and international travel.

REGISTRATION:

- ❑ US$475 Attendee
- ❑ US$200 Full-time student
  (CAD$200 for a Canadian student)
- ❑ US$120 Spouse

Rooms

- ❑ US$150 Single for four nights (Wed., Thu., Fri., Sat.)
- ❑ US$100 Student single for four nights
- ❑ US$200 Double (2 persons, 1 double bed) for four nights

Send registrations to:
Rochester Forth Conference
Box 1261
Annandale, VA 22003 USA
lforsley@jwk.com

US check, Visa, or MasterCard only

FOR MORE INFORMATION:

Elliott Chapin
24 Monteith St.
Toronto, ON Canada M4Y 1K7
echapin@interlog.com 416–921–9560

or see our World Wide Web page at
http://maccs.dcss.mcmaster.ca/~ns/96roch.html

SPONSORS:

Microtronix Systems Ltd. –
London, Ontario, Canada
JWK International Corporation
Annandale, Virginia, USA

Presented in cooperation with:
the Southern Ontario Forth Interest Group
and McMaster University

Using Forth to manipulate the real world

Forthware

An Introduction to Power Control

Skip Carter
Monterey, California

Introduction

In this month's column, we will expand our ability to control devices by looking at what we need to do in order to turn AC and DC power off and on. There won't be any code in this installment, instead we will be exploring part of the alphabet soup of devices that can be used to manage electrical power. This will provide us with the basic background we will need in the future.

Switching Direct Current

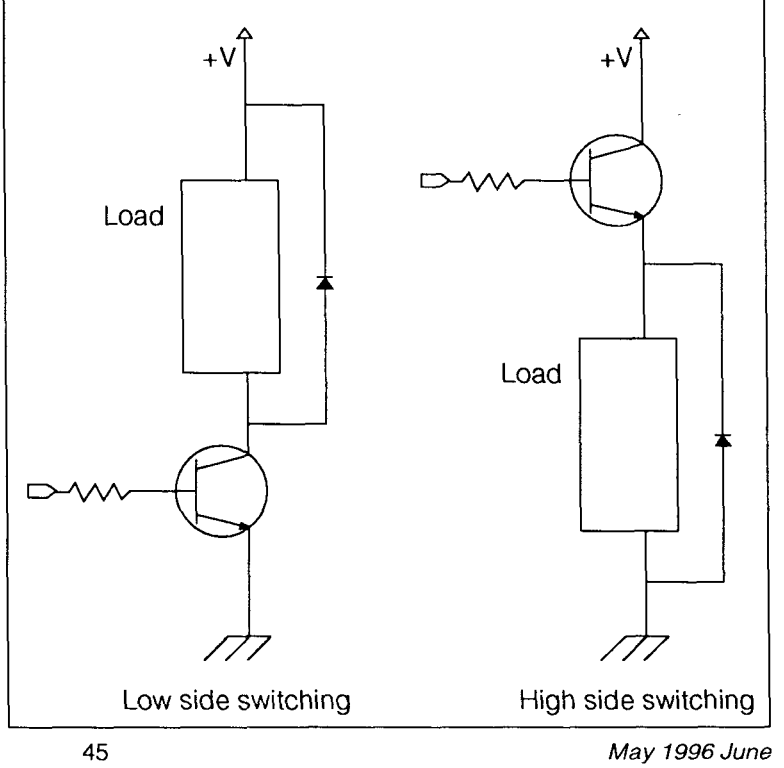
It was in disguise, but we have already seen one good method for controlling DC power to devices. This was the part of the stepper motor circuit that switched power to the different coils of the motor *[see* FD *XVII/5].* Figure One is nothing more than that circuit for one coil with a re-arranged layout; it primarily consists of a bipolar Darlington power transistor. The diode is there in order to protect the transistor during switching when the load is inductive. If you are going to be switching non-inductive loads, the diode can be omitted. The resistor at the base is to control the base current to the transistor when it is switched on. This resistor needs to be chosen with some care: since the controlling digital port needs to be able to supply this current, this implies that we want a relatively large resistor so that the current is small. But we cannot make it too small or we won't get a useful amount of current through the collector. For a given transistor, the ratio of the collector current, I_C, to the base current, I_B, is a fixed value, the *current gain*:

$$\beta = \frac{Ic}{IB}$$

This gain is typically around 50 for a power transistor. So a 10 milliamp current at the base can switch half an amp through the collector; okay, but not great. Signal transistors have a value of β that can be up around 200, but they have upper limits on the amount of current that can go through them; they are not designed to switch large currents. If we drive the power transistor with the output of a signal transistor, we can get very large gains and still switch lots of current. This configuration is what is done internally in a Darlington power transistor. With a value of β around 1000 we can switch 10 amps from our 10ma control signal (this is respectively large, not all power transistors can handle that much!).

Notice that the circuit consists of the voltage supply, the load to be driven, the transistor, then ground. This is called *low side switching*, i.e., we are connecting the circuit to ground when the circuit is turned on. Notice that the current flows through *two* PN junctions to get from the low side of the load to ground. Each of these junctions imposes a voltage drop. Depending upon the transistor, this drop is typically in the range of 0.3 to 0.6 volts per junction. This means that the low side of the load will actually be significantly *above* ground. One can avoid this by swapping the places of the load and the transistor so the transistor connects the load to the voltage supply when it is turned on

Figure One-a (left) **and One-b.** Switching DC power.



(Figure One-b). This is known as *high side switching*. The voltage drop still exists, but now it is on the supply voltage being applied to the load. The ground for both the load and the controlling circuit is the same. In some circumstances this arrangement is more desirable, but it comes at a very high price. Take a look at this circuit from the point of view of the driver at the base. It looks like a diode going from the base to the emitter. In order to turn on this diode, it must be *forward biased*. For the NPN transistors we are using here, this means the voltage at the base must be greater than the voltage at the emitter by at least the amount of the diode voltage drop. This is not too hard to achieve if we are controlling a 3 volt circuit from our 5 volt I/O pin that is driving the base. But what we'd really like to do is switch larger voltages, say 12 volts, from our processor. If we use the transistor this way, it ends up acting more like an *amplifier* giving some proportion of the voltage at the collector, instead of the *switch* that we really want; this can waste a lot of power. The only way to really achieve what we want is to put some kind of voltage amplifying device at the base, so that the base voltage is larger than the collector voltage. In summary, use low-side switching to control power unless you really must use high-side.

The bipolar transistors we have used so far are *current* controlled devices. Using a Darlington reduces the current required to turn on the transistor, but the current is still the controlling factor. If we switch to a MOSFET power transistor, we can do the equivalent things, but now we are dealing with a *voltage* controlled device. Because it is voltage controlled, very little current is needed to switch the transistor—which is nice for driving from digital output pins. There is a voltage drop between the *drain* and *source* (which are functionally equivalent to the collector and emitter), but in MOSFETs it is caused by the resistance of the semiconductor material in the on state. This resistance can be quite small, thus there can be very small voltage drops across the device. The small voltage drop means an efficiency gain in our transistor switch, but in order to achieve it we need to apply enough voltage at the gate (the equivalent of the base) in order to fully turn the transistor on (if the transistor is only partially on, the drain-to-source current path has a relatively large resistance). Trying to achieve the fully on state of the transistor is where we run into a disadvantage with MOSFETs: the typical power MOSFET requires gate voltages around 8 volts or more. So now our device must have a bipolar transistor switching power to the base of the MOSFET so *it* can turn on the real power to the contolled circuit. At least, with this arrangement the bipolar transistor does not need to handle much current. The situation with MOSFETs is even worse when used in a high-side configuration: in this case, the voltage at the gate must be substantially higher than the voltage supply at the drain (by 8 or 10 volts). As with bipolar transistors, high-side switching without the use of a voltage amplifier on the control signal is extremely inefficient.

The transistors we have worked with here have been NPN or, the equivalent MOSFET concept, n-channel. We could have used the alternate form, PNP (or p-channel for MOSFET). However, because of differences at the quantum mechanical level, PNP devices are less efficient, particularly at high frequencies. Therefore, we will avoid using them unless there is no other choice.

The SCR (Silicon Controlled Rectifier) is a device that acts like a diode with a control line attached, and it has some useful properties. It can be used like a transistor to switch current on or off. When turned on, it behaves like a diode in the circuit. But when it is turned off, an interesting thing happens: the SCR *continues to conduct until the source current stops*. This makes it useful in applications where you want to turn on something permenantly but only after, say, other parts of the system have been properly intialized. Figure Two shows a typical application of an SCR.

Switching Alternating Current

Now that we can control DC power, what about controlling AC? The SCR *almost* gives us what we want except, since it acts like a diode, it rectifies the voltage. What about putting a pair of SCRs together with their polarities reversed? It turns out that this is the right approach; such a device is known as a TRIAC. The matched, reversed SCRs means that a TRIAC will conduct current in both directions when it is turned on. In other

Figure Two. Switching DC with an SCR.

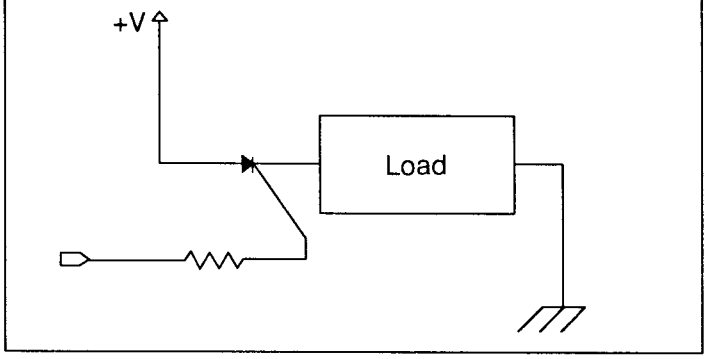
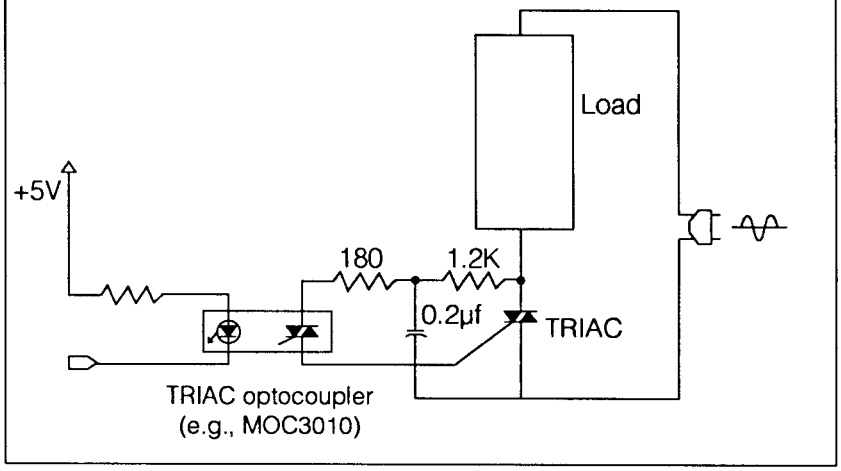


Figure Three. Example TRIAC circuit.



respects, a TRIAC behaves just like an SCR. This means that, if it is being used to control AC power and the control signal turns it off, the current will cease to flow when the source current drops to zero. The current drops to zero 120 times per second for AC (or 100 times for 50 Hz current), so the voltage out of the TRIAC will drop quickly (but not immediately) when the control signal turns it off.

If you are like me, the idea of getting AC power that close to your computer's logic-level circuitry will make you nervous. That is why, in my example TRIAC circuit (Figure Three), the logic signal is optically isolated from the AC supply: the optical isolator output drives the TRIAC control input.

For controlling *inductive* AC loads, we still need to consider the huge voltage spike we will get when the magnetic field collapses when power is switched off. Because of the alternating current, we don't use a diode to do the shunting; instead, we use a capacitor. For resistive loads, the circuit in Figure Three can be simplified by removing the capacitor and replacing the 1.2K resistor with just a piece of wire.

Feedback

My last column on using Linux seems to have resonated with many of you. I got lots of feedback, but one of the most interesting was a suggestion from András Zsótér for a possible alternative approach to getting to the I/O ports. András notes there is character device `/dev/port` which maps to *all* the hardware I/O ports. To get to a particular I/O port, use the port address (say, 0x378) as an offset into the device file and reposition the file pointer to that location.

So you can manipulate it in the manner [shown in Figure Four-a.]

Again, in order for this to work, your application must be allowed read-write access on `/dev/port`. The device `/dev/port` is rather special, it is much safer to use `makedev` to create a new entry in `/dev` with the identical major and minor device numbers, and adjust the permissions on *that* file [as in Figure Four-b.]

Here, `/dev/io` is a new device driver entry point (it points to the same code as `/dev/port`, since the device numbers are the same); this is the name of the "file" that should be opened in the above code. In this example, anyone in the group `users` can gain read-write access to the port (you can create a special group that not all users are members of and change the group of the device, if you want to restrict access to only some users—this is what András does for his system).

This is a clean and direct approach to getting to the ports, but it has one weakness: the management of multiple process access to the devices is problematic. Depending upon the way the system is configured and the file access flags used when the file was opened, a file is either exclusively for the use of the process or it can be shared among multiple processes. Either choice causes problems. If the file is *not shared* then once a process has opened the I/O port device, no other process can get to *any other ports* without resorting to very special tricks. So, if you are running one program in the background that uses an A/D board at 0x320, you cannot have another program running stepper motors on the parallel port at 0x378. Alternatively, if you open the file as shared, then two different processes can get access to the same device at the same time and not know it. Consider the confusion that would be caused by two different processes trying to simultaneously initialize a device (like the 8255) that takes multiple configuration bytes that are successively written to a single register. There are ways around this, but if you are going to be making a practice of doing this sort of thing, you should probably just write and install a proper device driver.

I am also pleased to report that there is at least one more ANS Forth compiler available for Linux/Unix: gForth by Bernd Paysan (paysan@informatik.tu-muenchen.de).

Figure Four-a. Alternative approach to I/O ports under Linux.

```
S" /dev/port" R/W OPEN-FILE ABORT" unable to open /dev/port"

CONSTANT IO/PORTS               \ file descriptor for I/O Ports

: in ( port -- value )
   S>D IO/PORTS REPOSITION-FILE THROW  \ seek to port location
   IO/PORTS pc@
;

: out ( value port -- )
   S>D IO/PORTS REPOSITION-FILE THROW
   IO/PORTS pc!
;
```

Figure Four-b.

```
crw-rw----   1 root     kmem      1,   4 Jul 17  1994 /dev/port
crw-rw-r--   1 root     users     1,   4 Mar 11 18:53 /dev/io
```

Conclusion

We now have the basic tools we need for controlling electrical power to devices. Next time we will combine these circuits with some Forth code to control DC motors. 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 the President of the Forth Interest Group.

CALL FOR PAPERS
FORML CONFERENCE

The original technical conference for professional Forth programmers and users.

Eighteenth 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 it's 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 "gotcha's" 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 that 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, PO Box 2154, Oakland, CA 94621 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, PO Box 2154, Oakland, CA 94621, (510) 893-6784, fax (510) 535-1295

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