$10 Volume XVIII, Number 4 November 1996 December

FORTH DIMENSIONS

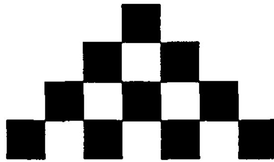
C-Style Arrays

Filters and Sponges

A Window Interface

A Discipline of ANS Forth Programming

Measuring Frequency and Sampling Time-Dependent Signals

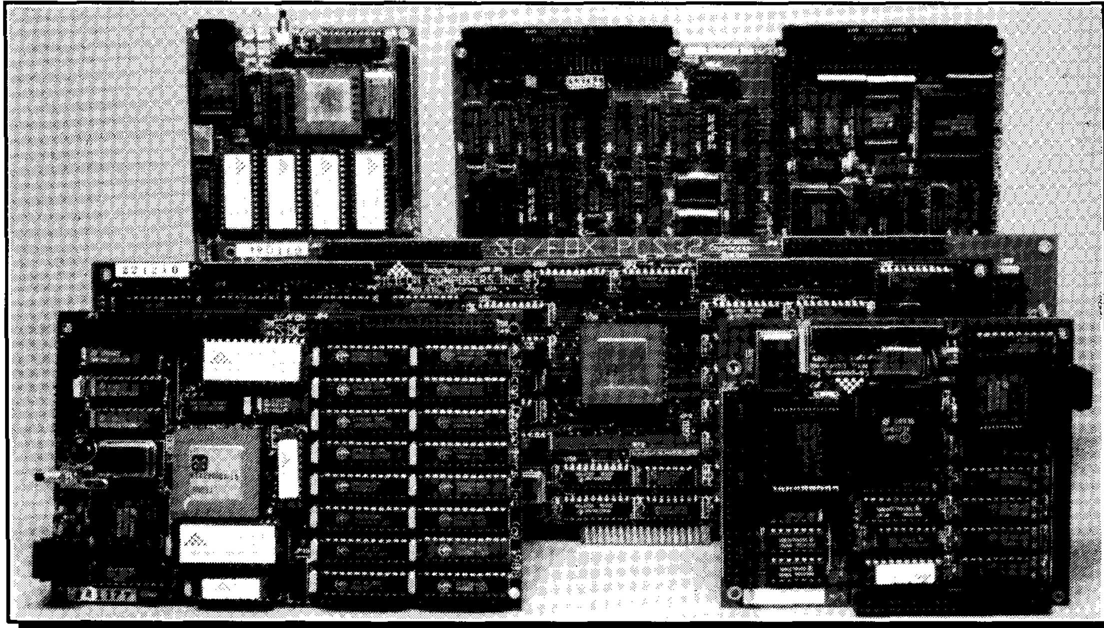


SILICON COMPOSERS INC

FAST Forth Native-Language Embedded Computers

DUP

>R



C@

R>

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

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

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

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

SC32^{tm} 32-bit Forth Microprocessor
- 8 or 10 MHz operation and **15 MIPS** speed.
- 1-clock cycle instruction execution.
- Contiguous 16 GB data and 2 GB code space.
- Stack depths limited only by available memory.
- Bus request/bus grant lines with on-chip tristate.

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

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

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

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

Contents

Features

5 ***Towards a Discipline of ANS Forth Programming***
M. Edward Borasky
Recent on-line discussion about structured programming, multiple entries and exits, finite state machines, and other issues of Forth programming style—including readability and, especially, enhancing the ANS Forth control structures—inspired the author. In this article, he implements the Dijkstra guarded command control structures.

15 ***C-Style Arrays in Forth***
M.L. Gassanenko
A feature of modern processors that is rarely utilized in Forth is *based indexed addressing*. This paper proposes a relevant notation for cell array indexing in Forth. The proposed syntax for the indexed access operations was inspired by C and Algol-68. It supports multi-dimensional arrays, and a similar syntax can be used for bit or double-cell arrays. The paper also shows how analysis of possible name conflicts should be performed.

22 ***Forth in Control — A Window Interface***
Ken Merk
In his last article, the author showed how to build a parallel-printer-port interface with a series of LEDs representing the state of each bit on the port. In this article, he shows how to control that display using Microsoft Windows as the platform. The code builds a graphical user interface—arrays of buttons which can be activated by the mouse to control peripheral devices. This point-and-click environment makes it easy to manipulate the output port.

Departments

4 ***Editorial***

4 ***dot-quote***

27 ***Stretching Forth*** ...... Filters and sponges.

33 ***Forthware*** .... Measuring frequency and sampling time-dependent signals.

Advertisers Index

The Computer Journal 42
FORML Conference 44
FORTH, Inc. 17
Forth Interest Group ... centerfold
Miller Microcomputer Services 14
Silicon Composers 2

Editorial

USENIX, the world-wide UNIX user association, gave a Lifetime Achievement award this year to Wil Baden for "Major Contributions to Software Tools." This is for work Wil did from 1976 – 1981. Wil's involvement with Forth began in 1979. We congratulate Wil and thank him for his ongoing contributions in our own arena, not least of which is his Stretching Forth column in this magazine.

I also want to take this opportunity to thank the world-wide Forth community for its collective contributions to *Forth Dimensions*. Your support—as responsive readers, as writers providing technical content, and as innovators using and refining the language—has ensured its survival even when times have been tough for small, special-interest publications in general.

As editor, I let the collective experience and needs of our writers and subscribers determine the direction and content of the publication. Feedback is very important if we are to keep on track, so let us know how we are doing. We hope you will participate in the magazine, share it with others, and encourage them to join us.

We especially welcome your written contributions to upcoming issues: articles, news, tutorials, and press releases about Forth-related products and events. With your continuing participation, we can look forward to maintaining high quality and to serving the diverse Forth community.

—Marlin Ouverson
editor@forth.org
ouversonm@aol.com

dot-quote

Maybe this is the problem: Forth programmers are too independent. Instead of agreeing on a standard and then helping each other work around the parts of the standard they don't like, everyone wants to design their own standard. This isn't necesarily a bad thing, but if Forth is to become more widespread, Forth programmers need to start thinking about working together instead of going off on their own tangents.

Look at C++... I don't care for it at all, but it's "successful" because the people who use it as their language of choice are willing to work with the limitations of the language (in order to maintain a standard) and write applications instead of trying to write a better version of C++ every month.

—Ken Deboy
glockr@delphi.com

Forth Dimensions
Volume XVIII, Number 4
November 1996 December

Published by the
Forth Interest Group

Editor
Marlin Ouverson

Circulation/Order Desk
Frank Hall

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

Subscription to *Forth Dimensions* is included with membership in the Forth Interest Group at $45 per year ($53 Canada/Mexico, $60 overseas air). For membership, change of address, and to submit items for publication, the address is: Forth Interest Group, 100 Dolores St., suite 183, Carmel, California 93923. Administrative offices: 510-89-FORTH Fax: 510-535-1295

Copyright © 1996 by Forth Interest Group, Inc. The material contained in this periodical (but not the code) is copyrighted by the individual authors of the articles and by Forth Interest Group, Inc., respectively. Any reproduction or use of this periodical as it is compiled or the articles, except reproductions for non-commercial purposes, without the written permission of Forth Interest Group, Inc. is a violation of the Copyright Laws. Any code bearing a copyright notice, however, can be used only with permission of the copyright holder.

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. 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, 100 Dolores St., suite 183, Carmel, CA 93923. Second-class postage paid at Oakland, CA. POSTMASTER: Send address changes to *Forth Dimensions*, 100 Dolores St., suite 183, Carmel, CA 93923."

Towards a Discipline of ANS Forth Programming

M. Edward Borasky
Beaverton, Oregon

1. Motivation

Recently, there has been a fair amount of discussion in *comp.lang.forth* about structured programming, multiple entries and exits, finite state machines, and other issues of Forth programming style. Concerns about readability seem to be foremost, and a number of attempts to enhance the ANS Forth collection of control structures have been posted. The draft proposed ANS standard ([2], section A.3.2.2, pages 136–138) gives an excellent description of the control flow stack and techniques for designing your own control structures, as does Jack Woehr in chapter six of [3]. As an exercise to learn about this useful capability of ANS Forth, and as my contribution to the debate, I have implemented the Dijkstra guarded command control structures in hForth.

2. Structured Programming and the Dijkstra Guarded Command Control Structures

In the early to mid 1970s, there was an explosion of interest in structured programming and the exciting possibility that one could actually prove mathematically that programs were correct. The latter efforts, pioneered by British computer scientist C.A.R. (Tony) Hoare and Dutch computer scientist Edsger W. Dijkstra, grew into a new subfield of computer science, now a major branch of Formal Semantics of Programming Languages. While much of this material is academically oriented and not readable by most working programmers and managers, Dijkstra's *A Discipline of Programming* ([1]) is a happy exception.

The concept of proving one's programs correct in general, and this book in particular, made a profound impression on me. Unfortunately, I don't have space to go into much detail about [1] or the mini-language Dijkstra created to illustrate the concepts. Nor do I have the time to translate the entire book into ANS Forth, although I'm convinced that it would be easier for ANS Forth than for most other languages. Instead, I will focus on the guarded command control structures and my implementation of them in Wonyong Koh's hForth. If you can find a copy of [1], you will find it very rewarding.

The basic syntax of Dijkstra's mini-language is similar to that of Algol and its descendants, such as Pascal. For example, to set the variables *x* and *y* to 3 and 5, Dijkstra would write:

```
x := 3; y := 5
```

using the semicolon as a statement separator. This is read *x becomes 3, then y becomes 5.* This is, of course,

```
3 x ! 5 y !
```

in Forth.

We will define the control structures from the bottom up. At the lowest level we have a *guarded command.* This is simply a Boolean expression followed by a right-arrow followed by one or more statements separated by semicolons. In Dijkstra's language, an example would be:

```
x > y -> x := x - y
```

which is read, *if x is greater than y, then x becomes x minus y.* A Forth programmer would write:

```
x @ y @ > IF y @ NEGATE x +! THEN
```

In the Dijkstra syntax, the statements after the right-arrow are executed only if the condition before the right-arrow, which is called the *guard,* is true. Some variant of this construct appears in nearly all modern programming languages, including Forth as we have just seen. This is the basic building block of the Dijkstra constructs. Figure One shows the flowchart of this simple guarded command.

Next, Dijkstra defines a *guarded command set.* This is a series of guarded commands separated by bars. For example, Dijkstra would write

```
x > y -> x := x - y | y > x -> y := y - x
```

The key word in this definition is *set.* This is a set in the mathematical sense; ordering of the alternatives is not defined. This is an intermediate step on our way to larger constructs, so I will not show a flowchart or a Forth translation.

Now we're ready to define our first real construct. We

Figure One. Flowchart for single guarded command.

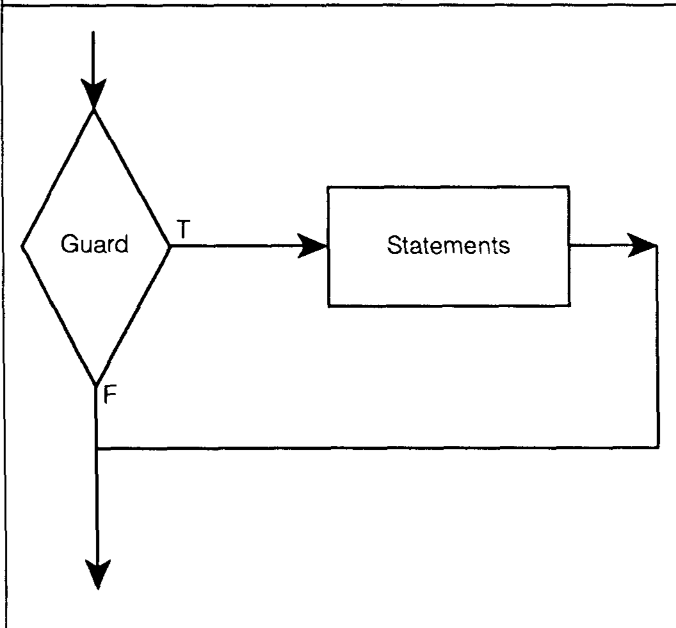
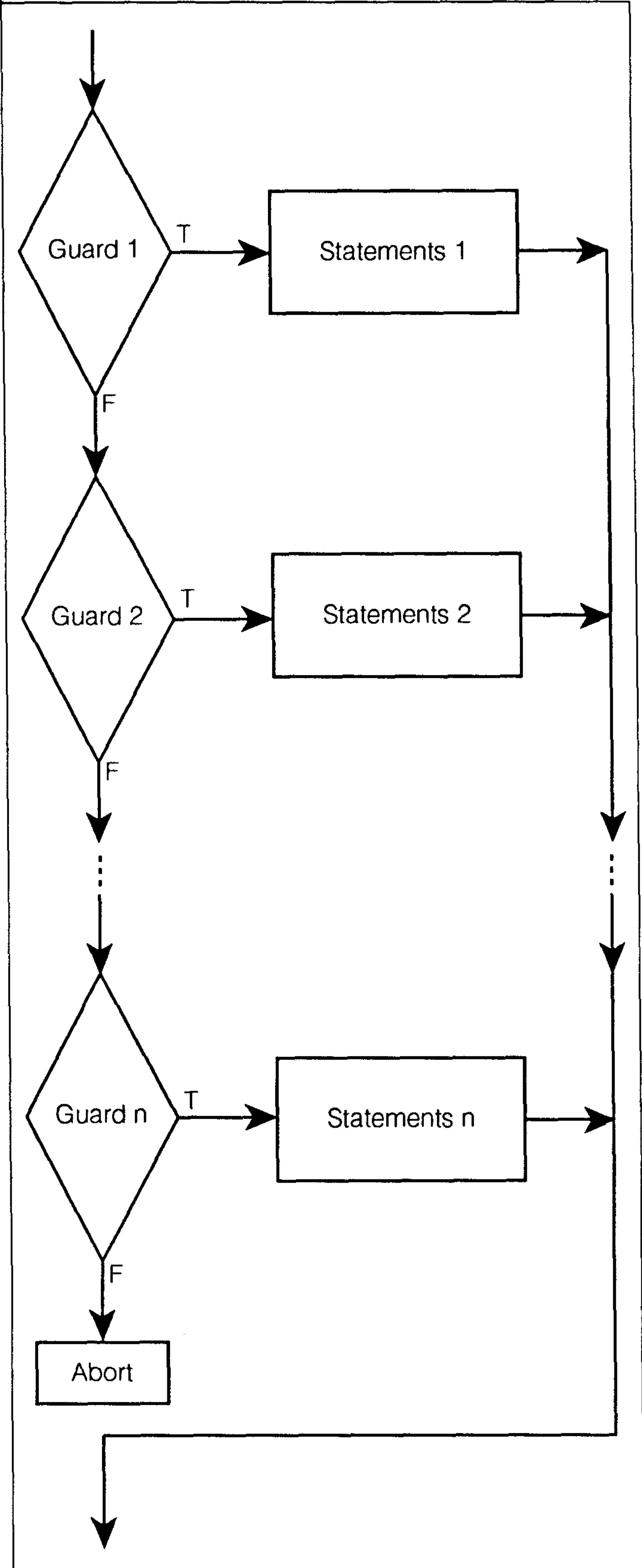


Figure Two. Flowchart for 'if ... fi' construct.



will enclose a guarded command set in a pair of *if ... fi* brackets:

```
if
        x > y -> x := x - y
|
        y > x -> y := y - x
fi
```

which, as written, has a serious flaw!

The exact semantics of this construct are as follows:

> Select one of the true guards and execute the corresponding statements. If none of the guards are true, abort. If more than one guard is true, only one will be selected, but the programmer will not be able to predict or control which one it is!

So, what's the flaw? This program will abort if *x* and *y* are equal! This seems fair enough; the programmer should have known that equality was possible and planned for it.

This construct, as Dijkstra defined it, is nondeterministic. The nondeterminacy is convenient in theoretical work but for most practical programming it is a nuisance. The programmer can, of course, prevent nondeterminacy by assuring that the guards are mutually exclusive. But, as we will see shortly, a different approach is usually taken.

This construct, in its full nondeterministic form, is not present in the widely used programming languages of today. It has, however, been used in academic programming languages. A form of it appeared in the Occam language. The common usage is the *sequential semantics*:

> Test the guards in the order written and execute the statements corresponding to the first true one found.

This form was used in Per Brinch Hansen's little-known Edison language ([6], pages 28–31) and is the form I have implemented.

Figure Two shows the flowchart of the sequential *if ... fi* construct.

Before moving on, let's manually translate the above flawed example into standard ANS Forth using only Core words and the sequential semantics:

```
x @ y @ > IF
        y @ NEGATE x +!
ELSE
        y @ x @ > IF
                x @ NEGATE y +!
        THEN
THEN
```

What happens when *x* and *y* are equal? Nothing; the programmer isn't informed that he forgot that possibility. In addition, this code is ugly. Even if you factor out the tests and statements, replacing them with single-word equivalents, it is still a sequence of nested `IF ... ELSE ... THEN` constructs. And the depth of nesting grows with the number of elements in the guarded command set!

This is exactly the kind of ugliness that the posters in *comp.lang.forth* are complaining about, and rightly so. As a preview of things to come, here's how you would write this in my implementation of the Dijkstra constructs:

```
{IF
        x @ y @ > IF> y @ NEGATE x +!
|IF|
        y @ x @ > IF> x @ NEGATE y +!
FI}
```

It's still flawed, but

1. It's not as ugly. There is a pleasing symmetry to the construct. Moreover, no matter how many alternatives there are, nesting is only one level deep.
2. If *x* and y are equal, my implementation will, in fact, abort.

The final construct is syntactically similar but semantically opposite. Instead of *if* ... *fi* our brackets are *do* ... *od* and the construct is a loop:

```
do
        x > y -> x := x - y
|
        y > x -> y := y - x
od
```

Not only is this program correct, it actually does something useful! Can you guess what it does?

The semantics are:

> If none of the guards are true, do nothing and terminate normally. Otherwise, select one of the true guards nondeterministically and execute the corresponding statements. Then go back to the do and repeat the process until none of the guards are true.

Once again, most practical implementations, including mine, test the guards in the order written and execute the statements corresponding to the first true one found.

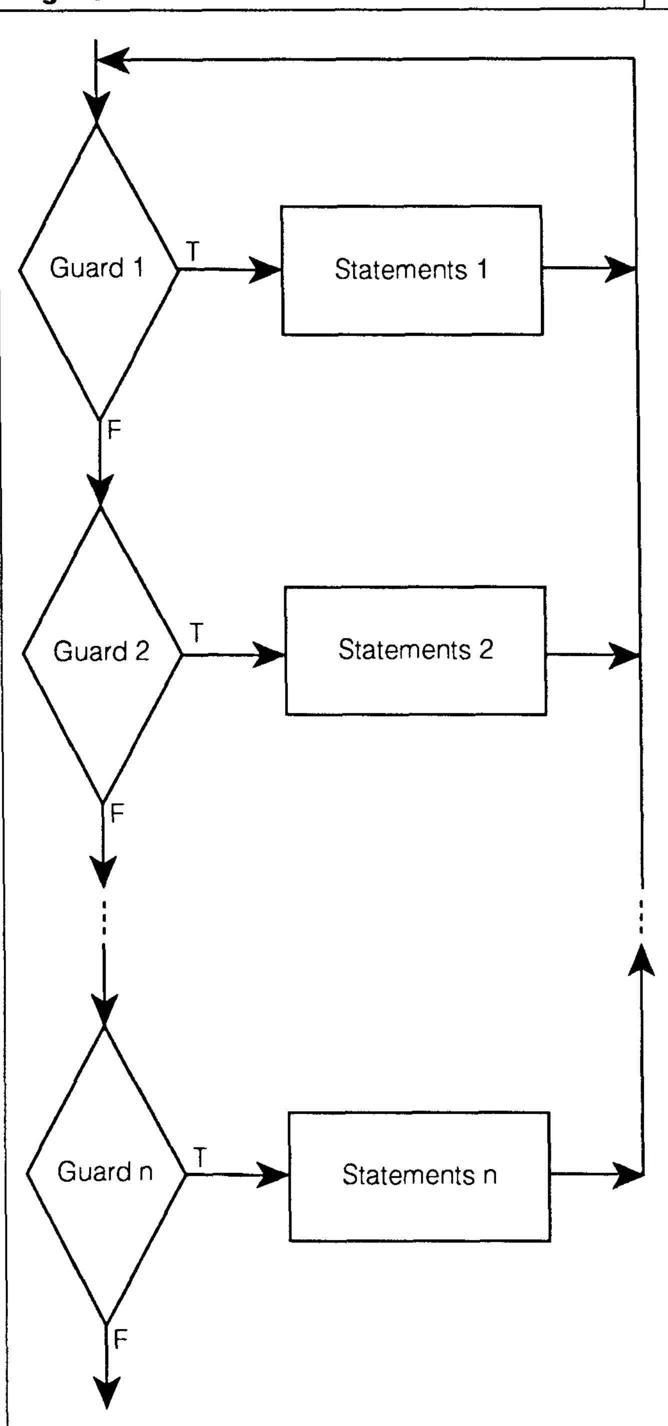
By being clever with the ANS standard words `BEGIN`, `UNTIL`, `WHILE`, `IF`, `THEN`, and others, I'm sure it's possible to duplicate the operation of the sequential form of this construct, just as we were able to duplicate the *if* ... *fi* construct. Even for this simple example, the code is ugly enough that I gave up trying to do it and implemented my own compiling words using a more readable syntax:

```
{DO
        x @ y @ > DO> y @ NEGATE x +!
|DO|
        y @ x @ > DO> x @ NEGATE y +!
OD}
```

Have you guessed what it does yet? Here's a hint: start with
`5 11 * x ! 5 19 * y !`

and simulate it on paper. Remember, the loop terminates when *x* and *y* are equal. Figure Three shows the flowchart

Figure Three. Flowchart for 'do ... od' construct.



of the sequential *do* ... *od* construct.

3. ANS Forth Control Flow Tools

First, let's see how ANS Forth compiles our simplest construct, the guarded command. As you will recall, our example is

```
: GUARDED-COMMAND ( -- )
       x @ y @ > IF \ x is larger than y
               y @ NEGATE x +!
       THEN
;
```

The compiler, started by the colon, starts a dictionary entry for the new word GUARDED-COMMAND. It then compiles each word it encounters in an implementation-defined manner. Figure Four shows schematically what this dictionary entry looks like just after the compiler has finished processing the >.

Then the compiler encounters the IF. We are now inside the diamond on the flowchart (Figure One).

IF is an immediate, compile-only word. So the compiler executes the IF. What IF does here is compile a conditional branch after the >. This branch will look at the flag on top of the stack at run time and, if it is FALSE (all bits zero), the branch will be taken to the point in the code just after the THEN. This corresponds to the down-arrow labeled F. If the flag is TRUE (any bits non-zero), the branch will not be taken and execution will continue with the words after the IF, in this case the code to subtract *y* from *x*. This corresponds to the right-arrow labeled T.

But how does the compiler know where the THEN is located? How does it know how far to branch when the flag is FALSE? It doesn't. So it compiles the branch with an empty spot reserved for the branch target, and places a token called an *orig*, short for "origin," on the control flow stack. This *orig* tells the compiler where to place the branch target when it does find the matching THEN. Figure Five shows the dictionary entry for GUARDED-COMMAND after the compiler has executed the IF.

Compilation continues normally until the compiler reads the THEN. THEN, like IF, is an immediate compile-only word. What does THEN have to do? It doesn't have to generate any code. All it has to do is fill in the target address in the open branch placed in the dictionary by IF, so that the branch points to the current location. How does it know where to find the branch? It gets this information from the *orig* on the control flow stack. This *orig* was placed there by the IF for just this purpose. This process of filling in the branch address and consuming the *orig* is called *resolving the orig*. Figure Six shows the completed guarded command.

Now let's look at another component we'll need: AHEAD. AHEAD is similar to IF; it compiles an open forward branch into the dictionary when encountered by the compiler, and places an *orig* onto the control flow stack for a subsequent THEN to resolve. However, instead of the conditional branch of IF, AHEAD compiles an unconditional branch. We will see AHEAD again, when we

Figure Four. Guarded command before 'IF'.

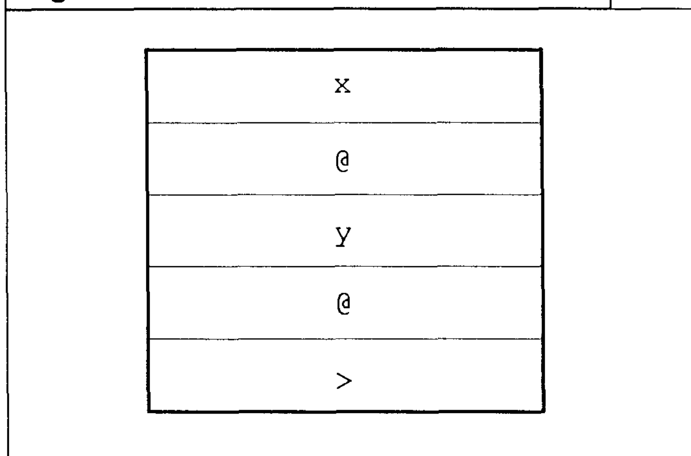
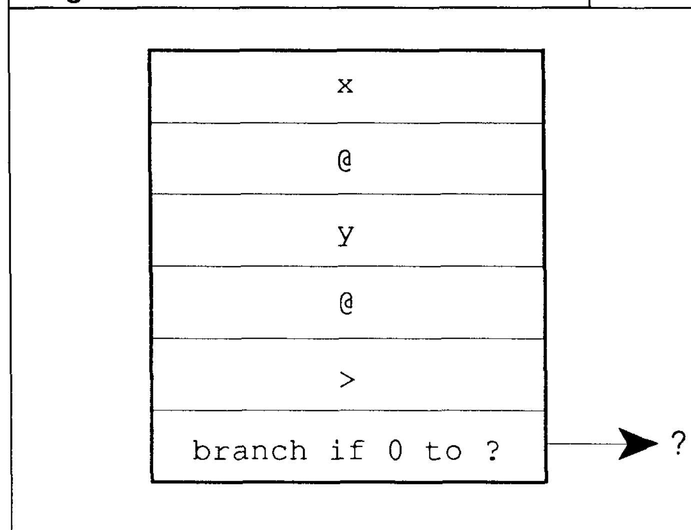


Figure Five. Guarded command after 'IF'.



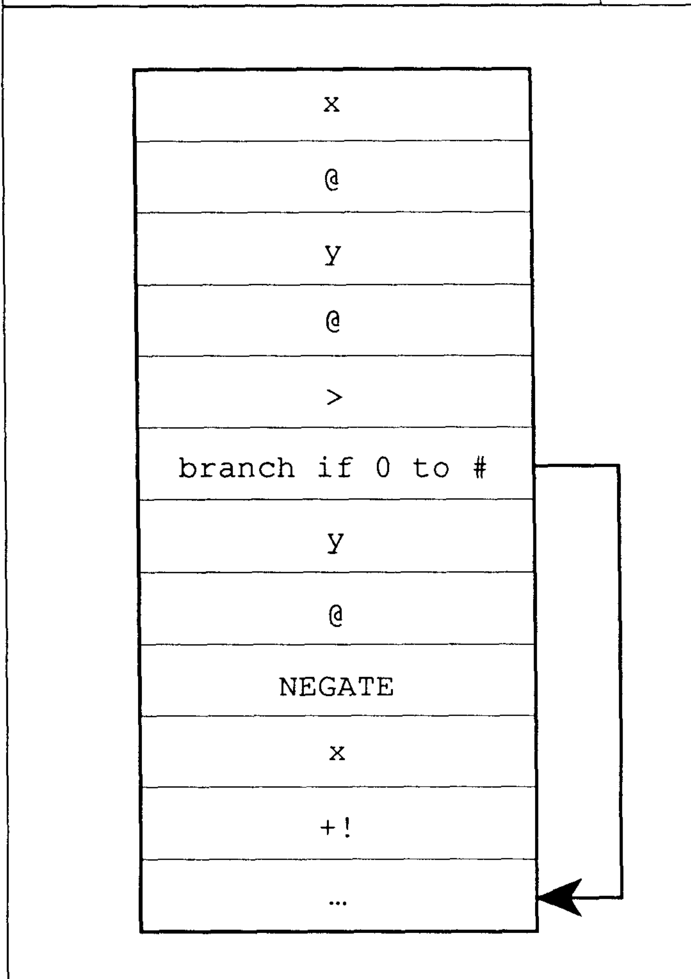
look at the code for | IF |. In summary, we have a forward conditional branch, IF; a forward unconditional branch, AHEAD; and a word that resolves either, THEN.

How does ANS Forth compile loops? We first need a word to mark where the top of the loop is. That word is BEGIN. BEGIN simply puts a token, called a *dest* for "destination," onto the control flow stack. Subsequent branches back to the BEGIN will use this *dest* to know where the target of the backward branch is. Two basic backward branches complete the loop construction set. The unconditional branch is called AGAIN. AGAIN generates an unconditional branch back to the location given by the *dest* on top of the control flow stack, then removes the *dest*. This process is called *resolving the dest*.

The conditional backward branch is called UNTIL. Like IF, the conditional branch is a branch if the flag is FALSE: all zeroes. If the flag is TRUE, the branch is not taken. And like AGAIN, the branch is back to the BEGIN, marked by the *dest* on top of the control flow stack. In summary, we have one word, BEGIN, that creates a *dest*; and two, AGAIN and UNTIL, that resolve one. When we examine my implementation of the Dijkstra constructs, we will see IF, THEN, AHEAD, BEGIN, and AGAIN in action.

Before we walk through the implementation, there are two more words we will need. As we've seen, the ANS

Figure Six. Guarded command after 'THEN'.



standard uses a special stack, called the control flow stack, to keep track of all these *orig* and *dest* tokens. In stack diagrams, this stack is denoted by `C:`. Sometimes, we will need to rearrange these tokens at compile time. The words that do this are `CS-PICK` and `CS-ROLL`. They are analogous to `PICK` and `ROLL` on the data stack.

4. hForth Implementation of the Dijkstra Constructs

hForth ([4]) is a public-domain, extended subset of ANS Forth. The version I used is available from the Taygeta Scientific Web page (ftp://ftp.taygeta.com/pub/Forth/Reviewed/hf86v097.zip). This is version 0.9.7 of hForth, and runs on any 8086 DOS system; I used the HP100LX Palmtop PC ([5]). With minor modifications, this code should run on any ANS Forth system that includes the control flow stack words. In the spirit of Forth, my implementation is a set of *compiling words*; they extend the Forth compiler to compile these control structures as written.

hForth does not define the control flow stack operators `CS-PICK` and `CS-ROLL`. However, hForth uses the data stack as the control flow stack during compilation. As a result, we can define them simply [see listing]. Because the control stack may or may not be the same as the data stack on various Forths, we need to be careful to write the code so it will work either way. We will look at the *if* ... *fi* construct first. You may want to follow along on the flowchart (Figure Two).

First, we define the opening bracket. Dijkstra calls it *if*, which already has a meaning in ANS Forth. Moreover, I wanted something that looked like a bracketing operator, so I picked `{IF`, pronounced "brace-if." As we will see, the `{IF ... FI}` construct will generate an arbitrary number of unresolved *origs* on the control flow stack which don't get resolved until the closing `FI}` is seen. This means we need to count them. All `{IF` does is place a zero on the data stack for this counter.

Now let's look at the right-arrow operator. `->` has a meaning already, so that's out. In addition, I wanted something that reinforced the construct type in the reader's mind, so I picked `IF>`, pronounced "if-arrow." If you're following along on the flowchart, we're inside one of the diamonds corresponding to a guard. At run time, a flag will be on the stack. The `IF>` needs to open up a new conditional branch, just like the standard Forth `IF`. And it needs to count that branch. If the flag is `FALSE`, our branch will skip over the code that follows the `IF>` and proceed to the next guard. Just like the IF in the guarded command, this is the down-arrow labeled `F`. If the flag is `TRUE`, we will take the right-arrow labeled `T` and execute the code following the `IF>`. We won't know the target of this branch until we see the next `|IF|` or the closing `FI}`.

Since, in ANS Forth, the control flow stack may or may not be the data stack, we have to write code that will work either way. So we increment the counter on top of the data stack, move it to the return stack, then insert the desired branch with a `POSTPONE IF` operation. This puts an *orig* on the control flow stack for a later `THEN` to resolve. Then we bring the count back to the data stack.

The *bar* for `{IF ... FI}`, written `|IF|` and pronounced "if-bar," has to do two things. The `|IF|` marks the end of the code associated with the previous guard and the beginning of the next guard. First, we need to compile in an unconditional branch to the end of the construct, which is marked by an as-yet-unseen `FI}`. If you're following along on the flowchart, this is the arrow coming out of the right of the *statements* box. This is an unconditional branch forward to an unknown location, a job for `POSTPONE AHEAD`.

Next, we need to resolve the open conditional branch of the previous guard, so that if the guard is false, control will end up just after the unconditional branch we just compiled. That way, we'll be ready to execute the code for the next guard which follows the `|IF|`. This is the arrow coming out of the bottom of the previous guard's diamond; the branch was compiled in by the preceding `IF>`. There's one small problem—the `POSTPONE AHEAD` covered up the *orig* we need with a new *orig*. `1 CS-ROLL` fixes this, and a `POSTPONE THEN` resolves the open `IF`. Since a new *orig* is created and an old one is resolved, the count does not change. However, we do need to save and restore it.

Now we need to define the closing bracket `FI}`, pronounced "fie-brace." As noted earlier, if none of the guards are true for the `{IF ... FI}`, we consider it a programming error and want to abort. In hForth, we have `CATCH` and `THROW` from the ANS Exception Handling

word set. So I defined a word BAD{IF...FI} ("bad-if-fie"), which will do the aborting. I used -22 for the THROW code; this stands for *control structure mismatch.*

So what does FI} need to do? As usual, we first save the count of open branches that need to be resolved on the return stack. Next, like |IF|, we have to insert a forward branch to the end of the construct to wrap up processing of the code following a true guard. As usual, this is done with POSTPONE AHEAD.

Next, we need to resolve the open forward branch generated by the last guard's IF>. As before, this is done with 1 CS-ROLL POSTPONE THEN. If you're following along on the flowchart, we're on the false branch out of the bottom diamond. Here is where we want to abort, which we do with POSTPONE BAD{IF...FI}.

Remember all those POSTPONE AHEAD operations? All those unconditional branches to the FI} that we compiled in after the code executed following a true guard? All those *origs* sitting on the control flow stack? All of them are now resolved to point to the present location, just after the abort. We retrieve the count from the return stack, then 0 ?DO POSTPONE THEN LOOP does exactly the right number of POSTPONE THEN operations!

For the {DO ... OD} construct, it turns out that we will not need to count open branches; each is resolved by the separating |DO| or the closing OD}. But we do need to place a *dest* on the control flow stack so we know where to branch back to. {DO, pronounced "brace-do," does this with a POSTPONE BEGIN operation.

The right-arrow for the {DO ... OD} construct doesn't have to deal with the count, but there is a *dest* on top of the control flow stack. We want to keep it on top so we always know where it is. Like IF>, DO> ("do-arrow") has to open up a conditional branch with POSTPONE IF. Then we use 1 CS-ROLL > to bring the *dest* back to the top of the control flow stack. On the flowchart (Figure Three), we're in the diamond corresponding to a guard, just like IF>.

The *bar* for the {DO ... OD} construct, |DO| ("do-bar") is very much different from its cousin |IF|. Since |DO| follows code that was executed after a true guard, we will be repeating the loop. On the flowchart, we're on the right-arrow coming out of a *statements* box. We will make an unconditional branch back to the {DO. In ANS Forth terminology, we resolve the *dest* on top of the control flow stack with POSTPONE AGAIN.

There are two tricky parts. First, we need to *copy* the *dest*; we'll need it again for subsequent |DO| operations. 0 CS-PICK makes the copy. Second, hForth keeps track of control structure balance; since we're creating a copy of the *dest* to resolve, we must use the hForth word dest+ to notify the hForth compiler of the extra operator.

Like |IF|, when we get to |DO| there is an open *orig* that needs to be resolved so a false guard will send control to the next guard. We resolve this *orig* with a POSTPONE THEN and everything is done.

Finally, let's look at OD} ("odd-brace"). We only have to do two things. First, the OD} marks the end of the code following the last guard, so we have to compile an unconditional branch back to the {DO just like we did for |DO|. POSTPONE AGAIN does this. Since this is the end of the construct, we don't need to copy the dest; this time, we want to consume it. Second, we have to fill in the target address of the conditional branch compiled by the last DO>. Like FI}, all the guards being false will cause a chain of conditional branches that ends up where we are now. POSTPONE THEN resolves the last *orig*, and we're done. Whew!

5. Testing/Demos

After all this work, we will perform some simple tests to demonstrate our code. First, let's look at a simple example of a correct {IF ... FI}: TEST1. This test simply compares the top two numbers on the stack and prints the comparison that was true. Next, let's see what happens if we accidentally forget that two numbers can be equal: TEST2.

And we close by executing our useful example {DO ... OD} loop: USEFUL.

If you haven't guessed yet, USEFUL is Euclid's algorithm for computing the greatest common divisor of *x* and *y!*

6. Summary

We have seen that it is easy to create custom control structures in ANS Forth. As a practical example, I developed an implementation of the Dijkstra guarded command control structures. These suffice for most of my own control structure needs beyond the ones already provided by ANS Forth. The Dijkstra structures are far more readable than the equivalent code done in terms of the existing ANS Forth control flow operators. And the Dijkstra constructs are an elegant way to express algorithms, as the simple code for the greatest common divisor shows.

I consider this a necessary first step towards the goal of being able to prove Forth programs correct: the ideal is to develop the correctness proof and the code together. An even more ambitious goal is a system for automatically translating specifications into correct Forth code. The supporting academic work has been done, almost always using simple functional programming languages rather than complex real-world languages like C++, Fortran 90, or Common LISP. It seems to me that Forth's simple syntax and semantics provide an opportunity for such a system unavailable to the other languages in common use.

7. References

[1] Dijkstra, Edsger W. (1976), *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, ISBN 0-13-215871-X

[2] ANSI (1993), *Draft Proposed American National Standard Forth* (X3J14 dpANS-6), American National Standards Institute, New York

[3] Woehr, Jack (1992), *Forth: The New Model*, M&T Books, San Mateo, CA, ISBN 1-55851-277-2

[4] Koh, Wonyong (1996), "hForth: A Small, Portable ANS Forth," *Forth Dimensions*, Volume XVIII, Number 2

[5] Borasky, M. Edward (1995), "Forth in the HP100LX," *Forth Dimensions*, Volume XVII, Number 4

[6] Hansen, Per Brinch (1982), *Programming a Personal Computer*, Prentice-Hall, Englewood Cliffs, NJ, ISBN 0-13-730283-5

Downloading: This code can be found as the file "discp.fth" at ftp://ftp.forth.org/pub/Forth/FD/1996

M. Edward Borasky is an applied mathematician and computer scientist who has written software for machines ranging from programmable calculators to massively parallel supercomputers. His interests include computer music, computational finance, computer system performance analysis and, of course, the Forth language. He currently works for a major vendor of turnkey business computers as a UNIX performance guru. Although his desk has been declared a Forth-free zone, his palmtop PC has been granted an exemption. He can be reached at znmeb@teleport.com or at http://www.teleport.com/~znmeb (his home Web page).

Listing. discp.f: Dijkstra's guarded command control structures.

```
\ Dijkstra Guarded Command Control Structures
\ M. Edward Borasky
\ 03-AUG-96
\
\ This code has been tested with both hForth 0.9.7 and ZENForth.
\ To compile for hForth, type
\
\    0 CONSTANT ZENForth
\
\ To compile for ZENForth, type
\
\    1 CONSTANT ZENForth
\
\ Then type
\
\    BL PARSE DISCP.F INCLUDED
\
\ These words were designed using Wonyong Koh's hForth 0.9.7,
\ They should work with minor modifications on any ANS Forth
\ system providing the words listed below.

\ Environmental dependencies:
\
\ Requires AGAIN from the CORE EXT word set
\ Requires AHEAD from the TOOLS EXT word set
\ Requires CS-PICK from the TOOLS EXT word set
\ Requires CS-ROLL from the TOOLS EXT word set
\ Requires PICK from the CORE EXT word set
\ Requires ROLL from the CORE EXT word set
\ Requires THROW from the EXCEPTION word set
\ Requires hForth word dest+ or equivalent
\ Requires hForth word COMPILE-ONLY or equivalent
\ Requires .( from CORE EXT word set (test sequence only)

\ hForth does not have CS-PICK or CS-ROLL.  However, hForth
\ uses the data stack as control flow stack, so they can be
\ defined simply:

: CS-PICK PICK ;

: CS-ROLL ROLL ;

\ hForth has the capability to flag a word COMPILE-ONLY.  On other systems,
\ COMPILE-ONLY can be ignored by defining it as follows:
```

(Continues on next page.)

```
ZENForth [IF] \ ZENForth compatibility
        : COMPILE-ONLY ;
[THEN]

: {IF \ start a conditional
        ( -- 0 )

        0 \ put counter on stack
; COMPILE-ONLY IMMEDIATE

: IF> \ right-arrow for {IF ... FI}
        ( count -- count+1 )
        ( C: -- orig1 )

        1+ >R \ increment and save count
        POSTPONE IF \ create orig1
        R> \ restore count
; COMPILE-ONLY IMMEDIATE

: |IF| \ bar for {IF ... FI}
        ( count -- count )
        ( C: orig ... orig1 -- orig ... orig2 )

        >R \ save count
        POSTPONE AHEAD \ new orig
        1 CS-ROLL \ old orig to top of CFStack
        POSTPONE THEN \ resolve old orig
        R> \ restore count
; COMPILE-ONLY IMMEDIATE

: BAD{IF...FI} \ abort if there is no TRUE condition
        ( -- )

        CR ." {IF ... FI}: no TRUE condition" CR \ error message
        -22 THROW \ 'control structure mismatch'
;

: FI} \ end of conditional
        ( count -- )
        ( C: orig1 ... orign -- )

        >R \ save count
        POSTPONE AHEAD \ new orig
        1 CS-ROLL \ old orig
        POSTPONE THEN \ resolve old orig

        \ if we got here, none of the guards were TRUE
        \ so abort
        POSTPONE BAD{IF...FI} \ compile the abort
        R> \ restore count

        0 ?DO \ resolve all remaining origs
                POSTPONE THEN
        LOOP
; COMPILE-ONLY IMMEDIATE
```

```
: {DO \ start a loop
        ( C: -- dest )

        POSTPONE BEGIN \ create dest
; COMPILE-ONLY IMMEDIATE

: DO> \ right arrow for {DO ... OD}
        ( C: dest -- orig1 dest )

        POSTPONE IF \ create orig
        1 CS-ROLL \ bring dest back to top of CFStack
; COMPILE-ONLY IMMEDIATE

\ hForth uses the word 'dest+' to count open destinations.  For other environments,
\ there may be a similar word.

ZENForth [IF]
        : dest+ 1 bal +! ;
[THEN]

: |DO| \ bar for {DO ... OD}
        ( C: orig1 dest -- dest )

        0 CS-PICK \ copy the dest
        POSTPONE AGAIN \ resolve the copy
        dest+ \ hForth control structure operation
        1 CS-ROLL \ old orig
        POSTPONE THEN \ resolve old orig
; COMPILE-ONLY IMMEDIATE

: OD} \ end of loop
        ( C: orig dest -- )
        POSTPONE AGAIN \ resolve dest
        POSTPONE THEN \ resolve orig
; COMPILE-ONLY IMMEDIATE


\ Simple test words

: TEST1 \ print the relationship between 'x' and 'y'
        ( x y -- )

        {IF
                2DUP = IF> CR ." = "
        |IF|
                2DUP > IF> CR ." > "
        |IF|
                2DUP < IF> CR ." < "
        FI}
        2DROP
;

\ execute TEST1 for all three combinations

CR .( 5 0 TEST1 )
5 0 TEST1

CR .( 5 5 TEST1 )
5 5 TEST1
```

(Continues on next page.)

```
CR .( 0 5 TEST1 )
0 5 TEST1

: TEST2 \ deliberately erroneous test case -- 'equal' case left out!
        ( x y -- )

        {IF
                2DUP < IF> CR ." < "
        |IF|
                2DUP > IF> CR ." > "
        FI}
        2DROP
;

CR .( Since TEST2 aborts if 'x' and 'y' are equal, we will )
CR .( test TEST2 later; first we will compile and test USEFUL )

\ define arguments
VARIABLE x  5 6553 * x !
VARIABLE y 6551 5 * y !

: USEFUL \ sets both 'x' and 'y' to GCD(x, y)
        ( -- )

        {DO
                x @ y @ > DO> y @ NEGATE x +!
        |DO|
                y @ x @ > DO> x @ NEGATE y +!
        OD}
;

CR .( Before: x, y = ) x @ . y @ . CR
CR .( USEFUL ) USEFUL
CR .( After: x, y = ) x @ . y @ . CR

CR .( Now we'll test TEST2 )

CR .( 5 0 TEST2 )
5 0 TEST2

CR .( 0 5 TEST2 )
0 5 TEST2

CR .( 5 5 TEST2 )
5 5 TEST2

\ That's all, folks!!
```


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

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

MMSFORTH System Disk from $179.95
Modular pricing — Integrate with System Disk only what you need:

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

MMS FORTH

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

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

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

MMSFORTH V2.4 System Disk from $179.95
Needs only 24K RAM compared to 100K for BASIC, C, Pascal and others. Convert your computer into a Forth virtual machine with sophisticated Forth editor and related tools. This can result in 4 to 10 times greater productivity.

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

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

and a little more!

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

Call for free brochure, technical info or pricing details.

C-Style Arrays in Forth

M.L. Gassanenko
St. Petersburg, Russia

Abstract

This paper proposes a C-like notation for cell array indexing in Forth. Although the idea is not new, the notation seems to be felicitous and might be included in the next standard. It can support multi-dimensional arrays, and a similar syntax may be used for bit or double-cell arrays. The paper also shows how analysis of possible name conflicts should be performed.

Introduction

One feature of (not too) modern processors that is rarely utilized by (even modern) Forth is *based indexed addressing*. There have been several approaches to array accessing but, so far, none of them has been considered felicitous enough to be included in the standard.

Array elements are usually accessed via `2* + @` (unless the programmer prefers to resort to assembler). There is also a traditional array implementation in which the operation of indexing is bound with and hidden in the array name (an array is a function that takes indexes from the stack and leaves the address of an element), but it has not become a *de facto* standard. The third way, also no *de facto* standard, uses words like `[]CELL` and the only difference of the proposed syntax from it is in better naming. This paper shows how one can implement multi-dimensional arrays using this idea.

The proposed syntax for the indexed access operations was inspired by (almost borrowed from) C and Algol-68. One is only sorry that this syntax did not appear 15 years ago.

Specifications

`[]` (n a-addr -- x) "brackets"
EXPERIMENTAL
x is the value stored into the *n*th cell of the cell array starting at *a-addr*. The array cells are numbered starting from zero. Semantically equivalent to:
`SWAP CELLS + @`

`[]!` (x n a-addr --) "brackets-store"
EXPERIMENTAL
Store the value *x* into the *n*th cell of the cell array starting at *a-addr*. The array cells are numbered starting from zero. Semantically equivalent to:
`SWAP CELLS + !`

`[]^` (n a-addr1 -- a-addr2) "brackets-pointer"
EXPERIMENTAL
Add the size in address units of *n* cells to ***a-addr1***, giving *a-addr2*. Semantically equivalent to:
`SWAP CELLS +`

Note: This is a proposal for the next Forth standard.

Implementation

What follows is an F-PC implementation of these words:

```
CODE [] ( index array -- value )
    pop bx
    pop di
    shl di
    push 0 [bx+di]
    next c;

CODE []! ( value index array -- )
    pop bx
    pop di
    shl di
    pop 0 [bx+di]
    next c;

CODE []^ ( index array -- address )
    pop bx
    pop di
    shl di
    add bx, di
    push bx
    next c;
```

On a '386 Forth that uses in-lining and keeps the data stack top in EBX, the code substituted for `[]` may look like this:

```
    POP EAX
    MOV EBX, [EBX] [4*EAX]
```

which is much better than the:

```
XCHG EBX, [ESP]      \ SWAP
SHL  EBX, # 2        \ CELLS
POP  EAX             \ +
ADD  EBX, EAX        \ +, continued
MOV  EBX, [EBX]      \ @
```

which we would have as the result of substituting `SWAP CELLS + @` in line.

Multi-Dimensional Arrays

This section shows that multi-dimensional arrays equally can be implemented this way.

A multi-dimensional array is implemented as an array of arrays. Fetching, storing, and pointing to a two-dimensional array element look like this:

```
    j i X [] []      \ X[i][j]
... j i X [] []!     \ X[i][j] = ...
    j i X [] []^     \ &X[i][j]
```

Here, *X* is an array containing addresses of arrays of cells. These cell arrays should not necessarily be of the same size. No index range checking is performed, though. The word `ARRAY,` creates an array of *n* elements *x0* ... *x[n-1]* and returns its address:

```
: array, ( x0 x1 ... x[n-1] n -- addr )
   ( align )  here >r
   ?dup
   if
         0 swap 1-
         do
                  i roll ,
        -1 +loop
   then
   r>
;
```

Although the usefulness of this word is restricted by the maximal stack depth, it enables us to create arrays of (sub)arrays, and these subarrays may have different lengths.

Once the subarrays can have different lengthes, we may wish to be able to determine them. Provided that all the subarrays are created by `ARRAY,` immediately—one after another—and that the last top-level array element is followed by a "dummy" pointer, the word `[]LEN` (i a -- l) given below returns the length of the *i*th subarray of the top-level array *a*.

```
code []len ( 1index array -- 2length )
      pop bx
      pop di
      shl di
      mov ax, 2 [bx+di]      \ address of the (i+1)-th subarray
      sub ax, 0 [bx+di]      \ minus address of the i-th subarray
      shr ax                 \ gives i-th subarray length in cells
      push ax
      next c;
```

The "dummy" pointer is the address of the cell that follows the last array element. We can consider it a zero-length subarray. If it is missed, the word `[]LEN` will not calculate the length of the last subarray.

Assessing the Multi-Dimensional Array Implementation

Here we compare the array-of-arrays implementation with the more traditonal one where index calculation involves multiplication by the number of columns.

The array-of-arrays implementation does not necessarily require more memory. For example, if we manipulate matrixes of a special form, say, symmetric, this technique almost halves the amount of memory required.

This implementation also works faster because, even on a '486, multiplication is slower than memory fetch. Probably, this consideration is not of too much importance, though.

Some Examples

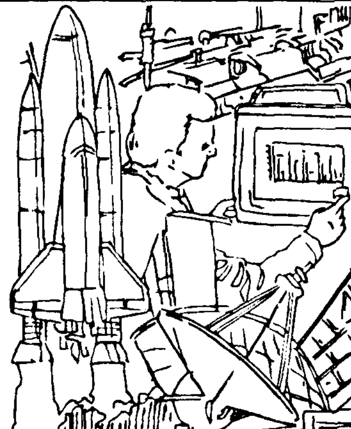
These examples scarcely need comments. The screen output is shown in Listing Two.

```
: .ARRAY ( array len -- )
   ." [ "
   0 ?DO   I OVER [] .   LOOP
   ." ] " DROP
;
: .2ARRAY ( array n_rows -- )
   CR ." [ "
   0 ?DO
     I OVER []  I PLUCK []LEN  CR .ARRAY
   LOOP
   ." ] " DROP
;

\ The last, delimiter, string of array
\ is needed for []len to calculate the
\ length properly
1  2  3  4  5   5 array,
6  7  8  9      4 array,
10 11 12        3 array,
                0 array,
                     4 array, constant x
0 0 x [] [] .
2 0 x [] [] .
25 1 2 x [] []!
1 2 x [] [] .
1 2 x [] []^ @ .
0 x []len .
2 x []len .
x 3 .2array
```

(Text continues on page 18.)

Listing One. arrays.seq

```
\ Arrays for Forth by M.L.Gassanenko
autoeditoff
CODE [] ( index array -- value )
     pop bx
     pop di
     shl di
     push 0 [bx+di]
     next c;

CODE []! ( value index array -- )
     pop bx
     pop di
     shl di
     pop 0 [bx+di]
     next c;

CODE []^ ( index array -- address )
     pop bx
     pop di
     shl di
     add bx, di
     push bx
     next c;

: array, ( x0 x1 ... x[n-1] n -- addr )
   ( align )  here >r
   ?dup
   if
         0 swap 1-
         do
                  i roll ,
        -1 +loop
   then
   r>
;

code []len ( 1index array -- 2length )
     pop bx
     pop di
     shl di
     mov ax, 2 [bx+di]
     sub ax, 0 [bx+di]
     shr ax
     push ax
     next c;

\ \s Auxiliary tools, rather examples
: .ARRAY ( array len -- )
   ." [ "
   0 ?DO   I OVER []  4 .R SPACE   LOOP
   ." ] " DROP
;
: .2ARRAY ( array n_rows -- )
   CR 6 SPACES ." ["
   0
   ?DO              ( array )
     I OVER []      ( array array[i] )
     I PLUCK []LEN ( array array[i] len[i] )
         CR  9 SPACES  .ARRAY
     LOOP
     CR 6 SPACES ." ] " DROP
;

\ \s Some examples
showlines
\ The last, delimiting, subarray is
\ needed for []len to
\ calculate the length properly
1  2  3  4  5   5 array,
6  7  8  9      4 array,
10 11 12        3 array,
                0 array,
                   4 array, constant x
0 0 x [] [] .
2 0 x [] [] .
0 2 x [] [] .
1 2 x [] [] .
25 1 2 x [] []!
1 2 x [] [] .
1 2 x [] []^ @ .
0 x []len .
1 x []len .
2 x []len .
x 3 .2array

off> listvar
```

From NASA space systems to package tracking for Federal Express...

chipFORTH

...gives you maximum performance, total control for embedded applications!

- Total control of target kernel size and content.
- Royalty-free multitasking kernels and libraries.
- Fully configurable for custom hardware.
- Compiles and downloads entire program in seconds.
- Includes all target source, extensive documentation.
- Full 32-bit protected mode host supports interactive development from any 386 or better PC.
- Versions for 8051, 80186/88, 80196, 68HC11, 68HC16, 68332, TMS320C31 and more!

Go with the systems the pros use... Call us today!

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

Listing Two. Screen output of the F-PC program shown in Listing One. The message about `[]` is displayed because `[]` is defined in the `ASSEMBLER` vocabulary.

```
E:\TO-SEND>f arrays.seq ok

 [] isn't unique
   70 \ The last, delimiting, subarray is needed for []len
   71 \ to calculate the length properly
   72 1  2  3  4  5    5 array,
   73 6  7  8  9       4 array,
   74 10 11 12         3 array,
   75                  0 array,
   76                                   4 array, constant x
   77 0 0 x [] [] .  1
   78 2 0 x [] [] .  3
   79 0 2 x [] [] .  10
   80 1 2 x [] [] .  11
   81 25 1 2 x [] []!
   82 1 2 x [] [] .  25
   83 1 2 x [] []^ @ .  25
   84 0 x []len .  5
   85 1 x []len .  4
   86 2 x []len .  3
   87 x 3 .2array
      [
         [    1    2    3    4    5 ]
         [    6    7    8    9 ]
         [   10   25   12 ]
      ]
   88
   89 off> listvar
```

Bit, Double-Cell, and Other Arrays

A similar syntax may be used to handle bit arrays.

`BIT[]` (u addr -- b) "bit-brackets"
EXPERIMENTAL
b is the value stored into the *n*th bit of the bit array starting at *addr*. The array bits are numbered starting from zero. The most significant bit has the largest number. The number of bits in an address unit is system dependent.

`BIT[]!` (x u a-addr --) "bit-brackets-store"
EXPERIMENTAL
Store the low bit of *x* into the *n*th bit of the bit array starting at *addr*.

`BITS` (u1 -- u2) "bits"
EXPERIMENTAL
u2 is the minimal size in address units of a memory area that contains at least *n* bits.

The specifications for the double-cell indexing words are evident:

```
D[]       ( n a-addr -- d )
D[]!      ( d n a-addr -- )
D[]^      ( n a-addr1 -- a-addr2 )
```

The number *n* above is the number of the two-cell array element we want to access.

In systems with 16-bit characters, it may be desirable to have character-array operations as well.

Implementation of all these words is a good exercise for a novice, studying either Forth or assembler. A good name for a double-cell `[]LEN` analog is `D[]LEN`.

Consistency

In this section, we ascertain that introduction of the proposed new names into the standard does not lead to naming problems.

We have introduced two new language elements: `[]` "indexing" and `^` "address." (A sequence, or a set, of characters used in a name and having a meaning for programmers we call a *language element*; for example, the name `CHAR+` consists of two language elements: `CHAR` and `+`).

The symbol ^ has not been used in the standard before. There is an old tradition to indicate "address" by `'` (tick), but in the modern standard `'` (tick) means only "execution token."

The symbol `[]` has not yet been used, but `[` and `]` usually denote state-switching or immediacy. This does not lead to naming conflicts, because `[` and `]` have never been used one immediately after another. The only

imaginable candidate on the [] name is:

```
: [] ' EXECUTE ; IMMEDIATE
```

which we can name [EX] or [EXEC] if we will need it.

(To illustrate importance of such analysis, we can give the following example. In an ANS Forth system we can define:

```
4 CONSTANT CELL
: CELLS CELL * ;
: #CELLS CELL + 1- CELL / ;
```

but we cannot define

```
1 CONSTANT CHAR
: CHARS CHAR * ;
: #CHARS CHAR + 1- CHAR / ;
```

and get a standard system, because, according to the standard, CHAR means "obtain a character from the input stream" while CHARS means "multiply by the size of a character.")

The name of the word [] that may be used to fetch a data address contains no mention of the data size. This makes the notation more natural: [] [] or [] D[] looks better than []CELL []CELL or []CELL []DOUBLE because the size of data is mentioned at most once, and this is the size of data that we want to access. The "size specifiers" ("D" and "BIT" in D[] and BIT[]) are placed before the "operation specifier" [] to keep the same style as in other such words, e.g., C@, C!, and CELL+. Again, the "CELL" address specifier is omitted because the size of the stack element is the default for operations that move data between stack and memory (e.g., @ and ! work with one cell, but they are not named CELL@ and CELL!).

Optional Range Checking

The most evident approach to this is to store the array length into the -1st cell and to redefine the array words to use it to perform range checking. This is shown in Listings Three and Four. Remember that such redefinition enables range checking only for words that get compiled after it; already-compiled words will still use the "unprotected" version. Although the naming issue is always up to the programmer's taste, it may be recommended to redefine the array words locally (within a module vocabulary), and to use their original, "unprotected" versions under some other names.

Conclusion

The array indexing words presented here enable Forth to utilize the based indexed addressing present on most processors. The words [], []!, and []^ are offered for inclusion in the next standard.

This syntax should have appeared 10–15 years ago.

Exercises for the Novice

Although this paper is a proposal, it inspires some good exercises for a novice who is already familiar with Forth but does not have much practice.

1. Implement the words [], []!, []^, and []LEN in Forth.

2. Implement the word 0ARRAY, (n -- addr) which creates an array of *n* cells, initializing it with zeroes. For example,

```
4 0ARRAY,
```

must be equivalent to

```
0 0 0 0  4 ARRAY,
```

3. Implement the double-cell array words D[], D[]!, D[]^, and D[]LEN and the words DARRAY, and 0DARRAY,.

4. Implement the bit array words (in Forth or assembler; the latter is easier).

5. Implement the word BITARRAY, (bit[n-1] ... bit[0] n -- addr) which creates a bit array and initializes it.

6. Implement range checking for double-cell arrays. Double-cell arrays must be accessible as both double- or single-cell arrays, and the range checking must work in both cases.

7. Create a range-checking scheme for bit arrays.

8. Take the array words as a basis and extend the array concept to allow arrays with indexes starting from an arbitrary number (not necessarily from zero); i.e., it must be possible to create, for example, an array for which indexes would be numbers -5 ... 4. You may use any approach, but do not redefine the word [] and the others. Be careful to choose good names.

Code begins on next page...

Downloading: This code can be found as the files arrays.seq, arrays.lst, acheck.seq, and acheck.lst at ftp://ftp.forth.org/pub/Forth/FD/1996

M.L. Gassanenko graduated from St. Petersburg University's Department of Applied Mathematics and Control Processes in 1992, and is now a post-graduate student at the St. Petersburg Institute of Informatics and Automitazation, at the Russian Academy of Sciences. He can be reached at gml@ag.pu.ru or at mlg@iias.spb.su via e-mail.

Listing Three. Range checking for the arrays.

```
needs arrays.seq
\ The -1st array element contains the number of elements in the array.

\ redefine ARRAY, to lay down such -1st element.
: array, ( a[n-1]...a[0] n -- )
    ( align ) dup ,
    array,
;
\ The checking word to be used with []
: _? ( index arr -- )  \ ensure that the index is correct
    2dup -1 swap [] <
    if  over 0< 0=
        if exit then
    then
    cr ." *** invalid index " over . ." for the array at " dup u. cr
;

\ To enable range checking, we can either globally redefine [] ,
\ or redefine it in the module's dictionary,
\ or find a new name (e.g. /[] ) for the protected version of [] ,
\ or redefine [] and use /[] for the original version of [] ,
\ or use _? every time we need a check.
\
\ For example, we can define
\ : /[] _? [] ;                \ if we want /[] to do range checking, and
\ : /[] ?comp compile [] ; immediate   \ if we want it to compile []

: []len  ( index2 2array -- len1 )
    _?
    2dup -1 swap [] 1- =
    if
        cr ." *** []LEN does not work for the last ( " over .
        ." ) subarray (of array at " dup u. ." )" cr
    then
    []len 1-
;
: []     _? [] ;
: []!    _? []! ;
: []^    _? []^ ;
\ NB: we have to recompile .ARRAY and .2ARRAY as well,
\ unless we want them to show 1 extra cell beyond each line.

\ \s Examples
showlines
\ 1-dimensional array:
11 22 33 44   4 array,  constant y
-1 y [] .
0 y [] .
1 y [] .
3 y [] .
4 y [] .

\ 2-dimensional array
\ The last, delimiting, subarray is needed for []len to calculate the length properly
\ NB: we use the same syntax but *different* versions of array words.
1  2  3  4  5   5 array,
6  7  8  9      4 array,
10 11 12        3 array,
                0 array,
                          4 array, constant x
0 0 x [] [] .
-2 0 x [] [] .
0 -2 x [] [] .
-1 -2 x [] [] .
0 x []len .
-1 x []len .
3 x []len .
4 x []len .

off> listvar
```

Listing Four. The screen output of the F-PC program shown in Listing Three.

```
The array words are redefined to perform range checking.

ARRAY, isn't unique
[]LEN isn't unique
[] isn't unique
[]! isn't unique
[]^ isn't unique
   48 \ 1-dimensional array:
   49 11 22 33 44   4 array,  constant y
   50 -1 y [] .
*** invalid index -1 for the array at 33595
4
   51 0 y [] .  11
   52 1 y [] .  22
   53 3 y [] .  44
   54 4 y [] .
*** invalid index 4 for the array at 33595
-28695
   55
   56 \ 2-dimensional array
   57 \ The last, delimiting, subarray is needed for []len
   58 \ to calculate the length properly
   59 \ NB: we use the same syntax but *different* versions of array words.
   60 1  2  3  4  5   5 array,
   61 6  7  8  9      4 array,
   62 10 11 12        3 array,
   63                 0 array,
   64                           4 array, constant x
X isn't unique
   65 0 0 x [] [] .  1
   66 -2 0 x [] [] .
*** invalid index -2 for the array at 33610
-31941
   67 0 -2 x [] [] .
*** invalid index -2 for the array at 33642
8397
   68 -1 -2 x [] [] .
*** invalid index -2 for the array at 33642

*** invalid index -1 for the array at 0
8238
   69 0 x []len .  5
   70 -1 x []len .
*** invalid index -1 for the array at 33642
16802
   71 3 x []len .
*** []LEN does not work for the last ( 3 ) subarray (of array at 33642 )
28351
   72 4 x []len .
*** invalid index 4 for the array at 33642
1226
   73
   74 off> listvar
```

Forth in Control: A Window Interface

Ken Merk
Langley, British Columbia, Canada

In my last article, "Forth in Control" (*FD* XVII/2), we built a parallel-printer-port interface using a series of LEDs to represent the on/off state of each bit on the port. We named all of the eight lines and assigned each a number according to its binary weighting on the port. The interface simulated a machine controller, so we named each output line after the device it was controlling:

```
DECIMAL                          \ Binary weight
  1     CONSTANT FAN             \ 00000001
  2     CONSTANT DRILL           \ 00000010
  4     CONSTANT PUMP            \ 00000100
  8     CONSTANT SPRINKLER       \ 00001000
 16     CONSTANT HEATER          \ 00010000
 32     CONSTANT LIGHT           \ 00100000
 64     CONSTANT MOTOR           \ 01000000
128     CONSTANT VALVE           \ 10000000
```

Tom Zimmer's F-PC was used to make words that would control each bit individually, so we could turn on or off any device we wanted:

```
MOTOR  >ON        \ turn motor ON
FAN    >OFF       \ turn fan OFF
```

In this article, we will use the same LED display interface attached to the parallel printer port, but we will control it using Windows as the platform. LMI's WinForth for Microsoft Windows will be used to create a graphical interface consisting of two arrays of command buttons which can be activated by the mouse to control each device. This creates a point-and-click environment, which makes it quick and easy to manipulate the output port. On/off buttons will be created for each output device, plus another button array with special functions to control groups of devices.

WinForth is a 16-bit system, but applications can run under Windows 3.1 and Windows 95. A 32-bit version of WinForth is currently under development. To test drive LMI's WinForth, a shareware version is available from Laboratory Microsystems' BBS at 310-306-3530 or from some Forth BBSs, one of which is Kenneth O'Heskin's Art of Programming at 604-826-9663. This version is equivalent in capabilities and performance to the retail WinForth, except it does not let you create end-user applications.

DOS vs. Windows

Most DOS programs consist of code written in a sequentially driven manner. Input data must be entered in a specific order to match the flow of the program. This puts a high priority on the order in which the job must be performed.

Windows is an event-driven operating system which allows data to be entered in whatever order seems appropriate. Whenever an event occurs, such as a mouse click or a keypress, Windows notifies the application about the event by sending it a message. A message is a 16-bit, unsigned value which is assigned to a symbolic constant that starts with the letters `WM_`. A procedure within the application intercepts these messages and responds to them. This procedure is called a Message Handler, in which the programmer can code what action has to be taken depending on the message received.

Source Code Overview

In the accompanying source code, FCONTROL.4TH, we create a pop-up modal dialog box containing two keyboards, each with an array of command push-buttons. The dialog box will have a caption bar, which makes the window movable, and a system menu to close the window. A modal dialog box disables the parent window and does not allow you to click or type anywhere outside the dialog box until it is closed.

Clicking on the push-buttons with the mouse will send `WM_COMMAND` messages to the dialog message handler, `CONTROLDLGPROC`, which will process them through the `DO.BUTTON` case statement to activate the appropriate output device. The word `MAIN` is executed to display the dialog and direct all messages resulting from user interaction with the dialog to the dialog message handler, thus running the program. To quit the program, double-click the system menu box, click on the Quit push-button, or press Esc to close the dialog box.

Each control (push-button) within a dialog box will be

given its own control ID. This ID number is assigned to a symbolic constant that starts with the letters `ID_`. This makes the source code easier to read. Because text controls do not send messages back to the message handler, their IDs are set to -1.

The `WM_COMMAND` message is sent to the dialog message handler by the controls (push-buttons) in the dialog box when clicked. The dialog box message handler will check `WM_COMMAND` messages for the control identifier of the push button. When it finds this identifier, which is in the message's wParam parameter, the handler knows which button was pushed and can carry out the corresponding task using the `DO.BUTTON` case statement.

`WM_INITDIALOG` is sent to a dialog box upon the box's first activation, but before it is made visible. In response to this message, a dialog box procedure will initialize each of the dialog box controls to the correct initial state. In our case, no initialization is needed, so a True is returned—verifying that we processed the message and causing Windows to set the focus on the first button created in the dialog box template (Fan On).

To run WinForth from the Program Manager, click on File, then on Run. Select the proper path and type FORTH.EXE, then click on OK. WinForth should load and run. At the OK prompt, type `INCLUDE FCONTROL.4TH` (include path if needed), which will load the file, then type `MAIN` to run the program. The dialog box will appear with two keyboard arrays of push-buttons.

Clicking on the on/off buttons will control each output device individually. Click on `KILL` to turn off all devices, click on `ALL-ON` to turn on all devices. The LED display should respond accordingly. The four preset function buttons can be programmed to turn on any combination of output devices. In this case, the preset buttons are programmed to drive the four phases of a stepper motor. Clicking function buttons 1, 2, 3, and 4 in sequence, and repeating, will step the motor in one direction. To reverse motor direction, click function buttons 4, 3, 2, and 1 in sequence, and repeat. See Skip Carter's article "Stepper Motors" (*FD* XVII/5) for an in-depth view of theory and interfacing to stepper motors.

After you get FCONTROL.4TH up and running, read through the source code and the comments to figure out what each section of code is doing. Have some fun by changing the parameters to see what happens. Change the size of the buttons. Move them around to different locations. Patch new functions into the case statement in `DO.BUTTON`, and change the text on the buttons accordingly. This will give you a feel of how the program works. Load some of the demonstration programs that come with WinForth and study their source code.

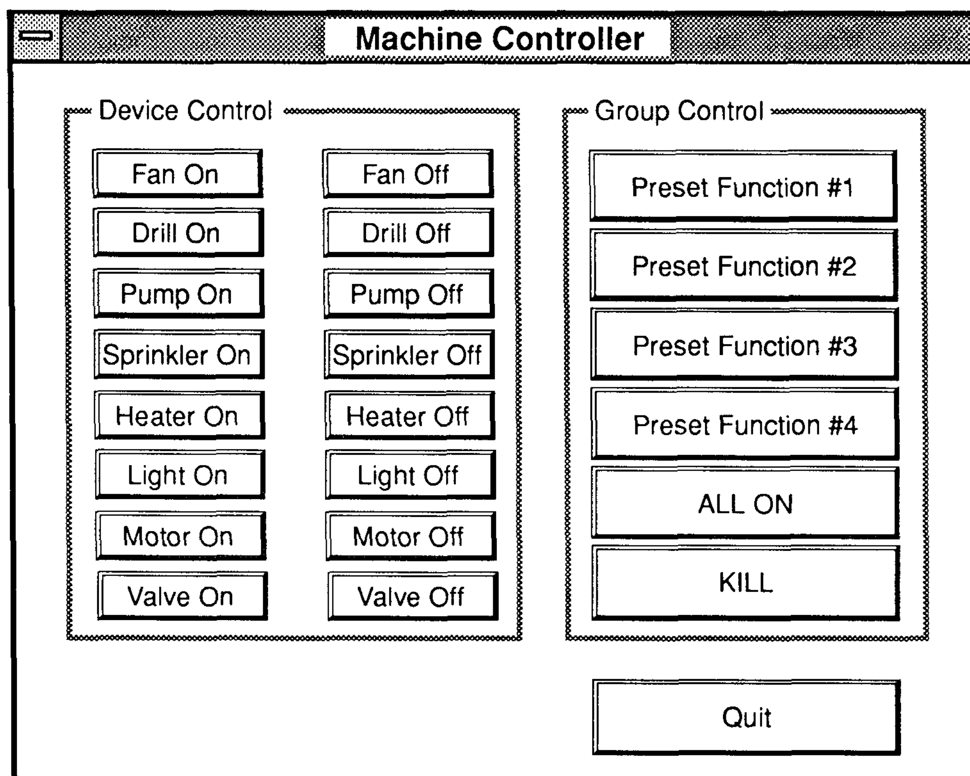
Read the WinForth programming overviews in the help files. WinForth has a built-in "windowing layer" that handles many Windows events automatically and hides much of the complexity of the Windows API, so you can focus on your application and get it completed faster. But, if you choose, you can write code using direct calls to the API functions or even third-party DLLs using the `APIHOOK` function.

Compiled end-user applications consist of two files, an EXE file and an OVL file. When FCONTROL.4TH was compiled into turnkey executable files, the total size was 184K bytes.

This will give you a good start in Windows programming, and will demonstrate how you can "Do More With Less" using Forth.

Code begins on next page.
Downloading: The code can be found under the filenames FCONTROL.4TH, FCONTROL.EXE, and FCONTROL.OVL at ftp://ftp.forth.org/pub/Forth/FD/1996

Ken Merk, who graduated from BCIT as an Electronic Technologist, is a married father of two girls and lives in Langley, B.C., Canada. He works for Canadian Pacific Railway, and is involved in a braking system used on caboose-less trains—the caboose is replaced by a black box which monitors many parameters of the train and sends them digitally by radio to the head end. In emergencies, a remote radio can trigger braking. Other projects include infrared bearing-failure detectors, wind detectors, and mountain-top radio communication sites. Merk originally used Forth to learn 8088 assembler, and found it a great tool to control electronic hardware.



Listing. Parallel port interface.

```
\ FCONTROL.4TH                                    Ken Merk   June/96
\ WINFORTH

\ ******************** Parallel Port Interface ********************

ASM                              \ Loads the Forth assembler

DECIMAL

 64 8 @L                        \ Look for active LPT1 port
        0= .IF                  \ If no port found then abort

        CLS
        23 8  GOTOXY .( Parallel printer port not found.)
        CLOSE QUIT

           .THEN

64 8 @L EQU #PORT               \ Find port addr for printer card
                                \ assign to constant #PORT

1   CONSTANT  FAN               \ assign each Device its binary weighting
2   CONSTANT  DRILL
4   CONSTANT  PUMP
8   CONSTANT  SPRINKLER
16  CONSTANT  HEATER
32  CONSTANT  LIGHT
64  CONSTANT  MOTOR
128 CONSTANT  VALVE

CODE BSET   ( b  #port -- )    \ will SET each bit in #port that matches
CX POP                         \ every high bit in byte B.
DX, TOS MOV
AX, DX IN
AL, CL OR
DX, AL OUT
TOS POP
NEXT,
END-CODE

CODE BRESET  ( b  #port -- )   \ will RESET each bit in #port that matches
CX POP                         \ every high bit in byte b.
CX NOT
DX, TOS MOV
AX, DX IN
AL, CL AND
DX, AL OUT
TOS POP
NEXT,
END-CODE

 : >ON      ( b  -- )     #PORT BSET    ;     \ turn ON device
 : >OFF     ( b  -- )     #PORT BRESET  ;     \ turn OFF device


 : KILL     ( -- )        00 #PORT pc!  ;     \ turn OFF all devices
 : ALL-ON   ( -- )        255 #PORT pc! ;     \ turn ON all devices
 : WRITE.PORT  ( b -- )   #PORT pc!   ;       \ WRITE byte to port

 KILL                                         \ kill all LEDs
```

```
\ *********************** Control Dialog box ****************************

200 CONSTANT ID_FANON          \ Assign each control an ID number
201 CONSTANT ID_FANOFF         \ which corresponds to a "ID_" constant
202 CONSTANT ID_DRILLON        \ to make code easier to follow.
203 CONSTANT ID_DRILLOFF
204 CONSTANT ID_PUMPON
205 CONSTANT ID_PUMPOFF
206 CONSTANT ID_SPRINKON
207 CONSTANT ID_SPRINKOFF
208 CONSTANT ID_HEATERON
209 CONSTANT ID_HEATEROFF
210 CONSTANT ID_LIGHTON
211 CONSTANT ID_LIGHTOFF
212 CONSTANT ID_MOTORON
213 CONSTANT ID_MOTOROFF
214 CONSTANT ID_VALVEON
215 CONSTANT ID_VALVEOFF
216 CONSTANT ID_ALLON
217 CONSTANT ID_KILL
218 CONSTANT ID_FUNC1
219 CONSTANT ID_FUNC2
220 CONSTANT ID_FUNC3
221 CONSTANT ID_FUNC4

" Machine Controller"                                   \ Caption text
35 10  235 175  WS_CAPTION  WS_POPUP D+                 \ Size and style
                 WS_SYSMENU D+  DS_MODALFRAME D+        \ of dialog box
                 DIALOG  CONTROLDLG                     \ Dialog name


\ ************************** Button Array1 ***************************

"     "      12  10     112 143   -1                     \ Border around
             WS_BORDER WS_VISIBLE D+ WS_CHILD D+         \ button array1
             SS_BLACKFRAME D+  " STATIC" CONTROL

\ Create button array1- Button text, x y position in box, width and height
\ of button, ID that identifies which button.

" Fan On"           20   19    45 14  ID_FANON       PUSHBUTTON
" Fan Off"          69   19    45 14  ID_FANOFF      PUSHBUTTON
" Drill On"         20   35    45 14  ID_DRILLON     PUSHBUTTON
" Drill Off"        69   35    45 14  ID_DRILLOFF    PUSHBUTTON
" Pump On"          20   51    45 14  ID_PUMPON      PUSHBUTTON
" Pump Off"         69   51    45 14  ID_PUMPOFF     PUSHBUTTON
" Sprinkler On"     20   67    45 14  ID_SPRINKON    PUSHBUTTON
" Sprinkler Off"    69   67    45 14  ID_SPRINKOFF   PUSHBUTTON
" Heater On"        20   83    45 14  ID_HEATERON    PUSHBUTTON
" Heater Off"       69   83    45 14  ID_HEATEROFF   PUSHBUTTON
" Light On"         20   99    45 14  ID_LIGHTON     PUSHBUTTON
" Light Off"        69   99    45 14  ID_LIGHTOFF    PUSHBUTTON
" Motor On"         20   115   45 14  ID_MOTORON     PUSHBUTTON
" Motor Off"        69   115   45 14  ID_MOTOROFF    PUSHBUTTON
" Valve On"         20   131   45 14  ID_VALVEON     PUSHBUTTON
" Valve Off"        69   131   45 14  ID_VALVEOFF    PUSHBUTTON
"  Device Control"  20   6     51 10      -1         LTEXT
```

(Continues on next page.)

```
\ ************************** Button Array2 ****************************
"      "       135 10     88 143  -1                    \ Border around
               WS_BORDER WS_VISIBLE D+ WS_CHILD D+      \ button array2
               SS_BLACKFRAME D+  " STATIC" CONTROL

\ Create button array2- Button text, x y position in box, width and height
\ of button, ID that identifies which button.
" Preset Function #1" 145 19    68 18  ID_FUNC1   PUSHBUTTON
" Preset Function #2" 145 39    68 18  ID_FUNC2   PUSHBUTTON
" Preset Function #3" 145 59    68 18  ID_FUNC3   PUSHBUTTON
" Preset Function #4" 145 79    68 18  ID_FUNC4   PUSHBUTTON
" ALL ON"             145 99    68 18  ID_ALLON   PUSHBUTTON
" KILL"               145 119   68 26  ID_KILL    PUSHBUTTON
" Quit"               173 158   40 14  IDCANCEL   PUSHBUTTON
"  Group Control"     145 6     48 10   -1        LTEXT
END-DIALOG

\ Case statement takes button ID's given by the message handler
\ to determine what action to take.
: DO.BUTTON
           CASE
            ID_FANON      OF    FAN        >ON   ENDOF
            ID_FANOFF     OF    FAN        >OFF  ENDOF
            ID_DRILLON    OF    DRILL      >ON   ENDOF
            ID_DRILLOFF   OF    DRILL      >OFF  ENDOF
            ID_PUMPON     OF    PUMP       >ON   ENDOF
            ID_PUMPOFF    OF    PUMP       >OFF  ENDOF
            ID_SPRINKON   OF    SPRINKLER  >ON   ENDOF
            ID_SPRINKOFF  OF    SPRINKLER  >OFF  ENDOF
            ID_HEATERON   OF    HEATER     >ON   ENDOF
            ID_HEATEROFF  OF    HEATER     >OFF  ENDOF
            ID_LIGHTON    OF    LIGHT      >ON   ENDOF
            ID_LIGHTOFF   OF    LIGHT      >OFF  ENDOF
            ID_MOTORON    OF    MOTOR      >ON   ENDOF
            ID_MOTOROFF   OF    MOTOR      >OFF  ENDOF
            ID_VALVEON    OF    VALVE      >ON   ENDOF
            ID_VALVEOFF   OF    VALVE      >OFF  ENDOF
            ID_ALLON      OF    ALL-ON           ENDOF
            ID_KILL       OF    KILL             ENDOF
            IDCANCEL      OF    0 CLOSEDLG       ENDOF
            ID_FUNC1      OF    5  WRITE.PORT    ENDOF
            ID_FUNC2      OF    9  WRITE.PORT    ENDOF
            ID_FUNC3      OF    10 WRITE.PORT    ENDOF
            ID_FUNC4      OF    6  WRITE.PORT    ENDOF
          ENDCASE ;

\ Dialog box message handler intercepts WM_INITDIALOG and WM_COMMAND
\ messages and then processes them.
\ Button ID's are taken from WM_COMMAND's wParam and sent to DO.BUTTON
\ case statement which determines what action to take.
: CONTROLDLGPROC
               wMsg
               CASE
              WM_INITDIALOG OF TRUE ENDOF
              WM_COMMAND   OF wParam  DO.BUTTON  TRUE ENDOF
              FALSE SWAP
              ENDCASE  ;

\ Runs the dialog template with the associated dialog message handler
\ which starts the program.
: MAIN        CONTROLDLG ['] CONTROLDLGPROC  RUNDLG DROP   ;
```

Filters and Sponges

Wil Baden
Costa Mesa, California

Filters

A *filter* is a program that takes a file as input, and does *something* to it line-by-line or character-by-character, producing output. The output from one filter can be "piped" as input to another filter, and so on.

Or as ERIC RAYMOND, *The New Hacker's Dictionary* (ISBN 0-262-26069-6), has it—

> **filter** n. [orig. **UNIX**, now also in **MS-DOS**] A program that processes an input data stream into an output data stream in some well-defined way, and does no I/O to anywhere else except possibly on error conditions; one designed to be used as a stage in a *pipeline*.

Filter programs are common and useful. In this section we show how to make them easy to write.

We'll presume some pet words, definitions given in Appendix A.

```
OPENED   INPUT   OUTPUT
CLOSED   REWIND
checked   needed
IN   OUT   INBUF
PLACE   BOUNDS
```

These words have already appeared in Stretching Forth articles.

As the first example, we make a filter where *something* doesn't do anything. Lines are simply copied. Call it what you like. I call it `COPY` here. [Figure One.]

If you can redirect the output from `TYPE` and `CR`, as can be done in traditional Forth systems, that's enough. Otherwise, replace `TYPE CR` by `OUT WRITE-LINE` checked. [Figure Two.]

Let's rearrange things so we can factor cleanly. [Figure Three.]

Figure One.

```
: COPY                                          ( -- )
    BEGIN                                         ( )
        INBUF /COUNTED-STRING
            IN READ-LINE checked                  ( u flag)
    WHILE
        INBUF SWAP TYPE CR                        ( )
    REPEAT                                        ( u)
    DROP                                          ( )
    IN REWIND
;
```

Figure Two.

```
: COPY                                          ( -- )
    BEGIN                                         ( )
        INBUF /COUNTED-STRING
      IN READ-LINE checked               ( u flag)
    WHILE
        INBUF SWAP OUT WRITE-LINE checked     ( )
    REPEAT                                    ( u)
    DROP                                      ( )
    IN REWIND
;
```

Figure Three.

```
: COPY                                    ( -- )
    BEGIN                                     ( )
        INBUF /COUNTED-STRING
            IN READ-LINE checked        ( u flag)
        IF INBUF SWAP ( s u) TRUE     ( s u true)
        ELSE DROP ( ) IN REWIND  FALSE  ( false)
        THEN
    WHILE ( s u)
        TYPE CR                              ( )
    REPEAT                                   ( u)
;
```

We factor what's between BEGIN and WHILE as filter-refill in Listing One. This leaves us with Figure Four.

Next we define a macro, using Standard Forth EVALUATE. [Figure Five]

Now we can define our filter. [Figure Six]

For character-by-character filters we define another macro, CYPHER, in Listing One. CYPHER gets its name from cryptography, where it is the term for alphabetic substitution.

As two simple character-by-character filters, we have RAISE-CASE and ROT13. [Figure Seven]

Another example of a filter is the program that prints my listings with line numbers on the non-blank lines.

Figure Four.

```
: COPY                                        ( -- )
    BEGIN                                       ( )
        filter-refill
    WHILE ( s u)
        TYPE CR                                 ( )
    REPEAT                                      ( u)
;
```

Figure Five.

```
: FILTER
    S" BEGIN filter-refill WHILE "
    EVALUATE ; IMMEDIATE
```

Figure Six.

```
: COPY FILTER TYPE CR REPEAT ;            ( -- )

or

: COPY FILTER OUT WRITE-LINE checked REPEAT ;    ( -- )
```

Figure Seven.

```
( Convert a character to upper-case. )
: >UPPER                                ( Char -- CHAR )
    DUP [CHAR] a -  26 U< IF   BL -   THEN
;

( Rotate letter 13 positions in the alphabet. )
: >ROT13                                ( Char -- Pune )
    DUP  BL OR  [CHAR] a -   13 U<
        IF   13 +   EXIT   THEN
    DUP  BL OR  [CHAR] n -   13 U<
        IF   13 -   EXIT   THEN
;

( Convert a file to uppercase. )
: RAISE-CASE FILTER 2DUP CYPHER >UPPER TYPE CR REPEAT ;

( Convert a file by rotating letters 13 positions. )
: ROT13      FILTER 2DUP CYPHER >ROT13 TYPE CR REPEAT ;
```

Sponges

Here are two more definitions from *The New Hacker's Dictionary.*

> **sponge** n. [UNIX] A special case of a filter that reads its entire input before writing any output; the canonical example is a sort utility. Unlike most filters, a sponge can conveniently overwrite the input file with the output data stream.
>
> **slurp** vt. To read a large data file entirely into core before working on it. This may be contrasted with the strategy of reading a small piece at a time, processing it, and then reading the next piece. "This program slurps in a 1K-by-1K matrix and does an FFT."

We now make a canonical example of a SPONGE. After using FILTER to SLURP a file into the HEREAFTER, we make an index for the lines of the fileimage, re-order the fileindex, and REGURGITATE the file.

The fileimage is placed in dataspace in the HEREAFTER, that is, a given distance after HERE. How far from HERE isn't important. The fileindex is placed after the fileimage.

To re-arrange the lines of a file in ASCII collating sequence,

```
S" name-of-file" SORTED
```

A text editor is often implemented as a sponge.

See Listing Two for SORTED.

See Appendix B for QSORT.

Wil Baden is a professional programmer with an interest in Forth. Send e-mail to wilbaden@netcom.com asking for a text-only version of "Filters and Sponges."

Listing One.

```
 1     : filter-refill                   ( -- s u true | false )
 2         INBUF /COUNTED-STRING
 3             IN READ-LINE checked ( u flag)
 4         IF INBUF SWAP ( s u) TRUE ( s u true)
 5         ELSE DROP ( ) IN REWIND  FALSE ( false)
 6         THEN                             ( s u true | false )
 7     ;

 9 : FILTER S" BEGIN filter-refill WHILE " EVALUATE ; IMMEDIATE

11 : CYPHER                         ( s u "word" -- )
12             S" CHARS BOUNDS ?DO I C@ "
13             PARSE-WORD S+
14             S"  I C! 1 CHARS +LOOP " S+
15             EVALUATE
16 ; IMMEDIATE

18 \ ( With PLEASE )
19 \ : CYPHER  PARSE-WORD >PAD PLEASE
20 \   " CHARS BOUNDS ?DO I C@ ~ I C! 1 +LOOP "
21 \ : IMMEDIATE
```

S+ is string catenation, and a definition was given in Stretching Forth article "Circular String Buffer" (*Forth Dimensions* XVIII/2).

If you lack PARSE-WORD you can make do with—

```
: PARSE-WORD BL WORD COUNT ;          ( "name" -- s u )
```

Listing Two.

```
 1 ( SORTED )

 3     CREATE filename   /COUNTED-STRING 1+ CHARS ALLOT

 5     : HEREAFTER   HERE 200 CHARS + ;   ( -- c_addr )

 7     0 VALUE fileimage
 8     0 VALUE fileindex

10     VARIABLE tally

12     : check-available-dataspace     ( n -- )
13         CHARS  fileindex +  ALIGNED  tally @ CELLS +
14         HERE -  UNUSED U< NOT ABORT" (Out of Dataspace) "
15     ;

17     : SLURP                          ( -- )
18         filename COUNT INPUT TO IN
19         HEREAFTER TO fileimage
20         fileimage TO fileindex
21         0 tally !
22         FILTER                           ( c_addr u)
23             DUP 1+ check-available-dataspace
24             fileindex PLACE              ( )
25             fileindex COUNT CHARS + TO fileindex
26             1 tally +!
```

(Listing Two continues on next page.)

```
27         REPEAT
28         IN CLOSED   0 TO IN
29     ;

31     : make-index                    ( -- )
32         fileindex ALIGNED TO fileindex
33         fileimage tally @ 0 ?DO         ( c_addr)
34             DUP I CELLS fileindex + !
35             COUNT CHARS +
36         LOOP                            DROP
37     ;

39     : REGURGITATE                   ( -- )
40         filename COUNT OUTPUT TO OUT
41         tally @ 0 ?DO
42             I CELLS fileindex + @ COUNT
43                 OUT WRITE-LINE checked
44         LOOP
45         OUT CLOSED   0 TO OUT
46     ;

49 : SORTED                            ( c_addr u -- )
50     filename PLACE
51     SLURP
52     make-index
53     fileindex tally @ ['] CCOMPARE QSORT
54     REGURGITATE
55 ;
```

Appendix A.

```
 1 ( Stock Words for Filters and Sponges )

 3 : OPENED OPEN-FILE ABORT" Can't open " ;
 5 : INPUT   R/O OPENED ;         ( c_addr u -- fileid )
 6 : OUTPUT  W/O OPENED ;         ( c_addr u -- fileid )

 8 : CLOSED  ?DUP IF CLOSE-FILE abort" Can't close. " THEN ;
 9 : REWIND  ?DUP IF
10         0 0 ROT REPOSITION-FILE ABORT" Can't rewind. "
11     THEN
12 ;

14 : checked   ABORT" (File Access Error) " ;   ( ior -- )
15 : needed                                    ( n -- )
16     DEPTH U< NOT ABORT" Not enough on the stack. "
17 ;

19 0 VALUE IN              ( Global Fileid for Input )
20 0 VALUE OUT             ( Global Fileid for Output )

22 : PLACE   2DUP 2>R   CHAR+ SWAP CHARS MOVE   2R> C! ;

24 : BOUNDS OVER + SWAP ;              ( a n -- a+n a )

26 ( Common Input Buffer for Filters )
27 255 CONSTANT /COUNTED-STRING
28 CREATE INBUF    /COUNTED-STRING 2 + CHARS ALLOT
```

Appendix B.

This is a tidying of my QSORT given in *Forth Dimensions* XVI/1. Look there for explanation.

```
 1 ( Hoare's Quicksort )( Non-Recursive )( Wil Baden 1967-1993 )
 2 ( Standard Forth CORE EXT with NOT )

 4 ( Use your definition of NOT. )

 6 VARIABLE  'inorder

 8 : exchange   2DUP  @ >R  @ SWAP !  R> SWAP ! ;   ( x y -- )

10 : order-three                        ( lo hi mid -- lo hi mid )
11     >R                                    ( lo hi)( R: mid)
12         OVER @ R@ @ 'inorder @ EXECUTE 0>
13             IF   OVER R@ exchange   THEN
14         R@ @ OVER @ 'inorder @ EXECUTE 0> IF
15             R@ OVER exchange
16             OVER @ R@ @ 'inorder @ EXECUTE 0>
17                 IF   OVER R@ exchange   THEN
18         THEN
19     R>                                    ( lo hi mid)( R: )
20 ;

22 VARIABLE  guess

24 : skip-lowers                     ( x y -- x y )
25     >R
26         BEGIN
27             CELL+
28             DUP @ GUESS @ 'inorder @ EXECUTE 0< NOT
29         UNTIL
30     R>
31 ;

33 : skip-highers                     ( . y -- . y )
34     BEGIN
35         1 CELLS -
36         guess @ OVER @ 'inorder @ EXECUTE 0< NOT
37     UNTIL
38 ;

40 : partition                        ( lo hi -- lo y x hi )
41     2DUP OVER - 2/ ALIGNED +            ( lo hi mid)
42     order-three
43     @ guess !                           ( lo hi)
44     2DUP                                ( lo hi x y)
45         BEGIN
46             skip-lowers
47             skip-highers
48             2DUP > NOT
49         WHILE
50             2DUP exchange
51             2DUP 2 CELLS - >
52         UNTIL
53             >R CELL+ R>
54             1 CELLS -
55         THEN
56     SWAP ROT                            ( lo y x hi)
57 ;
```

(Appendix B continues on next page.)

```
59 : smallersection-first          ( lo y x hi -- lo y x hi)
60     2OVER 2OVER SWAP - >R SWAP - R> <
61         IF    2SWAP    THEN
62 ;

64 : hoarify                       ( x y -- ... x y)
65     BEGIN
66         2DUP SWAP - 2 CELLS >
67     WHILE
68         partition                        ( ... lo y x hi)
69         smallersection-first
70     REPEAT                               ( ... lo hi)
71 ;

73 : order-a-pair                  ( lo hi -- )
74     2DUP = NOT IF
75         OVER @ OVER @ 'inorder @ EXECUTE 0>
76             IF    2DUP exchange    THEN
77     THEN                                          2DROP
78 ;

80 : short-order                   ( lo hi -- )
81     2DUP SWAP - 1 CELLS > IF
82         DUP 1 CELLS -                   ( lo hi mid)
83         order-three
84         DROP 2DROP
85     ELSE
86         order-a-pair                    ( )
87     THEN
88 ;

90 : QSORT                        ( a_addr n xt -- )
91     'inorder !                           ( a_addr n)
92     DUP 0= IF    2DROP EXIT    THEN
93     1-    CELLS    OVER +               ( lo ho)
94     DEPTH >R
95         BEGIN                            ( ... lo ho)
96             hoarify                      ( ... lo ho)
97             short-order                  ( ...)
98             DEPTH R@ <
99         UNTIL                            ( )
100    R> DROP
101 ;

103 : CCOMPARE                 ( c_addr c_addr -- -1|0|1 )
104     >R COUNT R> COUNT COMPARE
105 ;
```

Using Forth to manipulate the real world

Forthware

Measuring Frequency and Sampling Time-Dependent Signals

Skip Carter
Monterey, California

Introduction

This month I'd like to discuss a special kind of input signal: time-dependent signals whose frequency is important. We will take a look at how to determine the frequency of a digital signal and will consider some of the issues involved in getting a useful sample of an analog signal.

Frequency Measurement

Let us first consider the problem of how to measure the frequency of a digital signal (i.e., a square wave). Signals such as this can come from a digital source or from a suitably conditioned analog source. Examples include some A/D chips, a voltage-controlled oscillator, or a venerable 555 oscillator chip.

There are two basic ways of making this measurement:

- *Period counting*. This involves measuring the time between successive leading (or trailing) edges of the incoming pulses.
- *Frequency counting*. This is done by counting the number of edges that occur within a fixed time interval.

In either case, one takes several measurements, then averages them in order to get a useful measurement. Both methods require that the edges come in slow enough for the software to respond to their arrivals. This requirement makes the high-level code presented here for illustration (Listing One), of limited direct usefulness. To get a higher maximum frequency in a real system, the edge detection would be done in assembler and/or as an ISR. (Notice that the period counter changes the hardware timer to run at 1.1 MHz, instead of the normal 18.2 Hz, so we get a reasonable resolution. This messing around with the hardware timer is pretty system dependent; I never got it to work from a DOS shell within Windows.)

In addition, each technique has its own particular weaknesses. With period counting, there is the problem of what happens when there are missing edges. To illustrate, suppose the input signal was a 1 kHz square wave, so the time between leading edges is 1 millisecond. Given the normal vagaries of the measurement, we might expect to see a variation of, say, ±10%, so the individual measurements might vary from 0.9 to 1.1 milliseconds. Averaging several measurements will handle this and give us an estimate of 1 millisecond with a reasonable degree of confidence. But now suppose that, every once in a while, we miss an edge—each time this happens, we get a value of *two milliseconds* to fold into our average! This can significantly shift our estimated frequency, plus it will have a strong effect on the degree of confidence of our measurement. An occasional extra pulse can also cause problems by giving us *two* time values that are too short. Averaging over a large number of samples helps with this, but it might not be practical for the application.

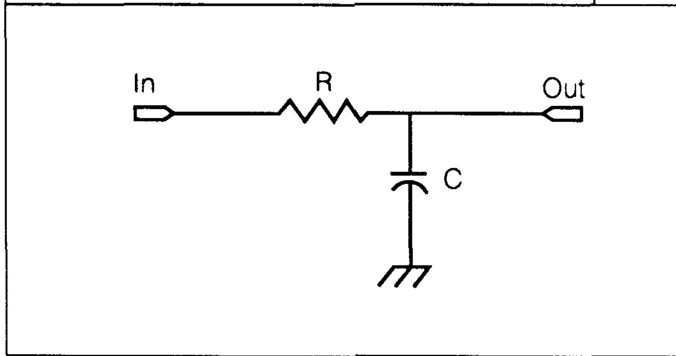
Another possibility is to sample adaptively. Adaptive sampling requires calculating a running mean and variance, and the sampling is stopped when the variance drops below some acceptance threshold. The problem with adaptive sampling is that it will consume significant computing resources between each sample. A practical period sampling routine will also have a timeout provision, otherwise it will wait forever for an edge that never happens if the signal stops or is interrupted.

Frequency counting is not as adversely affected by an occasional missing or extra edge. However, frequency counting is vulnerable to counter overrun. Even if the software can keep up with the edges coming in, if the sample time is too long the counter that is accumulating the edges could overrun. A short sample time helps this, since it increases the frequency at which an overrun will occur. But if the sample interval is too small, the measurement has a reduced degree of confidence. One could also detect the overrun and handle it in some way (e.g., setting an overrun flag, stopping the count, etc.).

Sampling a Time-Dependent Signal

The problems involved in handling a more general, time-dependent signal—say a digitized acoustic signal—are considerably trickier. A primary problem is how to make the measurements without high-frequency aliasing distorting the result. The problem is that if $2 \times n$ is the sample rate, there is no way to distinguish a signal of frequency n from a signal of $2n$ (or any other integer multiple of n), because in both cases one sees a full cycle in two samples. In the $2n$ case, there was a whole cycle ***in between*** our samples that we missed. This means that, if we are sampling at $2 \times n$, there is a whole range of frequencies from n upward that can contaminate our measurements. This spurious folding of high frequencies

Figure One. A low-pass R-C filter. The 3 db frequency cutoff is at $f = 1/(2\pi RC)$.



down to a lower one is called *aliasing*.

One can reduce this problem somewhat by sampling at a high rate, the idea being that it will increase the frequency at which aliasing begins to occur, and one hopes there is less of the higher frequency around to bother us. The problem is that it might not be the case that there is less at the higher frequency and, even worse, this solution uses more CPU resources. A better solution is to prevent the high frequencies from getting into your samples in the first place. This is done by placing an *anti-aliasing filter* in the analog circuit before the A/D converter. An anti-aliasing filter is just a low-pass filter designed to reject frequencies above *n*.

Anti-aliasing filters are frequently implemented as simple R-C filters like in Figure One. This filter is not that great, as filters go: the rate of attenuation of the higher frequencies is rather slow. The fraction of the signal passed through the filter as a function of frequency is called the *magnitude response*. The magnitude response curve is an important measure of the quality and suitability of a filter. Unfortunately, it is often used as the *only* measure.

Another measure that can be just as important to consider is the frequency-dependent effect the filter has upon the *phase* of the signal. This is the *phase response*. The phase response information is most useful in two forms, the *phase delay* and the *group delay*. The phase delay is just a dimensional form of the phase response: it gives the amount of time a signal of a given frequency is delayed by the filter. The group delay describes something slightly different. Suppose our signal is like an FM radio signal that is being modulated in frequency around a basic frequency. The modulation can be thought of as another signal (the envelope) riding on top of the basic frequency (the carrier). The phase delay of the envelope is not generally the same as the phase delay of the carrier. The group delay gives the delay time of the envelope.

So, to properly judge the suitability of a given filter, we really need to check all three functions: the magnitude response, the phase delay, and the group delay. As you might expect, all filters sacrifice performance in one of these three functions in order to gain in another. The best compromise depends on your application.

All these filter characteristics can be derived from the filter's *transfer function*. This is a complex function (that is, it contains complex numbers in it) that takes a bit of mathematics to be able to derive for an arbitrary filter. I will only give results here for the low-pass RC filter. If you are interested in learning more about this, Horowitz and Hill contains a readable introduction to the topic. The low-pass RC filter has the transfer function,

$$H(\omega)\frac{1-jRC\omega}{1+R^2C^2\omega^2}$$

Here I have used the engineer's notation for the $\sqrt{-1}$, *j*, not the scientist's notation, *i*. The term ω is the frequency in radians; to get Hertz, divide ω by 2π. With this equation, we can get the magnitude response:

$$|H(\omega)| = \frac{1}{\sqrt{1+R^2C^2\omega^2}}$$

The phase response is:

$$\theta(\omega) = \tan^{-1}\left(\frac{Im(H(\omega))}{Re(H(\omega))}\right)$$
$$= \tan^{-1}(RC\omega)$$

So the phase delay is:

$$\tau_p(\omega) = -\frac{\theta(\omega)}{\omega}$$
$$= -\frac{1}{\omega}\tan^{-1}(-RC\omega)$$

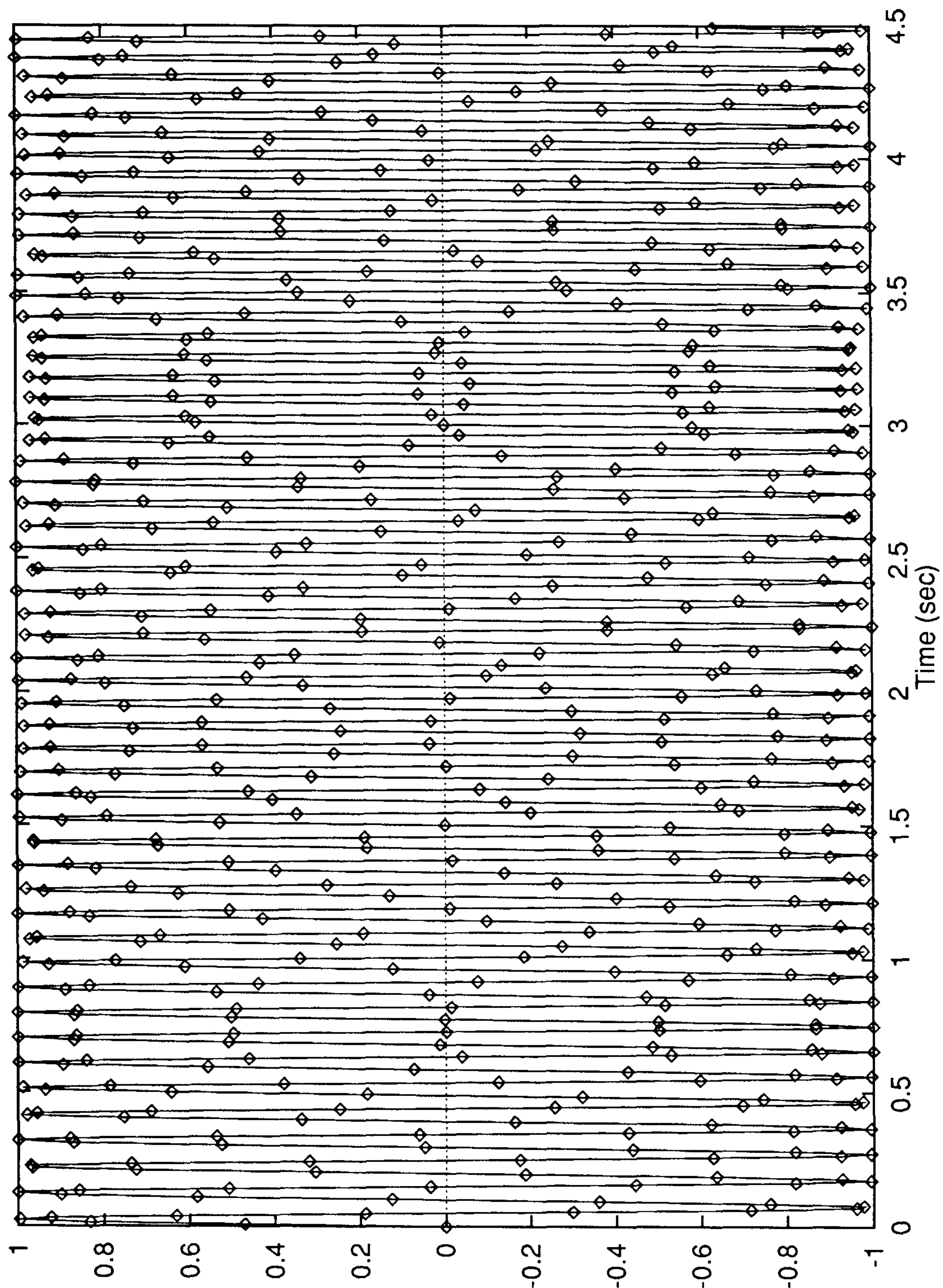
and the group delay is:

$$\tau_g(\omega) = -\frac{d\theta(\omega)}{d\omega}$$
$$= \frac{RC}{1+R^2C^2\omega^2}$$

With the anti-aliasing filter in place, we still need to decide on the proper sampling rate. You might recall reading elsewhere about something called the Nyquist sampling theorem. This theorem is what we want—it tells us the sample rate must be at least twice the bandwidth of the signal in order to avoid aliasing. Notice that I said *bandwidth*, which is the range from the lowest to the highest frequency (in the simplest case, where there are no gaps) in the signal. The Nyquist theorem is widely misquoted as stating that we must sample at twice the *highest frequency* of the signal. But the bandwidth and the highest frequency are not the same thing, unless we are dealing with a base band signal (one that has content from a frequency of zero all the way up to the highest frequency).

The distinction can be quite important. Consider the following real-life example. In the RAFOS subsurface ocean drifter I helped to develop, we navigate the float by listening to a tone emitted by a pre-placed acoustic beacon mooring. These beacons output a long tone that sweeps in frequency from 258.5 to 261.5 Hz. The bandwidth of this tone is the range of the sweep, 3 Hz. So the Nyquist theorem states that we need a sampling rate of at least 6 Hertz, *not* twice the highest frequency of 261.5 Hz (523 Hz). As a result, the RAFOS float can comfortably oversample the signal at 10

Figure Two. The reference chirp signal. The diamonds are the locations of the 128 Hz samples.



Hz, using a lowly 6805 microprocessor. Erroneously sampling at 523 Hz would have required a faster processor, which would have required more electrical power, which, in turn, would have made the instrument an impractical device (the drifter runs on batteries and has mission times measured in months, 48 being our current record).

If we are uniformly sampling a signal at the proper rate, and if there is no aliased signal contaminating our measurement, we can recover the value of the signal at any time. To do this, we need to do a convolution of our samples with the `sinc` function (this is the uniform sampling theorem),

$$x(t) = \sum_{n=-\infty}^{\infty} x(nT) sinc\left(\frac{\pi}{T}(t - nT)\right)$$

where t is the time we want to reconstruct the signal at, T is the interval between samples, and n is the sample index. (I am not going to explain the mathematics behind this here. It could provide material for several future columns to explain it. If you want to research this on your own, the book by Bracewell is highly recommended.) In order to make practical use of this equation, we will take the index

n over the number of data samples, instead of infinity.

A Forth implementation of the `sinc` function is,

```
: SINC  ( -- , F: x -- sincx )
    FDUP F0= IF    FDROP 1.0E0
             ELSE FDUP FSIN FSWAP F/
             THEN
;
```

By the way, if you look up equations like these in the literature, I guarantee you will have a horrible time reconciling factors of π, 2, and -1. (This is generally known as the π-throwing contest. Where did the π go?) In the mathematics literature, these factors tend to be missing from the equations altogether. In the engineering literature, they are in different places in different books. The reason is that such factors are immaterial, as far as the mathematical theory of all this is concerned—they are just normalization and dimensionalization factors. In the engineering context, there is no one way to do normalization and dimensionalization—they just need to be done self-consistently; so one book's version can differ from that of another book.

Now that we are armed with the uniform sampling theorem, we can do a little experiment to demonstrate what I said about sampling a bandlimited signal. Listing Two, gensig.fth, is a program that will generate a test signal that starts at one frequency and slides up to another (a "chirp"). I have set things up so the simulated signal sweeps from 10 Hz to 12 Hz in 4.5 seconds. When the constant `SAMPLING?` is `FALSE`, the output is at the equivalent of 128 samples per second.

A subsample of the output of this program is what we will be using as data; a plot is shown in Figure Two. This signal has a bandwidth of 2 Hz, so the Nyquist sampling rate is 4 Hz. We will oversample and sample at 6.4 Hz. Now, we can't just take every 20th sample from the data in Figure Two to use as our measurement data; such a signal would contain a serious amount of aliasing in it. To make the signal usable, we will mix it with an 11 Hz signal and then apply a low-pass filter with a 5 Hz cut-off to the result.

Why do we do that? From elementary trigonometry,

$$2 \times \sin(x) \times \cos(y) = \sin(x-y) + \sin(x+y)$$

which means that if we take a signal of one frequency, x, and multiply it by another signal of frequency, y, we end up with one signal with frequency $x-y$ and another at $x+y$. So if x is our original signal which sweeps from 10 to 12 Hz, and y is a fixed signal at 11 Hz, we end up with a signal centered at 0 Hz and another at 22 Hz. The one at 0 Hz is the one we want to keep; the other at 22 Hz we will filter out.

This technique for shifting the center frequency of a signal is the heart of what is known as a *heterodyne* mixer. The special case where we shift to a center frequency of zero is known as a *homodyne* mixer. After mixing and filtering the original signal, we can safely subsample it. The anti-aliasing filter being used here is a first-order, low-pass Butterworth filter. There are better choices for the filter (such as a Bessel filter), but I am using it here because it is simple and it illustrates a point we will get to later. All of these operations are in the code gensig.fth when the constant `SAMPLING?` is set to `TRUE`.

The output for the sampling case are the simulated measurements we want to use the uniform sampling theorem on. The code in regen.fth (Listing Three) reads the data and applies the theorem to it. Comparing the output of `regen` with its input, we see the smoother result one might expect. In order to see what we theoretically expect, go back and run `gensig` with `SAMPLING?` set to `FALSE` and the minimum and maximum frequencies set to -1.0 and 1.0, respectively. Running `gensig` this way generates the reference signal without the 11 Hz carrier.

Comparing the carrier-free signal with the reconstructed signal (Figure Three), we see that we generally do pretty well. There are two problems we can see with our reconstruction:

- The end points aren't that well matched. In this example, the starting point looks very good, but that is an artifact of the fact that the signal starts at zero. The end-point problem is due to the fact that the theorem we are using assumes there is data on both sides of our estimation point (and, in fact, an infinite amount of it) but, as we near the edges, the calculation gets most of its information from only one side of the point. More data helps here, but the end points are always going to be a problem.
- The reconstruction is slightly phase shifted late. If you look carefully, you'll notice that the phase shift is in the measurement data and that the reconstruction has the same shift. This is because the phase shift is caused by the anti-aliasing filter. The group delay of the Butterworth filter being used here is frequency dependent. It starts at zero and monotonically increases until about 10 Hz; at 2 Hz, it is about 0.05 seconds. This is a good example of where the group delay characteristics are more important than how sharp the high frequency cutoff is.

Conclusion

This installment further extends our ability to handle data coming in from the outside world. We have just scratched the surface of the issues involved in dealing with frequency-dependent data. For more information about handling time-dependent signals, see the references. The book by Oppenheim and Schafer is especially thorough, but it's not for the mathematically timid.

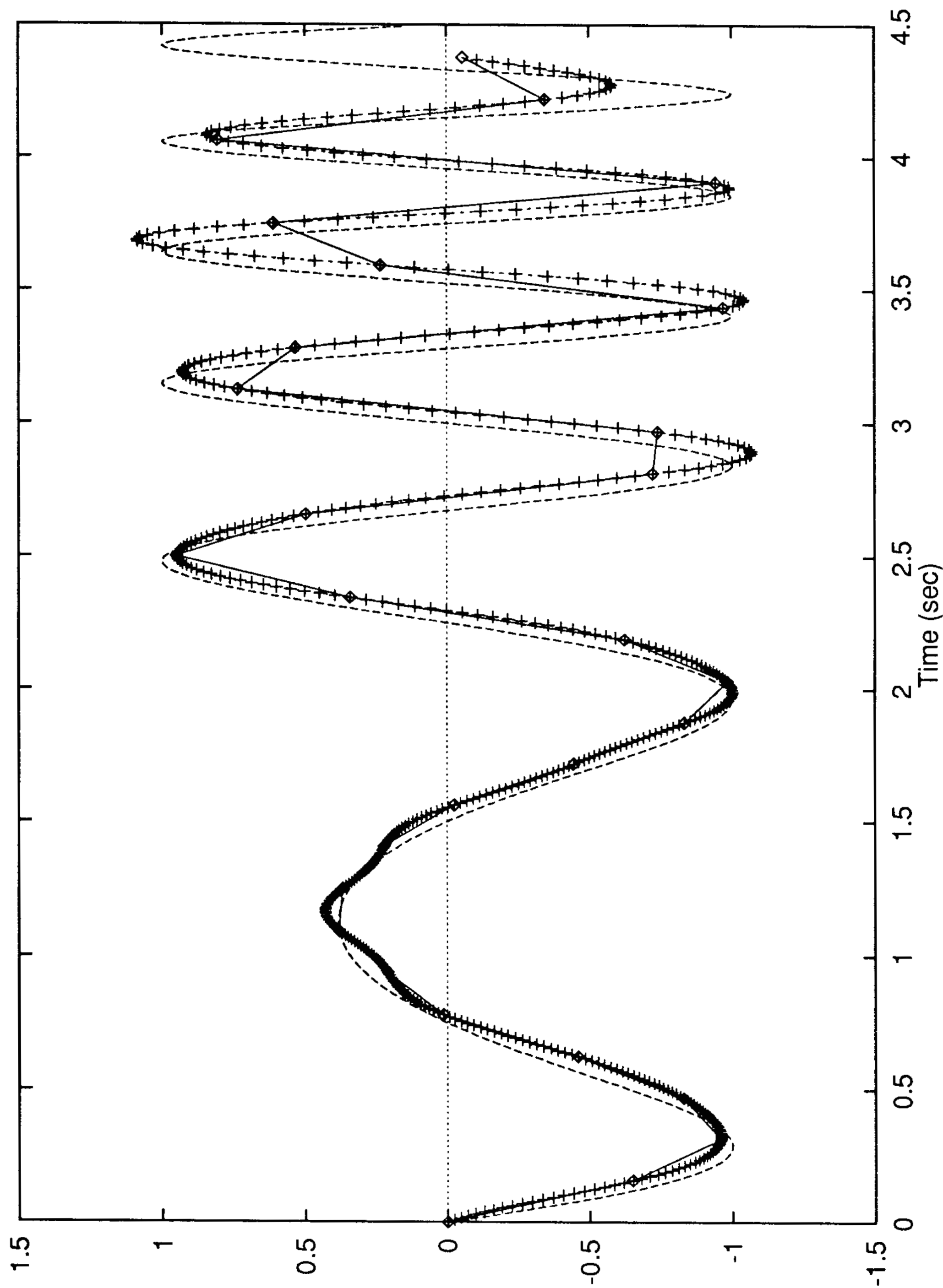
Next time, we will close the loop between our input and output handling by looking at how we can modify our outputs on the basis of the given inputs, in order to provide stable control of a system.

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

Code begins on page 38.
Downloading: freq.fth, gensig.fth, and regen.fth are available at ftp://ftp.Forth.org/pub/Forth/FD/1996

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.

Figure Three. The sampled and reconstructed signal. The dashed line shows the reference signal without the carrier—this what is to be reconstructed. The diamonds show the 6.4 Hz samples obtained by mixing, filtering, and subsampling the reference signal (Figure Two). The pluses show the signal reconstruction obtained by applying the uniform sampling theorem to the sampled data.



References

Bracewell, R.N., 1986; *The Fourier Transform and Its Applications* (McGraw-Hill, New York, 474 pages, ISBN 0-07-007015-6)

Horowitz, P. and W. Hill, 1980; *The Art of Electronics* (Cambridge University Press, Cambridge, 716 pages, ISBN 0-521-23151-5)

Oppenheim, A.V, and R.W. Schafer, 1975; *Digital Signal Processing* (Prentice-Hall, Englewood Cliffs NJ, 586 pages, ISBN 0-13-214635-5)

Listing One. Examples of simple period and frequency counters.

```
\ freq.fth   Simple implementations of a period and frequency counter
\            sampling data on the parallel #STATUS port bit 7

\ This code released to the public domain September 1996 Taygeta Scientific Inc.

\ $Author:   skip  $
\ $Workfile:   freq.fth  $
\ $Revision:   1.0  $
\ $Date:   28 Sep 1996 20:05:14  $
\ ===================================================================================

HEX
378        CONSTANT #DATA
#DATA 1+   CONSTANT #STATUS
#DATA 2 +  CONSTANT #COMMAND
DECIMAL

\ ===================================================================================
\ The following two words are adapted from: Hendtlass, T., 1993; Real Time Forth
\ If you don't have it, GET THIS BOOK!     contact: tim@brain.physics.swin.oz.au

\ An F-PC specific <read_clock>
code <read_clock> ( -- n )
    push ax
    mov ax, # 0
    int 26
    pop ax
    push dx
    next
end-code

: DOWN-COUNTER          \ creates a countdown timer
    CREATE   ( -- )
      2 CELLS ALLOT     \ set aside 2 slots, user value and read value
    DOES>
      <read_clock>      \ read hardware clock
      OVER CELL+ @      \ get last clock value
      OVER -            \ get the change
      2 PICK +!         \ update user value
      OVER CELL+ !      \ save last read value
;

\ ================================================================
\ PC-specific words to set hardware timer to 1193181.667 ticks/second
\ and then later to restore it back to the standard 18.2 ticks/second
\ Note: this won't work in a Windows DOS shell.

HEX

43 CONSTANT TIMER_CONTROL
40 CONSTANT TIMER_0

: initialize_timer ( -- )
    34 TIMER_CONTROL PC!
    0 TIMER_0 PC!
    0 TIMER_0 PC!
;

: restore_timer ( -- )
    36 TIMER_CONTROL PC!
    0 TIMER_0 PC!
    0 TIMER_0 PC!
;
```

```
\ ==================================================================
DECIMAL

: wait_for_low ( -- )      \ wait until #STATUS bit 7 is low
    BEGIN
      #STATUS pc@ 128 AND 0=
    UNTIL
;

: wait_for_high ( -- )      \ wait until #STATUS bit 7 is high
    BEGIN
      #STATUS pc@ 128 AND
    UNTIL
;

: edge_high? ( -- t/f )      \ return status of #STATUS bit 7
    #STATUS pc@ 128 AND
;

VARIABLE accumulate

\ ===============the period counter================================
: PERIOD ( n -- x )      \ n is number of samples, x is average period
    0 accumulate !
    initialize_timer    \ run the timer at a fast rate

     0 DO
        wait_for_low        \ make sure the level is low first
        <read_clock>
        wait_for_high       \ now poll for a rising edge
        <read_clock> -      \ neglecting rollover
        accumulate +!
      LOOP

   restore_timer

   accumulate @
;

\ ===============the frequency counter==========================================
\ depending upon your hardware, this simple counter is good to up to about 20 Khz
DOWN-COUNTER count_down

: FREQUENCY ( n -- x )    \ n is number of timer cycles, x is average freq. in Hz
    0 accumulate !

    wait_for_low       \ make sure the level is low first
    DUP count_down !

    BEGIN
      edge_high?      \ test to see if edge is high
            IF 1 accumulate +!
               wait_for_low   \ make sure level goes back low
            THEN
      count_down @ 0 <=
    UNTIL

    \ convert counts to Hz
    10 *
    accumulate @
    182 ROT */
;
```

Listing Two. The program to generate either the reference signal or the data samples.

```
\ gensig.fth   Generates a reference chirp test signal
\              or, if SAMPLING? is true, generate a sampled signal

\ This is an ANS Forth program requiring:
\         1. The Floating point word set
\         2. The conditional compilation words in the
\            PROGRAMMING-TOOLS wordset
\   There is an environmental dependency in that it is assumed
\   that the float stack is separate from the parameter stack

\ This code released to the public domain September 1996 Taygeta Scientific Inc.

\ $Author:   skip  $
\ $Workfile:   gensig.fth  $
\ $Revision:   1.0  $
\ $Date:   28 Sep 1996 20:04:22  $
\ ================================================================
FALSE CONSTANT SAMPLING?
10.0E0 FCONSTANT F_MIN             \ 10 "Hz" minimum frequency
12.0E0 FCONSTANT F_MAX             \ 12 "Hz" maximum frequency
4.50E0 FCONSTANT SWEEP             \ the frequency sweep time

FVARIABLE DF

6.283185307E0 FCONSTANT TWO_PI
0.0078125E0 FCONSTANT DT              \ 128 "Hz" sample rate

: tone ( -- , F: t f -- x )
    F* TWO_PI F* FSIN
;

: S>F ( x -- , F: -- fx )
    S>D D>F
;

SAMPLING? [IF]

\ 16 CONSTANT DECIMATE                     \ effective sampling at 8 Hz
20 CONSTANT DECIMATE                      \ effective sampling at 6.4 Hz

F_MAX F_MIN F+ 2.0E0 F/ FCONSTANT F_CENTER

: MIX ( -- , F: x t -- x' )
    F_CENTER F* TWO_PI F* FCOS
    F*

    \ account for loss due to the mixer shifting stuff to both a low and high band
    2.0E0 F*
;

 0.01279E0 FCONSTANT COEF_A
-1.65561E0 FCONSTANT COEF_B
 0.70676E0 FCONSTANT COEF_C

FVARIABLE IN_0     0.0E0 IN_0 F!
FVARIABLE IN_1     0.0E0 IN_1 F!
FVARIABLE IN_2     0.0E0 IN_2 F!
FVARIABLE OUT_0    0.0E0 OUT_0 F!
FVARIABLE OUT_1    0.0E0 OUT_1 F!
FVARIABLE OUT_2    0.0E0 OUT_2 F!
```

```
\ first order low-pass Butterworth filter
: FILTER ( -- , F: x -- x' )
    IN_0 F!

    IN_0 F@ IN_2  F@ F+
    IN_1 F@ 2.0E0 F* F+
    COEF_A F*

    OUT_1 F@ COEF_B F* F-
    OUT_2 F@ COEF_C F* F-

    FDUP OUT_0 F!

    \ shift data for next time
    IN_1 F@ IN_2 F!
    IN_0 F@ IN_1 F!

    OUT_1 F@ OUT_2 F!
    OUT_0 F@ OUT_1 F!
;

[ELSE]

1 CONSTANT DECIMATE

: MIX ( -- , F: x t -- x )
    FDROP    \ do nothing for ref. signal
;

: FILTER ( -- , F: x -- x )   ; IMMEDIATE

[THEN]

: freq ( -- , F: t -- f )
\ calculate the frequency for this time
    FDUP       F0< IF FDROP F_MIN EXIT THEN
    SWEEP FOVER F< IF FDROP F_MAX EXIT THEN
    SWEEP F/ DF F@ F* F_MIN F+
;

: gensig ( -- , F: maxt -- )
    DT F/ F>D DROP

    F_MAX F_MIN F- DF F!
    CR

    0.0E0        \ the time
    0 DO
      FDUP FDUP freq
        tone
      FOVER MIX FILTER

      I DECIMATE MOD 0= IF
                          FOVER F. F. CR
                     ELSE
                       FDROP
                     THEN
        DT F+
    LOOP

    FDROP
;

SWEEP gensig bye
```

Statement of Ownership, Management, and Circulation

1. Publication Title: *Forth Dimensions*
2. Publication No. 0002-191
3. Filing Date: Sept. 27, 1996
4. Issue Frequency: bi-monthly
5. No. of Issues Published Annually: 6
6. Annual Subscription Price: $45
7. Complete mailing address of known office of publication: P.O. Box 2154, Oakland, California 94621-0054
8. Complete mailing address of headquarters or general business office of Publisher: same as above
9. Publisher: Forth Interest Group, P.O. Box 2154, Oakland, California 94621-0054. Editor: P.O. Box 2154, Oakland, California 94621-0054.
10. Owner: Forth Interest Group (non-profit), P.O. Box 2154, Oakland, California 94621-0054.
11. Known bondholders, mortgagees, and other security holders owning or holding 1 percent or more of total amount of bonds, mortgages, or other securities: none.
12. The purpose, function, and nonprofit status of this organization and the exempt status for federal income tax purposes: has not changed during preceding 12 months.
13. Publication Name: *Forth Dimensions*
14. Issue Date for Circulation Data Below: Nov.-Dec. 1996

15. Extent and Nature of Circulation:	*Average No. Copies Each Issue During Preceding 12 Months*	*Actual No. Copies of Single Issue Published Nearest to Filing Date*
a. Total No. Copies (*net press run*)	1200	1200
b. Paid and/or Requested Circulation		
(1) Sales through Dealers and Carriers, Street Vendors, and Counter Sales *(not mailed)*	0	0
(2) Paid or Requested Mail Subscriptions *(Include Advertisers' Proof Copies/Exchange Copies)*	1020	914
c. Total Paid and/or Requested Circulation	1020	914
d. Free Distribution by Mail *(Samples, Complimentary, and Other Free)*	10	10
e. Free Distribution Outside the Mail *(Carriers or Other Means)*	0	0
f. Total Free Distribution	10	10
g. Total Distribution	1040	934
h. Copies Not Distributed		
(1) Office Use, Leftovers, Spoiled	160	266
(2) Return from News Agents	0	0
i. Total	1200	1200
Percent Paid and/or Requested Circulation	98.07%	97.85%

Signature and Title of Editor, Publisher, Business Manager, or Owner:
John D. Hall, President, September 27, 1996.

Listing Three. The program to apply the uniform sampling theorem to the sampled data.

```
\ regen.fth   reconstructs the original signal from the sampled signal
\ This is an ANS Forth program requiring:
\         1. The Floating point word set
\         2. The File wordset
\         3. The conditional compilation words in the PROGRAMMING-TOOLS wordset
\         4. The Forth Scientific Library Array words
\         5. The Forth Scientific Library ASCII file I/O words
\   There is an environmental dependency in that it is assumed
\   that the float stack is separate from the parameter stack

\ This code released to the public domain September 1996 Taygeta Scientific Inc.
\ $Author:   skip  $
\ $Workfile:   regen.fth  $
\ $Revision:   1.0  $
\ $Date:   28 Sep 1996 20:04:50  $
\ ==============================================================
S" /usr/local/lib/forth/fsl-util.fth" INCLUDED
S" /usr/local/lib/forth/fileio.fth"   INCLUDED

FALSE CONSTANT STANDALONE
-1 VALUE fin   \ input file handle
-1 VALUE fout \ output file handle

0.0078125E0 FCONSTANT OUT_DT          \ 128 "Hz" output rate
0.15625E0 FCONSTANT DT              \ 6.4 "Hz" sample rate
3.1415926536E0 FCONSTANT PI
6.283185307E0  FCONSTANT TWO_PI
29  CONSTANT NUM_SAMPLES
560 CONSTANT NUM_OUTPUT

NUM_SAMPLES FLOAT ARRAY t{
NUM_SAMPLES FLOAT ARRAY x{   \ the samples

CREATE outbuf 32 ALLOT
CREATE CRLF   2 ALLOT

FVARIABLE PI/T
9 CONSTANT TAB_CHAR    \ TAB character
CREATE TAB 1 ALLOT

STANDALONE [IF]
variable f_index   1 f_index !

: next_file  ( -- c-addr u )
    f_index @ argc >= if
        0 0
    else
        f_index @ argv
        1 f_index +!
    then
;
[ELSE]
: next_file  ( -- c-addr u )
    bl word count
;
[THEN]
: S>F  ( x -- , F: -- fx )
    S>D D>F
;

: }zero  ( n x -- )
    SWAP 0 DO
           DUP I } 0.0E0 F!
    LOOP
    DROP
;
```

The Computer Journal

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

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

TCJ

The Computer Journal

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

```
: sinc ( -- , F: x -- sincx )
   FDUP F0= IF   FDROP 1.0E0
            ELSE FDUP FSIN FSWAP F/
            THEN
;

: estimate ( n -- , F: t -- x )
   PI DT F/ PI/T F!
   0.0E0 FSWAP   ( F: sum t )
   0 DO
       I S>F DT F*
     FOVER FSWAP F-     ( F: sum t t-nT )
       PI/T F@ F*
       sinc
     X{ I } F@ F*
     FROT F+
       FSWAP
   LOOP
   FDROP
;

: print_endline ( -- )
   CRLF
   1               \ for MS-DOS use 2 instead of 1
   fout write-token
;

: print_tab ( -- )
   TAB
   1
   fout write-token
;

: regen ( --<infile outfile>-- )
   10 CRLF   C!  13  CRLF 1+ C!
   TAB_CHAR TAB C!

   next_file
   R/O OPEN-FILE ABORT" unable to open data file"
   TO fin

   next_file
   \ open the output file
   W/O CREATE-FILE ABORT" unable to open output file" TO fout
   CR

   NUM_SAMPLES x{ }zero
   NUM_SAMPLES t{ }zero
   NUM_SAMPLES 0 DO
       I .
       fin get-float  FDUP F. t{ I } F!
     fin get-float  FDUP F. x{ I } F!
       CR
   LOOP
   fin CLOSE-FILE DROP

   NUM_OUTPUT 0 DO
       I S>F OUT_DT F* FDUP outbuf fout write-float
     print_tab

     NUM_SAMPLES  estimate
     outbuf fout write-float
       print_endline
   LOOP
   fout CLOSE-FILE DROP
;
```

Asilomar
FORML CONFERENCE

The original technical conference for professional Forth programmers and users.

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

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

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. 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. By calling attention to the successes and the problems now, before the review process begins, others will repeat your experiments, confirming or refuting your hypotheses.

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.

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

Advance Registration Required • Call FIG Today 510-893-6784

Registration fee for conference attendees includes conference registration, coffee breaks, and notebook of papers submitted, and for everyone rooms Friday and Saturday, all meals including lunch Friday through lunch Sunday, wine and cheese parties Friday and Saturday nights, and use of Asilomar facilities.

Conference attendee in double room—$440 • Non-conference guest in same room—$320 • Children under 18 years old in same room—$190 • Infants under 2 years old in same room—free • Conference attendee in single room—$570

The Asilomar Conference Center combines excellent meeting and comfortable living accommodations with secluded forests on a Pacific Ocean beach. Early registration is recommended, space for this conference is limited.

Forth Interest Group members and their guests are eligible for a ten percent discount on registration fees.

Registration and membership information available by calling, fax or writing to:

Forth Interest Group, PO Box 2154, Oakland, CA 94621
voice 510-893-6784, fax 510-535-1295

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