

# F O R T H

---

D I M E N S I O N S



*Point and Do*

*A Forth Switchblade*

*The Stuttering Context Switch*

*Porting hForth to the StrongARM*

*Linearizing a Thermocouple*





5

**Porting hForth to the StrongARM SA-110 RISC Processor**  
*by Neal Crook*

The author was working for DEC's semiconductor division as an applications engineer and settled upon the idea of doing a port to the 64-bit Alpha RISC processor. But his group won the task of supporting StrongARM chip sales, and he started work on the design of a board that would be used as a hardware verification and evaluation platform for the first StrongARM chip, the SA-110.

11

**The Stuttering Context Switch**  
*by Martin Schaaf*

How to build the context-switching part of a Forth engine? The author had been focussed on optimizing the time-wasting stack-shuffling operations, devising a method of buffering the top three items on the stack and performing stack shuffling in parallel with other operations. Task switching, however, he had to learn about from his plumbing.

12

**Linearizing a Thermocouple with Two-Step Interpolation**  
*by Jerry Avins*

When building a profiling temperature controller for a small oven, one of the necessary details is a way to read a thermocouple that is to indicate temperature in degrees F and be suitable for use in a control loop. Thermocouples are only slightly nonlinear. Nevertheless, a simple way to linearize them also works well with functions that have much greater nonlinearity, and it is presented here.

17

**ANS Appendix to "Finite State Machines in Forth"**  
*by Julian V. Noble*

ANS-compatible code to accompany the author's article (which appeared in our preceding issue), and an erratum to the code that appeared previously in these pages.

19

**A Forth Switchblade**  
*by Rick VanNorman*

An example of a *switch* in Forth is the CASE statement. The execution-time behavior of CASE and OF can be optimized until your system implementor is exhausted, and performance will be similar to that of a C version. So why would anyone want to implement a new switch construct in Forth? For SwiftForth, the reason was the need for extensibility—to be able to define the base structure and to extend it at will. The traditional CASE statement does not lend itself to being extended after it is defined.

23

**Point and Do**  
*by Richard W. "Dick" Fergus*

A pointing device can be very useful to interface the user with the intricacies of a program. Herewith, the author supplies relevant support code for Pygmy Forth although, with minor modifications, they should be applicable to other Forth dialects.

**DEPARTMENTS**

2	<b>OFFICE NEWS</b> Changes on the horizon	26	<b>STANDARD FORTH TOOL BELT</b> Number Conversion and Literals
4	<b>EDITORIAL</b>	29	<b>STRETCHING STANDARD FORTH</b> Only Standard Definitions
15	<b>CROSSWORD</b> — "Stacks"	34	<b>URLs</b> — a selection of on-line Forth resources
16	<b>PRESIDENT'S LETTER</b> Ready for an eFD?	35	<b>SPONSORS &amp; BENEFACTORS</b>

# This and Errata...

Please see this issue's "Office News" for important information about changes taking place to the rates on our mail-order form—current prices will only remain in effect until our revised form can be published (which is planned for the next issue).

We received the following suggestion in response to a plea we issued some months ago for more Forth articles, both in this magazine and in publications directed outside the immediate Forth community:

Dear editor,

An opportunity has come up that could propel Forth to the forefront of computer languages. I am speaking of the Design Your Own Processor™ Tools at:

<http://www.dnai.com/~jfox/fpgakit.htm>

If we get on top of this and write about it, we could be *the* language of reconfigurable computing.

—M. Simon • [msimon@tefbbs.com](mailto:msimon@tefbbs.com)

I hope that both activists and the curious will take note of this and other opportunities to explore, and to point out to others, Forth's suitability in particular application and engineering domains. Waiting to be discovered is a sure way to insularity!

Those who tried Julian Noble's Finite State Machines code (see our preceding issue) might have had a bit of difficulty, and a one-line "fix" is provided in this issue. Julian remarks, "This is a good example of a cautionary tale—why one must never trust a listing printed in a book or journal. (I have long known that scribal errors make it impossible to trust formulas taken from texts and journals, and it looks as though this is the case with program listings as well.)" It's also a good example of the perils of technical publishing, whatever the medium, although we go to great lengths to avoid such things.

As a concluding note for now (we are already working on the next issue and will have more to say then), Fred Behringer's ([behringe@mathematik.tu-muenchen.de](mailto:behringe@mathematik.tu-muenchen.de)) transputer Forth package now is also available from <ftp://ftp.taygeta.com/pub/forth/compilers/native/dos/transputer/> for downloading.

—Marlin Ouverson

Would you like to brush up on your German and, at the same time, get first-hand information about the activities of your Forth friends in Germany?

## Become a member of the German Forth Society ("Deutsche Forth-Gesellschaft")

80 DM (50 US-\$) per year  
or 32 DM (20 US-\$) for students or retirees

Read about programs, projects, vendors, and our annual conventions in the quarterly issues of *Vierte Dimension*. For more information, please contact:

Forth-Gesellschaft e.V.  
Postfach 161204  
18025 Rostock  
e-mail: [SECRETARY@ADMIN.FORTH-EV.DE](mailto:SECRETARY@ADMIN.FORTH-EV.DE)

**Forth Dimensions**  
Volume XX, Number 3  
September 1998 October

Published by the  
**Forth Interest Group**

Editor  
Marlin Ouverson

Circulation/Order Desk  
Trace Carter

*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 (U.S.) \$60 (international). For membership, change of address, and to submit items for publication, the address is:

Forth Interest Group  
100 Dolores Street, suite 183  
Carmel, California 93923  
Administrative offices:  
408-37-FORTH Fax: 408-373-2845

Copyright © 1998 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 \$45/60 per year by Forth Interest Group at 1340 Munras Avenue, Suite 314, Monterey CA 93940. Periodicals postage rates paid at Monterey CA and at additional mailing offices.

POSTMASTER: Send address changes to FORTH DIMENSIONS, 100 Dolores Street, Suite 183, Carmel CA 93923-8665.

# Porting hForth to the StrongARM SA-110 RISC Processor

## 1. Background

Once upon a time I downloaded Julian Noble's FPRIMER.ZIP from a SIMTEL archive and discovered eForth V1.0. I was fascinated by the way eForth used an assembler's macro expansion capability to generate all of the header and dictionary structures for a Forth compiler. I ported eForth to the Z80 (not knowing that this had already been done); I chose the Z80 because I was familiar with it and I had a development environment for it. Doing the Z80 port was not useful, except as a Great Learning Experience.

At the time I started playing with eForth, I was working for Digital Equipment Corporation's semiconductor division (Digital Semiconductor, DS) as an applications engineer. Having completed the eForth port to the Z80, I was casting around for another fun spare-time project, and I naturally settled upon the idea of doing a port to the 64-bit Alpha RISC processor. However, before I could get started on this project, DS took a license for the Advanced RISC Machines Ltd. ARM architecture, and announced that it was developing StrongARM. My group won the task of supporting StrongARM chip sales, and I started work on the design of a board that would be used as a hardware verification and evaluation platform for the first StrongARM chip, the SA-110.

When the board design was completed, I had about a month to spare whilst the board was in layout and manufacture. The SA-110 itself was still in the last stages of design. I took some code examples from another engineer, who was writing the diagnostic and test code, and set about the task of learning ARM assembler programming, with a view to porting eForth to the SA-110.

I debugged the ARM eForth port on an instruction-set simulator, and then on an ARM610 processor evaluation board. Meanwhile, my board had come back from assembly and I had done as much testing as you can do on a processor board when it has no processor. Boards were shipped to Austin, Texas where the SA-110 design team were headquartered, and I eagerly awaited SA-110 prototypes.

Finally, we had word that the SA-110 was due out of fab imminently. A software engineer and I travelled to Austin. We powered the very first SA-110 chip up in the last week of November, 1995 and, within a day, the diagnostics were up and running (well done, Anthony). The code for talking to the debug tools presented more of a problem, and there wasn't much I could do to help. It was time to blow an EPROM on my own account.

A couple of days (and many cycles through the EPROM eraser) later, eForth was up and running (it was the third or fourth program ever to run on SA-110 silicon). It was immediately useful for writing code one-liners to exercise logic and

to measure power consumption. In addition, I added facilities to allow the processor's caches to be turned on and off under software control so we could measure the impact on speed and power consumption.

Overall, eForth proved to be very useful during the course of the project, but in the meantime I had discovered the ANS Forth standard, and I wanted to try some of the features that eForth lacked. I began to modify eForth to bring it in line with the ANS standard. One day, during a bit of web browsing, I came across Dr. Wonyong Koh's hForth[1]. When I saw what Dr. Koh had achieved, using eForth as a starting point, I abandoned eForth and started a port of hForth.

## 2. Problems

There were three basic problems to address:

- Coding low-level routines for the target processor
- Tool chain
- Portability issues in the code

### 2.1 Coding low-level routines for the target processor

hForth is a direct-threaded code (DTC) Forth, and it is designed to be built using a macro assembler. Macros are used in the source code to express Forth constructs like constants, variables, colon definitions, and code definitions. To simplify the porting effort, a minimal number of definitions must be coded in assembler; the remainder are colon definitions. The first step in the porting process is to map registers of the Forth virtual machine to real registers in the target processor. The ARM has 15 general-purpose registers, named R0 through R14; R15 is the program counter. R14 has a special role during subroutine calls: it stores the return address from the subroutine (unlike CISC processors, RISC processors do not tend to have dedicated hardware stack pointers). The instruction set is highly orthogonal (another RISC characteristic), so it makes little difference which register is used for which function. I chose this register assignment:

<u>Name</u>	<u>Register</u>	<u>Description</u>
dsp	R12	Data stack pointer
rsp	R11	Return stack pointer
fpc	R10	Forth virtual machine program counter
tos	R9	Top of (data) stack

The assembler source defines aliases for these four registers, so they can be referred to by name.

Having allocated registers, the second stage in the porting process is to design the virtual machine and code the low-level routines. The main difference between eForth and hForth in these areas is that hForth uses the common technique of

keeping top-of-stack in a processor register. That meant that I could reuse much of my existing code with only minor modifications. In any case, the amount of work is small; in ARM assembler, the longest "required" code definition is about ten lines of assembler code. The hForth source highlights a number of words that should be coded in assembler for speed, but also provides colon definitions that can be used during the initial debug of a new port.

### 2.1.1 Example code fragments

This section shows how hForth definitions are expressed in the assembler source and how the macros expand to generate code for the target. The ARM and 8086 implementations are compared by considering this colon definition:

```
: DOUBLE ( n -- n ) DUP + ;
```

In the source code, this could be represented as a colon definition, which would be portable across processors:

```
$COLON 6, 'DOUBLE', DUBBLE, _SLINK
DEFW DUP, PLUS, EXIT
```

\$COLON is a macro that expands to perform three tasks:

- generate an entry in the name dictionary for the word DOUBLE, and associate an execution token (*xt*) with the name. The value of the *xt* is the assembler label DUBBLE, and its value is a forward reference that will be resolved by the assembler in the usual way. \_SLINK is an assembler variable used to build a link to the previous entry in the name dictionary. By using different variables here, multiple wordlists can be intertwined in the name dictionary.
- generate a label in the code dictionary with the name DUBBLE.
- generate a processor-dependent call to the inner interpreter, DoLIST.

DEFW is an assembler pseudo-op, and is followed by a list of labels. Each label corresponds to an *xt* that will have been created by some other macro expansion. The labels may be forward or backward references because they will all be resolved by the assembler in the usual way. In this example, the values will be the execution tokens for DUP, +, and EXIT, respectively.

For the 8086, the cell size is 16 bits and the opcode size is variable. The call to the inner interpreter is a call to an absolute address. The opcode for CALL is one byte, so this is prefixed with a one-byte NOP to keep the code aligned to a cell boundary. The definition looks like Figure One.

DoLIST is a label, resolved by the assembler. The execution tokens are absolute addresses. The CALL pushes a return address onto the hardware stack and this return address is used by the inner interpreter to access the execution tokens that make up the definition.

For the ARM, the cell size and the opcode size are both 32 bits; the definition looks like Figure Two.

The BL (branch-and-link) instruction is a single 32-bit opcode. Rather than specifying an absolute

address, the branch destination (to the label DoLIST) is encoded as a 24-bit, signed, PC-relative offset within the opcode. This only makes a sub-set of the 32-bit address space accessible, but the range is more than adequate. As before, the execution tokens are absolute addresses. The BL stores a return address in processor register R14 (R14 must be preserved before another BL can be executed). The value of R14 is used by the inner interpreter to access the execution tokens that make up the definition.

The result of using the \$COLON macro is that the colon definition of DOUBLE is portable, even though the macro and result of the macro expansion are not portable. Next, we will look at how the same definition would be expressed as a (processor-dependent) code definition. For the 8086 it looks like this:

```
$CODE 6, 'DOUBLE', DUBBLE, _SLINK
MOV AX, BX
ADD BX, AX
$NEXT
```

While for the ARM it looks like this:

```
$CODE 6, 'DOUBLE', DUBBLE, _SLINK
ADD tos, tos, tos
$NEXT
```

The macro \$CODE expands out to generate a label and a name dictionary entry as before, but does not generate anything in the code dictionary. The macro \$NEXT terminates the definition by returning control to the caller of this definition. Everything in between is expanded by the assembler to generate opcodes for the particular processor. Remember that *tos* is simply an alias for the register R9, which is used to hold the top-of-stack value.

For the 8086, the expansion of \$NEXT generates this code:

```
LODSW ; get the next code address into AX
JMP AX ; jump directly to the code address
```

Whilst for the ARM, the expansion of \$NEXT generates this code:

```
MOV pc, [ fpc ], #CELLL
```

This instruction can be read as "load the PC (i.e., branch to) with the value that is stored in the cell addressed by the current value of *fpc*, and post-increment *fpc* (by the cell-size) to address the subsequent cell."

To understand these examples more clearly, we need to

**Figure One**

NOP	1 byte	} Macro expansion.. processor
CALL DoLIST	1 + 2 bytes	} native code
XT-DUP	2 bytes	}
XT-+	2 bytes	} Executed by inner interpreter
XT-EXIT	2 bytes	} on Forth Virtual Machine

**Figure Two**

BL DoLIST	4 bytes	} Macro expansion.. processor
		} native code
XT-DUP	4 bytes	}
XT-+	4 bytes	} Executed by inner interpreter
XT-EXIT	4 bytes	} on Forth Virtual Machine

see how the inner interpreter, DoLIST, is implemented. Remember from the discussion above that DoLIST takes an input parameter; the address of the first xt to be executed, and that this parameter is passed to the DoLIST code in a processor-specific way:

- For the 8086, DoLIST is entered through a native CALL, and the parameter is passed on the hardware stack, since it is the return address for the call.
- For the ARM, DoLIST is entered through a native BL and the parameter is passed in R14, since this is the return (link) address for the BL.

For the 8086, DoLIST looks like this:

```

$CODE COMPO+6, 'doLIST', DoLIST, _SLINK
SUB BP, 2
MOV [BP], SI ;push return stack
POP SI ;new list address
$NEXT

```

For the ARM, DoLIST looks like Figure Three.

The STR (store) instruction performs a store of the current fpc value onto the return stack, then updates the fpc with the parameter passed in R14. The [rsp, # - CELL]! means, “store at the location addressed by rsp but first decrement rsp by the value of CELL”—in other words, this instruction implements a “push” with rsp as the stack pointer

and fpc as the data.

Now that we’ve seen how definitions are generated by the assembler, there’s one final thing we need to consider: the processor-dependent parts of generating a new definition when hForth is up and running on the target. Again, we will consider the definition for DOUBLE.

The only processor-dependent part of the compilation process is the generation and detection of the call to DoLIST. In hForth, this is handled by the words ?call and xt, .?call is used to check whether a given location contains a direct-threaded code call; it is used for optimisation purposes and by SEE (the word decompiler). xt, takes an xt as a parameter and compiles a direct-threaded code call to that location.

8086 versions, where call-code is 0xE890 (opcode for a NOP followed by a CALL) [see Figure Four.]

ARM versions, where call-code is 0xEB000000 (opcode for BL, with an offset of 0) [see Figure Five.]

The final call to IDflushline is required to support the caches on the SA-110, and it is discussed further below.

## 2.2 Tool chain

eForth and hForth both rely on macro expansion in an 8086 assembler in order to build code and name dictionaries for the target image. Some ports to other processors have continued to use the 8086 macro assembler; in this technique,

the low-level words are hand-assembled and edited into the assembler source files as DEFW (define word) statements. This is somewhat tedious but entirely effective. That technique was unsuitable for the ARM port because the 8086 macro assembler is designed to use 16-bit addresses, whereas the ARM uses 32-bit addresses. Therefore, it was logical to use the assembler and linker in ARM Ltd.’s Software Development Toolkit (SDT). This is where I hit a major problem.

The macros work by repeatedly changing the value of ORG—the position in the target image at which code/data is being generated. They do this because each macro expansion generates stuff in both the code dictionary and name dictionary, and these are in separate memory areas. The problem is that the ARM assembler does not allow ORG to be changed. (At the time I learned this, it came as something of a shock. I have since learnt that it is a common restriction in modern single-pass assemblers.)

The only solution to this problem was to change the structure of the assembler source so that every definition was broken into two parts (one that generated code dictionary entry and one that generated name dictionary entry). Rather than embarking on a major editing

### Figure Three

```

$CODE COMPO+6, 'doLIST', DoLIST, _SLINK
STR fpc, [rsp, # - CELL]! ;preserve forth PC
MOV fpc, R14 ;first xt of definition
$NEXT

```

### Figure Four

```

: ?call DUP @ call-code =
  IF CELL+ DUP @ SWAP CELL+ DUP ROT + EXIT THEN
    \ Direct Threaded Code 8086 relative call
    0 ;

: xt, xhere ALIGNED DUP TOxhere SWAP
  call-code code, \ Direct Threaded Code
  xhere CELL+ - code, ; \ 8086 relative call

```

### Figure Five

```

: ?call DUP @ 0ff000000h AND call-code =
  IF DUP DUP @ 00ffffffh AND \ it's a branch.. get offset
    DUP 007ffffffh > IF
      00ff000000h OR \ sign extend the offset
    THEN
      2 LSHIFT \ convert to byte offset
      + CELL+ CELL+ \ fix up for pipeline prefetch
      SWAP CELL+ SWAP EXIT
    THEN 0 ;

: xt, xhere ALIGNED DUP TOxhere SWAP
  xhere - cell- cell- 2 RSHIFT \ get signed offset
  00ffffffh AND \ mask off high-order sign bits
  call-code OR \ make the opcode
  xhere swap \ remember where it will go
  code, IDflushline ; \ emit it and purge the block

```

session, I used the AWK scripting language to process the assembler source. I ended up with three separate scripts:

- The first script makes syntax changes to the assembler source to suit the ARM assembler
- The second script expands all the macros and generates three output files: one representing the code dictionary, one representing the name dictionary, and one representing a jump table and ASCII strings for the system THROW (error) messages
- The third script reverses the order of the entries in the name dictionary so that entries logically grow down from high memory.

The assembler source is run through these three scripts, and the three output files (code dictionary, reversed name dictionary, and throw table) are concatenated and fed through the ARM assembler. The final stage is to link them using the ARM linker. The entire build process takes about five seconds.

The AWK scripts took some weeks to develop, but I had already made that investment for eForth, and the modifications for hForth were relatively minor (adding the throw table, for example, since this was not present in eForth). The whole process had a major benefit that I did not anticipate: my assembler source file had a relatively small number of changes from the 8086 version. When Dr. Koh made new releases of his code, I was able to use the excellent *ediff* feature in GNU Emacs to view differences between my old code and Dr. Koh's new release, and patch (with a single keystroke) any revision that affected my port.

### 2.3 Portability issues

eForth and hForth were originally written for a 16-bit processor, the 8086, with a 16-bit cell size. My target machine was a 32-bit processor, with a 32-bit cell size. I had found a couple of places in eForth (loop counters in the division and multiplication routines) where the code relied on a 16-bit cell size, and I had changed these to get the 32-bit version working. I checked for these same problems in hForth but I found they had already been abstracted to a constant, cell-size-in-bits. I was later able to conclude that there were no portability issues in the code related to cell size (at least, none that affected the transition from 16 to 32 bits). In addition, as Dr. Koh predicted[1], the multitasker ran without modification.

One area that limited portability was an environment string called `systemID`. As previously described in [1], hForth has three closely associated implementations; ROM model, RAM model, and EXE model. Different assembler source code is used to build each model, and generates the basic kernel of the Forth system. Additional functionality is added by including Forth source files on the running system. The definitions in these files are coded to work correctly for any of the models. Where data structures vary for the different models, `systemID` is tested to see which version to use. Originally, the environment string `systemID` expanded to "8086 ROM Model". For the ARM port, this was changed to "ARM ROM Model", but this stopped the Forth source files from working. Dr. Koh revised hForth to solve this problem; he split the environment string into two parts; CPU (for example, "8086") and Model (for example, "ROM Model"). As a result, most of the high-level files only needed to test Model, and became CPU-independent. The only time where the CPU environment string must be tested is for definitions that use

(CPU-dependent) assembler. For example, see Figure Six.

### 3. Additions to the functionality

In addition to re-coding the low-level routines, I made these modifications to hForth:

- Changed the I/O to support simple terminal I/O and file download.
- Added some primitive code to help in the debug of new ports.
- Added support for processor caches.

#### 3.1 I/O routines

The 8086 hForth is designed to run under MS-DOS. It uses software interrupts to DOS to perform character I/O and file I/O. My target platforms had no underlying operating environment, so I had to write initialisation code for the system memory controller and I/O devices, and character input and output routines to control a UART. I connected to the UART on the target using an RS232 connection from a PC running a terminal emulator.

I added a simple file-download function, which relies on an ASCII file download from the terminal emulator and XON/XOFF flow control within hForth. This facility copies the FILE/HAND technique used by eForth.

All the target boards I ran hForth on had on-board Flash ROM. hForth was stored in ROM but copied into RAM at startup so it would run more quickly. I added Forth definitions to allow me to take a running RAM image of hForth (including all the definitions that had been added interactively or by file download) and program this image back into Flash.

#### 3.2 Debugging

The initial debug of both the eForth and hForth ports was done using ARM Ltd.'s SDT. This includes an instruction set simulator that runs under the control of a debugger to allow single-stepping, source-level debug, and breakpointing.

Both eForth and hForth use a minimal number of words defined in machine code (code definitions); the bulk of the image consists of the name dictionary (which the debugger just treats as data) and threaded lists of execution tokens. By definition, a breakpoint can only be set on an opcode, and for a DTC Forth there is only one opcode in each colon definition: the DTC call to `DoLIST`.

Simply trapping on the call to `DoLIST` leads to multiple unwanted traps. For example, consider a definition that includes this fragment:

```
R> SWAP 2DUP + ALIGNED >R
```

If a breakpoint is set on the call to `DoLIST` for each of these words, the breakpoint would also be triggered if, for example, the definition of `ALIGNED` used `SWAP`. It would be useful to step through each word in turn (and check its effect on stacks and other data areas) without diving down into other definitions. The threaded nature of the code makes it very difficult to step through a particular definition in this way using breakpoints.

Conventional Forth programming philosophy encourages you to test and debug each low-level word and work your way upwards to a complete, debugged program. However, when you are trying to bring up Forth with no particular tools to help you, you have no "test harness" to exercise a word other than the entirety of the Forth compiler.

My solution to this problem was to modify \$NEXT to implement a *micro debugger*, uDebug.

All definitions end with \$NEXT—either directly (code definitions) or indirectly (colon definitions terminating in EXIT, which is itself a code definition). The normal action of \$NEXT is to use the fpc to fetch the xt of the next word and jump to it. The modified action of \$NEXT is to make a jump (not a call) to the routine uDebug. Invoking this modified behavior is a build-time option that requires you to reassemble the code.

In ARM assembler, uDebug looks like Figure Seven.

To invoke uDebug for a particular definition:

1. Set a debugger breakpoint at the DTC call to DoLIST at the start of the definition to be debugged, and run until you hit this breakpoint.
2. Load the location trapfpc with the address of the first xt in the definition to be debugged.
3. Set a debugger breakpoint on the final instruction in the uDebug routine.

When you run the code, the debugger will now trap after the execution of the first xt in the definition. Run again and it will stop after the execution of the second. To disable uDebug, set the location trapfpc to 0.

This technique has a number of limitations:

- It depends upon an xt of 0 being illegal (since this acts as a magic value to turn uDebug off)
- It does not allow you to automatically debug a code stream that includes inline string definitions, or any other kind of inline literal; you must step into the word that includes the definition, then hand-edit the appropriate new value into trapfpc.

These limitations could be overcome by making uDebug more complex—but at a risk of introducing bugs into the debugger code itself. uDebug has now been incorporated into Dr. Koh's hForth source.

Another technique I used early in eforth debug was even simpler: a definition called DXIT, which has behavior identical to EXIT, but with a different xt. To use this to debug a definition:

1. Set a debugger breakpoint on the DTC call at the start of DXIT.
2. In the definition to be debugged, patch the xt of EXIT with the xt of DXIT.

Now when you run the code, the debugger will trap at the end of the definition to be debugged, an ideal point at which to examine the stack effects. A duplicate DoLIST could be used in a similar way but, for the ARM, patching in a BL to DoLIST requires a fiddly calculation of a relative offset.

Once hForth was up and running on my target hardware, I re-coded some colon definitions as code definitions, to improve performance. I started by giving a code definition a different name from its colon definition and debugging it interactively. After testing, I replaced the colon definition with the code definition and reassembled.

### 3.3 Caches

Everything described so far applies equally to SA-110 and any other ARM processor. However, the architecture of the SA-110 caches differs from that of earlier ARM processors. In common with many RISC processors, but unusual for an ARM processor, the SA-110 has a modified Harvard architecture: separate instruction and data caches, but a unified 32-bit address space accessed through a single external bus interface. This cache architecture introduced two problems for the hForth port:

- keeping the I-cache coherent during code generation
- achieving high cache utilisation

#### 3.3.1 Cache coherence

As is usual on RISC processors, the SA-110 has no hardware mechanism to keep the I-cache coherent with the rest

of the system (D-cache and main memory). Therefore, whenever a value is written into memory and that value is to be used as an opcode, the coherence of the caches must be enforced under software control. This has two well-known consequences:

- self-modifying code requires careful attention
- after loading a new executable image into memory, the caches must be flushed before the code can be executed

Forth can be regarded as a special case of self-modifying code, in the sense that an image that is executing makes additions to its own code space. When hForth is running, the only opcode generated is the BL DoLIST at the start of a definition.

### Figure Six

```
CHAR " PARSE CPU" ENVIRONMENT? DROP
CHAR " PARSE 8086" COMPARE
[ IF] DROP
  CODE D-
    BX DX MOV,    AX POP,    BX POP,    CX POP,    AX CX SUB,
    CX PUSH,     DX BX SBB,    NEXT,
  END-CODE
[ ELSE]
  : D-    DNEGATE D+ ;
[ THEN]
```

### Figure Seven

```
uDebug    ldr r0,=AddrTrapfpc
          ldr r1,[ r0]
          cmps r1,fpc          ; compare the stored address with
                               ; the address we're about to get the
                               ; next xt from
          ldrne pc, [ fpc], #CELLL ; not the trap address, so we're done
          add r1,fpc,#CELLL      ; next time trap on the next xt
          str r1, [ r0]
          ldr pc, [ fpc], #CELLL ; make debugger TRAP at this address
```

This is generated by `xt`, and so, for the ARM port, `xt`, was modified by the addition of a call to `IDflushline`. The function of `IDflushline` is to take an address and to force cache coherence at that address. The SA-110 has a write-through data cache and, therefore, the sequence performed by `IDflushline` is:

- clean D-cache entry at this address (force dirty data line to main memory)
- flush I-cache entry (force a cache miss at this address)

Subsequently, an opcode fetch from the address will cause the I-cache to miss and force the opcode to be fetched from main memory.

For a system without caches, or where I-cache coherence is enforced in hardware, `IDflushline` can simply be `DROP`.

### 3.3.2 Cache utilisation

Consider what happens when the colon definition of `DOUBLE` is executed for the first time. Recall that the definition occupies 16 bytes:

```
[ BL DoLIST] [ XT-DUP] [ XT-+] [ XT-EXIT]
```

To start execution of the word, the SA-110's program counter is loaded with the address of the `BL DoLIST`. The SA-110 checks the I-cache to see if a value for this address is present, and cache misses. A cache miss is serviced by loading a naturally aligned block of eight 32-bit words from main memory into the cache (in this case, the I-cache). The size of the block is called the *line size*, and results in seven other 32-bit words being read into the I-cache. Depending upon the alignment of `DOUBLE` in memory, some of these words may be part of the definition of `DOUBLE` or they may be values associated with earlier or later definitions in memory. Once the cache-miss data has been loaded, the SA-110 executes the `BL` and branches to the inner interpreter which will generate a fetch from the address at which `[ XT-DUP]` is stored. This is a *data* fetch, so the SA-110 checks the D-cache and, again, cache misses. Again, the miss is serviced by loading a naturally aligned block of eight 32-bit words into the D-cache. Often, these will be exactly the same eight words already stored in the I-cache.

This example shows that intermingling code and data leads to low cache utilisation; the I-cache is polluted with execution tokens that can only be used as data and, to a lesser extent, the D-cache is polluted with branches to `DoLIST`, which can only be executed as instructions.

Cache utilisation is a "figure of merit" for a piece of code; it is calculated as the proportion of values that, having been loaded into a cache line, are subsequently used at least once before being discarded to make way for some other value. Low cache utilisation reduces performance for two reasons:

- The processor is stalled whilst the cache line is loaded; loading values that never get used wastes processing cycles.
- Compared with an ideal system (one with full cache utilisation), the system performs as though it had a cache that is only a fraction of its actual size.

Intermingled code and data would be more appropriate for a system with a unified cache, but this architecture is rarely used in high-performance systems, because a modified Harvard architecture is an easy way of increasing the instruction/data bandwidth into a processor core.

For the SA-110, the cache utilisation could be improved dra-

matically by changing from a direct-threaded code to a subroutines-threaded code implementation. This would eliminate the `BL DoLIST` at the start of each definition, and change the list of execution tokens in a definition to a list of `BL` instructions. The design of the compiler and decompiler would be complicated slightly, but the whole thing probably could be factored efficiently and incorporated into `hForth` as a build-time option.

## 4. Applications of hForth

My use of `hForth` on SA110-based target systems has been for testing and debugging hardware. Since the ARM port was released, there have been a few sightings of its use elsewhere, including modifications to the build procedure to support the use of the GNU ARM assembler/linker.

## 5. Other projects in progress

The frustration of having to use AWK scripts to preprocess the assembler source file led me to start thinking about other ways to generate an executable image. Several Forth implementations have successfully used C as a source environment, but I was reluctant to go down that path, because the existing structure of `hForth` makes it suitable for processors for which no C compiler is available.

The logical solution is to metacompile `hForth` and thereby do away with any external tool problems. I have a prototype system running on `pfe` (a 32-bit ANS Forth compiler) under Linux. After loading two ANS programs (an ARM assembler and the metacompiler), it is possible to read the `hForth` source (somewhat modified, since the source is now entirely expressed in Forth) and spit out an ARM binary. More about that in another article...

## 6. Conclusions

`hForth` lived up to its author's goal of being easily portable to other processors. If you want a public-domain Forth that runs on an embedded target, it is worthy of serious consideration.

### A. Acknowledgments

I am grateful for Dr. Koh's timely responses to numerous e-mails when I asked questions about various aspects of his implementation that were unclear to me. We should be grateful that Dr. Koh was kind enough to take comments and code fragments from many people and use them to improve the clarity and portability of his source code.

Most of the work I did on porting `hForth` to the SA-110 was done in my private time. However, some of it was also supported by my then-employer, and I am grateful to acknowledge Digital Semiconductor's permission to place all this work in the public domain under the same restrictions as Dr. Koh's original work: *all* commercial and non-commercial uses are granted.

### B. Download

`hForth` packages for the 8086, Z80, and StrongARM are on-line at:

<http://www.taygeta.com/forthcomp.html> or  
<ftp://ftp.taygeta.com/pub/Forth/Compilers/native/dos/hForth>

These packages include an HTML version of Dr. Koh's article from *FD XVIII.2*.

### C. References

- [1] "hForth: a Small, Portable ANS Forth" Wonyong Koh, *FD XVIII.2*.

# The Stuttering Context Switch

My toilet has developed a stuttering problem. While performing the foreground process of flushing, the background process of refilling the tank proceeds in a noisy, stuttering manner. However, the tank still fills in a reasonable time frame. So, being a software kind of guy, I'm willing to live with it.

My toilet's current mode of operation is rather much like the answer to a question asked of me some fifteen years ago, or rather, the answer I should have come up with fifteen years ago. The question I was asked was how to build the context-switching part of a Forth engine. At the time, I was focussed on optimizing the time-wasting stack-shuffling operations. My theory is that the ideal computer-in-the-sky will always have its data available. Time spent finding and getting the data is time wasted! I had come up with a method of buffering the top three items on the stack and performing stack shuffling in parallel with other operations. Task switching, however, had not yet shown up on my radar and the question brought my pattern-matching processor to a complete halt.

Fifteen years of mulling over the problem produced this solution:

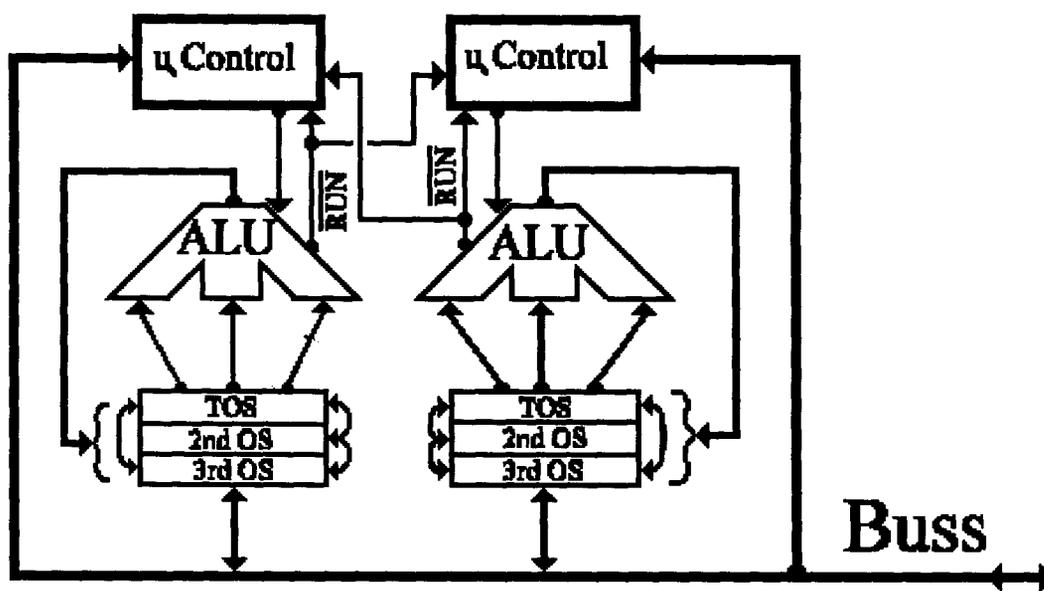
1. Internally, the processor contains two identical processors. (Stop throwing things at me! I am not reinventing the Pentium!)
2. While the foreground processor is running, the background processor is suppressed, except during excessively long instructions, such as `DIVIDE`. During such instructions, the background processor flushes the prior process and loads the next process.
3. If a context switch happens before the background processor has completed loading the next process, the loading of the next process will proceed at full speed, followed by running the next process.

Context switching is thus reduced to switching from one processor to the other, one or two clock ticks. There are, however, some consequences to this design:

1. The flushing and refilling time of the background processor is determined by the mix of instructions running in the foreground processor. In the worst case, none of the previous state will be saved before the next context switch. However, this is the same overhead as on conventional processors.
2. At least one hardware instruction must be a multi-cycle instruction.
3. Programming such a system will include optimizing tradeoffs, such as using `DIVIDE` rather than `SHIFT` or throwing in `NOOP` instructions to allow the background processor time to flush and refill. A small, tightly optimized process could actually degrade performance by interfering with background context switching!

Pattern-matching computers, such as the human brain, are orders of magnitude slower than digital computers at sequential processing. Given enough time, however, they can come up with solutions that, like my toilet's current mode of operating, are elegantly just good enough.

Arrows show the multiple data paths necessary to read, write, and shuffle the top three stack items. These are going to be five port RAM!



Martin Schaaf, M.S. • mas@jps.net  
Alameda, California

Martin Schaaf is a licensed therapist, the webmaster of the Bay Area Association of Disabled Sailors (<http://www.jps.net/personality/baads/baads.html>) and the slave of a 9 1/2 pound calico cat.

# Linearizing a Thermocouple with Two-Step Interpolation

## Problem statement

I am building a profiling temperature controller for a small oven that is used for enameling and the preparation of investment casting molds. One of the necessary details is a way to read a thermocouple that is to indicate temperature in degrees F and be suitable for use in a control loop. Thermocouples are only slightly nonlinear. Nevertheless, a simple way to linearize them also works well with functions that have much greater nonlinearity, and I present it here.

Function approximation is often done by expanding a polynomial. The polynomial can require many terms, even if the function is only modestly nonlinear, and determining the best coefficients can be time consuming. (Thermocouple polynomials are typically ninth order.) Evaluating the polynomial at run time may take too long, especially on the small, slow processors used in many embedded systems. For these systems, a good routine will execute in few cycles using fixed-point arithmetic, and will have adequate accuracy and resolution for the job at hand. It may be important that the resolution be greater than the accuracy. Control systems usually need to differentiate the approximated result, and smooth differentiation requires high resolution.

Polynomial approximations require more terms as the range of the function increases. The technique described divides the function into segments small enough that each segment is adequately characterized by a parabola. For each segment, we must calculate  $ax^2 + bx + c$ . The programmer's task is to determine the proper form of  $x$ , and the values of  $a$  and  $b$ . Errors can be minimized by proper choice of the end points of the segments and the internal value at which the error becomes zero. The method I use here is devoid of subtlety; I simply use some of the leading bits—in this case, three—to define the segment, and construe the rest as a fraction  $0 < f \leq 1$ . I make the end points exact, and force the error to zero also at the midpoint of the segment. With such a "tame" curve as a thermocouple's, more sophistication gives no better results, not even for continuity of slope. To achieve smooth control in the intended application, I want to read to the nearest degree F. I therefore calculate temperature times four.

## Design method

My measurements come from a 12-bit converter, making the full range 4096 counts. This corresponds to 50 mv., given the gain of the converter system, and represents 2250 degrees F. After dividing to identify one of eight segments, the remainder is up to 511. The variable  $x$  now takes the form  $[\text{remainder}/512]$ , and  $c$  is evidently the temperature of the beginning of the segment. It is fairly easy to see what  $a$  and  $b$  must be. The temperature at the end of the segment is  $a + b + c$ , and the middle temperature is  $a/4 + b/2 + c$ . The comput-

ing equation is  $x(ax + b) + c$ , and the multiplications by  $x$  require normalizing divisions. Raw thermocouple data are in tables with one degree increments that give the signal to the nearest microvolt. I use a spreadsheet to interpolate the temperature values from the table and to calculate the coefficients. They are then used in a Forth table.

Tables One and Two are examples of the work. The first column is the count from the converter. The second shows the corresponding millivolts. The next three columns are read from the table and entered by hand; highest temperature not exceeding the millivolt column, millivolts at that temperature, and millivolts one degree higher. The next columns are the interpolated temperature, four times that, rounded to an integer (eight times for Celsius), and the calculated coefficients. (The column marked "4T (c)" shows temperatures at the end and midpoint of segments. These are all needed for subsequent calculations, but only the endpoint values are coefficients.) Working code is also shown.

## Implementation details

Since the error is forced to zero (within the accuracy of the coefficients) at the ends and at the middle of each segment, the obvious places to look for errors are the one- and three-quarter points. I have found no error exceeding  $^{\circ}$  degree, the best that could be expected. Since it is unlikely that any given thermocouple will give a reading closer than two degrees of the reference value, the approximation is clearly better than necessary. It might seem that four segments would be adequate. There is good reason to retain eight, and little incentive to reduce the number. (Naturally, it is desirable to make the number of segments a power of two.) The computation time would be the same in either case, and only 12 cells would be saved. However, the effect on the computation would be drastic. The maximum  $x$  (before normalization) would double,  $b$  would double, and  $a$  would quadruple. It would not then be possible to control round-off. Notice the rounding step in the second line of interpolate, adding 256 to the product:  $2@ \text{ROT} * 256 + 512 / +$ . That keeps the error from being one-sided over the range. In order to do that,  $*/$  cannot be used, so the multiplication must be kept in bounds. With only four segments, that couldn't be done in single precision. The net result would be going from unnecessarily good to unacceptably poor. Without additional tricks, there is nothing in between. Such tricks aren't warranted here.

We could get by with four segments if the precision were limited to one degree; fine for display and adequate for proportional control, but skimpy for the derivative. However, the raw converter data could be used for that. The sensitivity of the thermocouple varies between 20 and 24 microvolts (1.6 to 2 counts) per degree over the range, a variation of  $\pm 10$

percent. In some cases, the gain variation in the derivative might actually be less objectionable than the inevitable staircasing of the linearized value. Clearly, we can't read to <sup>2</sup> degree with a 12-bit converter. What we can do is, given a count, report accurately the temperature that would produce it. Table Two shows the (unimplemented) calculated coefficients to return 8x Celsius temperature directly. I have no reason to believe that its performance would be inferior.

A few variations make it possible to use this two-step interpolation method with more difficult functions. Increasing the number of segments is the most obvious. There is much less to be gained by moving the end points off the true curve than with segmented (piecewise) linear interpolation, but moving the internal point of no error from the midpoint toward a region of greater curvature can sometimes halve the maximum error in the segment. That complicates the computation of the *a*'s and *b*'s, but not inordinately. The endpoint remains  $a + b + c$ , but the internal point of no error becomes  $a/n^2 + b/n + c$ , in which *n* is the point in the segment where the error is to be removed, expressed as a fraction of the segment size. Segmented third-order interpolation would probably handle the most difficult cases encountered in practice.

Common practice might place interpolate as a DOES> in fahrenheit. Separating them allows more than one table to use the same code, provided that the segment sizes are the same. There are two reasons why interpolate does not separate the argument into index and remainder: the address of the table is already on the stack when it begins, complicating the stack, and there may be a need to adjust the index before using it for tables which do not start from zero.

#### Another example

The rough-and-ready sine/cosine generator shown in Listing Two, with the calculations in Table Three, is another example of what this interpolation method can do. It has roughly slide-rule accuracy, enough for many purposes. As written, the routines work for angles of any size, positive or negative. In the application described, this generality is unnecessary. It is merely a side effect of the 8181 AND needed for cosine to work, and of the cyclic nature of the function.

I have a two-phase incremental rotation encoder with 2048

**Table One. Fahrenheit coefficient calculation**

Count	Millivolts	T lower	mV lower	mV upper	T	4T (c)	a	b
0	0.000	32	0.000	0.022	32.00	128	-8	1108
256	3.125	169	3.104	3.127	169.91	680		
512	6.250	307	6.249	6.271	307.05	1228	-26	1137
768	9.375	447	9.363	9.385	447.55	1790		
1024	12.500	584	12.484	12.505	584.76	2339	-14	1083
1280	15.625	719	15.622	15.646	719.13	2877		
1536	18.750	852	18.749	18.772	852.04	3408	-2	1059
1792	21.875	984	21.872	21.895	984.13	3937		
2048	25.000	1116	24.996	25.020	1116.17	4465	14	1055
2304	28.125	1249	28.124	28.148	1249.04	4996		
2560	31.250	1383	31.237	31.260	1383.57	5534	20	1086
2816	34.375	1520	34.366	34.389	1520.39	6082		
3072	37.500	1659	34.480	37.502	1660.00	6640	26	1125
3328	40.625	1802	40.619	40.640	1802.29	7209		
3584	43.750	1947	43.734	43.756	1947.73	7791	36	1174
3840	46.875	2096	46.861	46.881	2096.70	8387		
4096	50.000	2250	49.996	50.016	2250.20	9001		

**Table Two. Celsius coefficient calculation**

Count	Millivolts	T lower	mV lower	mV upper	T	8T (c)	a	b
0	0.000	0	0.000	0.050	0.00	0	-8	1230
256	3.125	76	3.100	3.141	76.61	613		
512	6.250	152	6.218	6.258	152.80	1222	-32	1266
768	9.375	230	9.341	9.381	230.85	1847		
1024	12.500	307	12.498	12.539	307.05	2456	-14	1203
1280	15.625	381	15.594	15.636	381.74	3054		
1536	18.750	455	18.725	18.768	455.58	3645	14	1167
1792	21.875	528	21.834	21.876	528.98	4232		
2048	25.000	603	24.987	25.029	603.31	4826	30	1151
2304	28.125	676	28.12	28.162	676.12	5409		
2560	31.250	750	31.214	31.256	750.86	6007	34	1199
2816	34.375	826	34.339	34.380	826.88	6615		
3072	37.500	904	34.484	37.524	904.99	7240	36	1238
3328	40.625	983	40.605	40.645	983.50	7868		
3584	43.750	1064	43.739	43.777	1064.29	8514	38	1305
3840	46.875	1147	46.873	46.910	1147.05	9176		
4096	50.000	1232	49.998	50.024	1232.08	9857		

pulses per turn on each phase. Such encoders can produce errors if they move (or vibrate) around a single transition, and my (patented) circuit to prevent that automatically provides double or quadruple resolution. It is not magic: the four states of the two phases already contain the extra information. This is not the place for hardware discussion, but I will be happy to respond privately to any who want to know how to do this with two XOR gates in front of the a counter, or with software.

Some day, this encoder may be used in a robot arm, where trigonometry would be needed to calculate the hand position. It will be convenient to get sines and cosines directly, rather than to determine the quadrant as a preliminary. I therefore generate the values over a full turn, with the count of 8192 representing 360 degrees. The values returned are 512 times actual, allowing nine bits of precision, about three decimal digits; that is as much as can be had without modifying interpolate. interpolate also dictates segments of 512 counts spanning 22.5 degrees, so that one turn requires sixteens of them.

The computed table entries make the error zero at the center of the segment, and no significant improvement seems possible, at least as far as I have investigated. The "corrected"

table entries in the listing slightly increase the average error over the entire segment, but provide better continuity of slope at the peaks.

### Listing One

```

: interpolate ( rem index adr -- value )
  SWAP 3 cells * + 2DUP      ( rem adr rem adr )
  2@ ROT * 256 + 512 / +    ( rem adr partial )
  ROT 512 * /                ( adr offset )
  SWAP 2 cells + @ + ;      ( n*temperature )

CREATE fahrenheit ( -- adr )      -8 , 1108 , 128 ,
  ( 4 times actual temperature ) -26 , 1137 , 1228 ,
  -14 , 1083 , 2339 ,
  -2 , 1059 , 3408 ,
  14 , 1055 , 4465 ,
  20 , 1086 , 5534 ,
  26 , 1125 , 6640 ,
  36 , 1174 , 7791 ,

: >temp ( n -- 4*temperature )
  512 /MOD DUP 0 8 WITHIN
  IF fahrenheit interpolate
  ELSE ABORT" Out of range."
  \Real code will shut down.
  THEN ;

\ Words for testing:

: mv 4096 25 * / 1+ 2/ ;

: c >temp 4 /mod 1 .R ASCII . EMIT 25 * . ;

: t mv c ;

```

### Listing Two

```

CREATE sine-table ( -- adr )
  -8 , 204 , 0 ,
  -20 , 186 , 196 ,
  -34 , 145 , 362 ,
  ( Corrected segment ) -37 , 78 , 473 ,
  \ Computed segment -38 , 77 , 473 ,
  ( Corrected segment ) -39 , 0 , 512 ,
  \ Computed segment -38 , -1 , 512 ,
  -34 , -77 , 473 ,
  -20 , -146 , 362 ,
  -8 , -188 , 196 ,
  8 , -204 , 0 ,
  20 , -186 , -196 ,
  34 , -145 , -362 ,
  ( Corrected segment ) 37 , -78 , -473 ,
  \ Computed segment 38 , -77 , -473 ,
  ( Corrected segment ) 38 , 1 , -512 ,
  \ Computed segment 39 , 0 , -512 ,
  34 , 77 , -473 ,
  20 , 146 , -362 ,
  8 , 188 , -196 ,

: sin ( n -- 512*sine )
  8191 AND
  512 /MOD
  sine-table interpolate ;

: cos ( n -- 512*cosine ) 2048 + sin ;

```

**Table Three. Sinecoefficient calculation**

Degrees	Count	Sine (c)	a	b
0.00	0	0	-8	204
11.25	256	100		
22.50	512	196	-20	186
33.75	768	284		
45.00	1024	362	-34	145
56.25	1280	426		
67.50	1536	473	-38	77
78.75	1792	502		
90.00	2048	512	-38	-1
101.25	2304	502		
112.50	2560	473	-34	-77
123.75	2816	426		
135.00	3072	362	-20	-146
146.25	3328	284		
157.50	3584	196	-8	-188
168.75	3840	100		
180.00	4096	0	8	-204
191.25	4352	-100		
202.50	4608	-196	20	-186
213.75	4864	-284		
225.00	5120	-362	34	-145
236.25	5376	-426		
247.50	5632	-473	38	-77
258.75	5888	-502		
270.00	6144	-512	38	1
281.25	6400	-502		
292.50	6656	-473	34	77
303.75	6912	-426		
315.00	7168	-362	20	146
326.25	7424	-284		
337.50	7680	-196	8	188
348.75	7936	-100		
360.00	8192	0		

### Summary

This is a useful (but not earthshaking) way to produce arbitrary functions by segmented second-order interpolation. The accuracy is modest, but enough for many applications, and can approach all that can be expected from integer calculation. Computation time is much less than for ordinary polynomial expansions of the same accuracy (which usually need many more terms), especially on processors without cell-wide hardware multipliers. The examples shown are for 16-bit systems. Of course, 32-bit systems can directly extend the method to much higher precision, but they need more segments to attain it. I have shown that even with 16 bits, the method provides as much accuracy as is useful for thermometry, and accurate enough trigonometry for most control applications.

### Acknowledgments: sorry!

The idea of making interpolate a separate word came from reading Julian V. Noble's recent "Finite State Machines in Forth" (*Forth Dimensions* XX.2). I invented the rest about 15 years ago out of necessity, for use on a 12 MHz 8086 that would not otherwise have been fast enough, despite its built-in assembly-coded polynomial evaluator. I need to use it again, and polished it out of pride of craftsmanship. It has likely been done many times by many others, but I don't know who or when. Priority is hereby ceded to all who wish to claim it.

# Stacks

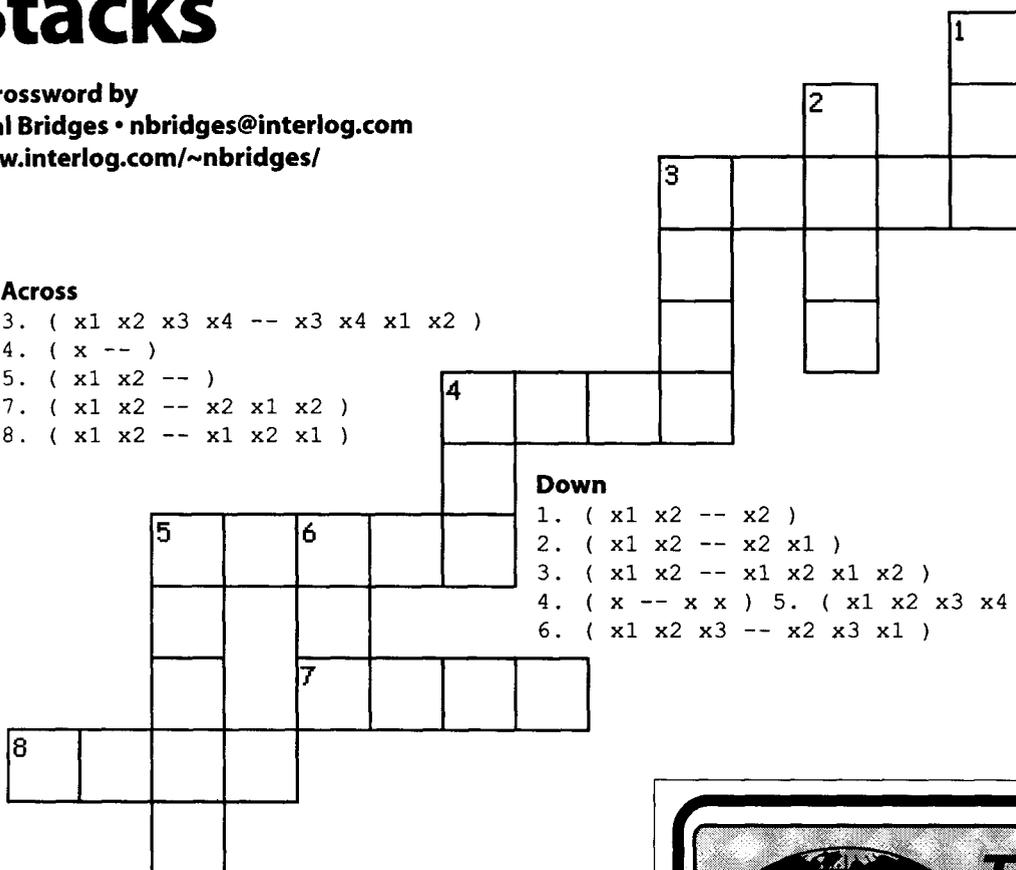
A crossword by  
 Neal Bridges • nbridges@interlog.com  
 www.interlog.com/~nbridges/

### Across

- 3. ( x1 x2 x3 x4 -- x3 x4 x1 x2 )
- 4. ( x -- )
- 5. ( x1 x2 -- )
- 7. ( x1 x2 -- x2 x1 x2 )
- 8. ( x1 x2 -- x1 x2 x1 )

### Down

- 1. ( x1 x2 -- x2 )
- 2. ( x1 x2 -- x2 x1 )
- 3. ( x1 x2 -- x1 x2 x1 x2 )
- 4. ( x -- x x ) 5. ( x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2 )
- 6. ( x1 x2 x3 -- x2 x3 x1 )





## 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

# Growth and Changes

It has been some time since your president has written to you, and a lot has happened, so now is a good time to update you on where things are and where they are going.

I think 1998 ended on a downside for pretty much everybody. This includes FIG. We started the year with a steadily growing membership, starting with just under 900 members. However, at the end of the year the membership dropped substantially and is now about 700. In order to improve this situation, the FIG Board of Directors has decided upon several changes that we hope will help. But remember, FIG is a member-driven organization; if you want FIG to go in a certain direction, please tell us and we will do what we can.

**We have decided to add a new FIG membership category, the *e-member*. An *e-member* will have all the benefits of a regular member, except ... [they obtain] an electronic copy of *Forth Dimensions*... look for an announcement in *FD* and on the web site.**

First, the Board itself has changed. Jeff Fox and Nick Soltseff are no longer on the Board, and new member Randy Leberknight has been added. This new Board membership, and the long interval since the last Board elections, has prompted the current Board to call for new elections. The Forth Interest Group will hold elections for a new Board of Directors, consisting of nine members, in June 1999. The November/December issue of *Forth Dimensions* will contain the official announcement of the elections and the date. The Board has appointed a nominating committee to select nine nominees. In addition to the nominees that the committee submits, you, the members of FIG, can also nominate someone. The requirements are that the candidate be a current

member of FIG and obtain 25 signatures of other FIG members in a nominating petition which must be delivered to the FIG office 90 days before the election.

We have also decided to add a new membership category, the *e-member*. An *e-member* will have all the benefits of a regular member, except they do not receive a mailed copy of *Forth Dimensions*. Instead, an *e-member* obtains an electronic copy of *Forth Dimensions* (in PDF format) through the web site. We are working out the logistical details of this now, look for an announcement in *FD* and on the web site for when it becomes available.

We keep enhancing the web and FTP sites, with lots of help from you. We get an average of 700 accesses (not just hits) per day from all over the world. The demand for Forth and information about Forth continues to be quite vigorous.

Interest in Forth will contribute to interest in FIG. Helping keep Forth visible is vital. We desperately need authors. Not only do we need authors for *Forth Dimensions*, but also for other journals (*Dr. Dobbs* has had a few Forth articles in the last couple of years, and *Embedded Systems Journal* has given Forth the occasional nod). Even more important, we need Forth books! I can go to my local Borders bookstore and pick up more books on Rexx than on Forth! This needs to change. I am working on a couple of writing projects and would be happy to hear of what others are doing. When you are able to get Forth noticed, make sure it's a positive experience. Be professional about it, use a consistent coding style, provide comments that are useful enough for someone else (!) to maintain the code, and document both the design and the implementation.

The Forth Interest Group needs your help in moving forward. Please remember that the FIG office is run by contributed and volunteer labor—there is no paid staff. This means that sometimes not everything that needs to be done can actually get done. Consequently, a major contribution that you can make to FIG is to volunteer to help out.

# Finite State Machines in Forth

*Editor's note: Following is ANS compatible code to accompany the author's paper which was published in our preceding issue. A correction to the code that appeared in the original paper appears on the following page.*

```
\ code to create state machines from tabular representations

\ If needed, : PERFORM @ EXECUTE ;

: || ' , ' , ; \ add two xt's to data field
: wide 0 ; \ aesthetic, initial state = 0
: fsm: ( width state --) \ define fsm
  CREATE , ( state) , ( width in double-cells) ;

: ;fsm DOES> ( x col# adr -- x' )
  DUP >R 2@ ( x col# width state)
  * + ( x col#+width*state )
  2* 2 + CELLS ( x relative offset )
  R@ + ( x adr[ action] )
  DUP >R ( x adr[ action] )
  PERFORM ( x' )
  R> CELL+ ( x' adr[ update] )
  PERFORM ( x' state' )
  R> ! ; ( x' ) \ update state

\ set fsm's state, as in: 0 >state fsm-name
: >state POSTPONE defines ; IMMEDIATE ( state "fsm-name" --)

: state: ( "fsm-name" -- state) \ get fsm's state
  'dfa \ get dfa
  POSTPONE LITERAL POSTPONE @ ; IMMEDIATE

0 CONSTANT >0 3 CONSTANT >3 6 CONSTANT >6 \ these indicate state
1 CONSTANT >1 4 CONSTANT >4 7 CONSTANT >7 \ transitions in tabular
2 CONSTANT >2 5 CONSTANT >5 \ representations
\ end fsm code
```

The automatic conversion tables are useful but not necessary for fast conversion of input to column numbers (in the state table).

```
\ Automatic conversion tables
: table: ( #bytes -- )
  CREATE HERE OVER ALLOT SWAP 0 FILL.
  DOES> + C@ ;

: install ( col# adr char.n char.1 -- ) \ fast fill
  SWAP 1+ SWAP DO 2DUP I + C! LOOP 2DROP ;
\ end automatic conversion tables
```

## Errata

# Finite State Machines in Forth

I am grateful to Jerry [Avins] for pointing out a lacuna in the FSM code that appeared in *FD*. I hasten to add the same line is missing from the code that appeared in *JFAR* (<http://www.jfar.org/article001.html>).

Here is how the word `;FSM` should actually have appeared, and my heartfelt apologies to anyone who was inconvenienced by the error (except Jerry Avins, who owes me a beer for providing him with a wonderful learning experience).

```
: ;FSM  DOES>                ( col# adr - )
      DUP >R  2@                ( - x col# width state)
      *  +                    ( - x col#+width*state )
      2*  2 +  CELLS           ( - x relative offset )

\ the following line was missing
      R@  +                    ( - x offset-to-action )
\ I sure am sorry.

      DUP >R                    ( - x offset-to-action )
      PERFORM                   ( - x' )
      R>  CELL+                 ( - x' offset-to-update )
      PERFORM                   ( - x' state' )
      R>  !  ;                  ( x' ) \ update state
```

I have separated the `DOES>` portion from the `CREATE` section of the FSM compiler:

```
: FSM:  ( width 0 - )      CREATE  , , ;
```

...following a suggestion from Morgenstern in an old *FD*. (I think that is the right reference.) It is not necessary to do this, and the code Jerry sent me keeps this in the `FSM:` definition. *De gustibus non disputandum est*.

—Julian V. Noble • [jvn@virginia.edu](mailto:jvn@virginia.edu)

# FORTH INTEREST GROUP MAIL ORDER FORM

**HOW TO ORDER:** Complete form on back page and send with payment to the Forth Interest Group. All items have one price. Enter price on order form and calculate shipping & handling based on location and total.

## FORTH DIMENSIONS BACK VOLUMES

A volume consists of the six issues from the volume year (May–April).

**Volume 1 *Forth Dimensions* (1979–80) 101 – \$35**

Introduction to FIG, threaded code, TO variables, fig-Forth.

**Volume 6 *Forth Dimensions* (1984–85) 106 – \$35**

Interactive editors, anonymous variables, list handling, integer solutions, control structures, debugging techniques, recursion, semaphores, simple I/O words, Quicksort, high-level packet communications, China FORML.

**Volume 7 *Forth Dimensions* (1985–86) 107 – \$35**

Generic sort, Forth spreadsheet, control structures, pseudo-interrupts, number editing, Atari Forth, pretty printing, code modules, universal stack word, polynomial evaluation, F83 strings.

**Volume 8 *Forth Dimensions* (1986–87) 108 – \$35**

Interrupt-driven serial input, database functions, TI 99/4A, XMODEM, on-line documentation, dual CFAs, random numbers, arrays, file query, Batcher's sort, screenless Forth, classes in Forth, Bresenham line-drawing algorithm, unsigned division, DOS file I/O.

**Volume 9 *Forth Dimensions* (1987–88) 109 – \$35**

Fractal landscapes, stack error checking, perpetual date routines, headless compiler, execution security, ANS-Forth meeting, computer-aided instruction, local variables, transcendental functions, education, relocatable Forth for 68000.

**Volume 10 *Forth Dimensions* (1988–89) 110 – \$35**

dBase file access, string handling, local variables, data structures, object-oriented Forth, linear automata, stand-alone applications, 8250 drivers, serial data compression.

**Volume 11 *Forth Dimensions* (1989–90) 111 – \$35**

Local variables, graphic filling algorithms, 80286 extended memory, expert systems, quaternion rotation calculation, multiprocessor Forth, double-entry bookkeeping, binary table search, phase-angle differential analyzer, sort contest.

**Volume 12 *Forth Dimensions* (1990–91) 112 – \$35**

Floored division, stack variables, embedded control, Atari Forth, optimizing compiler, dynamic memory allocation, smart RAM, extended-precision math, interrupt handling, neural nets, Soviet Forth, arrays, metacompilation.

**Volume 13 *Forth Dimensions* (1991–92) 113 – \$35**

**Volume 14 *Forth Dimensions* (1992–93) 114 – \$35**

**Volume 15 *Forth Dimensions* (1993–94) 115 – \$35**

**Volume 16 *Forth Dimensions* (1994–95) 116 – \$35**

**Volume 17 *Forth Dimensions* (1995–96) 117 – \$35**

**EW Volume 18 *Forth Dimensions* (1996–97) 118 – \$35**

## FORML CONFERENCE PROCEEDINGS

**FORML (Forth Modification Laboratory) is an educational forum for sharing and discussing new or unproven proposals intended to benefit Forth, and is for discussion of technical aspects of applications in Forth. Proceedings are a compilation of the papers and abstracts presented at the annual conference. FORML is part of the Forth Interest Group.**

**1981 FORML PROCEEDINGS 311 – \$45**

CODE-less Forth machine, quadruple-precision arithmetic, overlays, executable vocabulary stack, data typing in Forth, vectored data structures, using Forth in a classroom, pyramid files, BASIC, LOGO, automatic cueing language for multimedia, NEXOS – a ROM-based multitasking operating system. 655 pp.

**1982 FORML PROCEEDINGS 312 – \$30**

Rockwell Forth processor, virtual execution, 32-bit Forth, ONLY for vocabularies, non-IMMEDIATE looping words, number-input wordset, I/O vectoring, recursive data structures, programmable-logic compiler. 295 pp.

**1983 FORML PROCEEDINGS 313 – \$30**

Non-Von Neuman machines, Forth instruction set, Chinese Forth, F83, compiler & interpreter co-routines, log & exponential function, rational arithmetic, transcendental functions in variable-precision Forth, portable file-system interface, Forth coding conventions, expert systems. 352 pp.

**1984 FORML PROCEEDINGS 314 – \$30**

Forth expert systems, consequent-reasoning inference engine, Zen floating point, portable graphics wordset, 32-bit Forth, HP71B Forth, NEON – object-oriented programming, decompiler design, arrays and stack variables. 378 pp.

**1986 FORML PROCEEDINGS 316 – \$30**

Threading techniques, Prolog, VLSI Forth microprocessor, natural-language interface, expert system shell, inference engine, multiple-inheritance system, automatic programming environment. 323 pp.

**1988 FORML PROCEEDINGS 318 – \$40**

*Includes 1988 Australian FORML.* Human interfaces, simple robotics kernel, MODUL Forth, parallel processing, programmable controllers, Prolog, simulations, language topics, hardware, Wil's workings & Ting's philosophy, Forth hardware applications, ANS Forth session, future of Forth in AI applications. 310 pp.

**1989 FORML PROCEEDINGS 319 – \$40**

*Includes papers from '89 euroFORML.* Pascal to Forth, extensible optimizer for compiling, 3D measurement with object-oriented Forth, CRC polynomials, F-PC, Harris C cross-compiler, modular approach to robotic control, RTX recompiler for on-line maintenance, modules, trainable neural nets. 433 pp.

**1992 FORML PROCEEDINGS 322 – \$40**

Object-oriented Forth based on classes rather than prototypes, color vision sizing processor, virtual file systems, transparent target development, signal-processing pattern classification, optimization in low-level Forth, local variables, embedded Forth, auto display of digital images, graphics package for F-PC, B-tree in Forth 200 pp.

**1993 FORML PROCEEDINGS 323 – \$45**

*Includes papers from '92 euroForth and '93 euroForth Conferences.* Forth in 32-bit protected mode, HDTV format converter, graphing functions, MIPS eForth, umbilical compilation, portable Forth engine, formal specifications of Forth, writing better Forth, Holon – a new way of Forth, FOSM – a Forth string matcher, Logo in Forth, programming productivity. 509 pp.

**1994–1995 FORML PROCEEDINGS (in one volume!) 325 – \$50**

## BOOKS ABOUT FORTH

**ALL ABOUT FORTH**, 3rd ed., June 1990, Glen B. Haydon 201 - \$90

Annotated glossary of most Forth words in common use, including Forth-79, Forth-83, F-PC, MVP-Forth. Implementation examples in high-level Forth and/or 8086/88 assembler. Useful commentary given for each entry. 504 pp.

**eFORTH IMPLEMENTATION GUIDE**, C.H. Ting 215 - \$25

eForth is a Forth model designed to be portable to many of the newer, more powerful processors available now and becoming available in the near future. 54 pp. (w/disk)

**Embedded Controller FORTH, 8051**, William H. Payne 216 - \$76

Describes the implementation of an 8051 version of Forth. More than half of this book is composed of source listings (w/disks C050) 511 pp.

**F83 SOURCE**, Henry Laxen & Michael Perry 217 - \$20

A complete listing of F83, including source and shadow screens. Includes introduction on getting started. 208 pp.

**F-PC USERS MANUAL** (2nd ed., V3.5) 350 - \$20

Users manual to the public-domain Forth system optimized for IBM PC/XT/AT computers. A fat, fast system with many tools. 143 pp.

**F-PC TECHNICAL REFERENCE MANUAL** 351 - \$30

A must if you need to know F-PC's inner workings. 269 pp.

**THE FIRST COURSE**, C.H. Ting 223 - \$25

This tutorial goal exposes you to the minimum set of Forth instructions you need to use Forth to solve practical problems in the shortest possible time. "...This tutorial was developed to complement *The Forth Course* which skims too fast on the elementary Forth instructions and dives too quickly in the advanced topics in an upper-level college microcomputer laboratory ..." A running F-PC Forth system would be very useful. 44 pp.

**THE FORTH COURSE**, Richard E. Haskell 225 - \$25

This set of 11 lessons is designed to make it easy for you to learn Forth. The material was developed over several years of teaching Forth as part of a senior/graduate course in the design of embedded software computer systems at Oakland University in Rochester, Michigan. 156 pp. (w/disk)

**FORTH NOTEBOOK**, Dr. C.H. Ting 232 - \$25

Good examples and applications - a great learning aid. polyFORTH is the dialect used, but some conversion advice is included. Code is well documented. 286 pp.

**FORTH NOTEBOOK II**, Dr. C.H. Ting 232a - \$25

Collection of research papers on various topics, such as image processing, parallel processing, and miscellaneous applications. 237 pp.

### "We're Sure You Wanted To Know..."

**Forth Dimensions, Article Reference** 151 - \$4  
An index of Forth articles, by keyword, from *Forth Dimensions* Volumes 1-15 (1978-94).

**FORML, Article Reference** 152 - \$4  
An index of Forth articles by keyword, author, and date from the FORML Conference Proceedings (1980-92).

Fast service by fax: 408.373.2845

**FORTH PROGRAMMERS HANDBOOK**, 260 - \$57  
Edward K. Conklin and Elizabeth D. Rather

This reference book documents all ANS Forth wordsets (with details of more than 250 words), and describes the Forth virtual machine, implementation strategies, the impact of multitasking on program design, Forth assemblers, and coding style recommendations.

**EXCITING  
NEW TITLE!**

**INSIDE F-83**, Dr. C.H. Ting 235 - \$25

Invaluable for those using F-83. 226 pp.

**OBJECT-ORIENTED FORTH**, Dick Pountain 242 - \$37

Implementation of data structures. First book to make object-oriented programming available to users of even very small home computers. 118 pp.

**STARTING FORTH** (2nd ed.) Limited Reprint, Leo Brodie 245a - \$50

In this edition of *Starting Forth*—the most popular and complete introduction to Forth—syntax has been expanded to include the Forth-83 Standard. (*The original printing is now out of stock, but we are making available a special, limited-edition reprint with all the original content.*) 346 pp.

**LIMITED  
TIME!**

**THINKING FORTH**, Leo Brodie 255 - \$35

*Back by popular demand!* To program intelligently, you must first think intelligently, and that's where *Thinking Forth* comes in. The bestselling author of *Starting Forth* is back again with the first guide to using Forth for applications. This book captures the philosophy of the language, showing users how to write more readable, better maintainable applications. Both beginning and experienced programmers will gain a better understanding and mastery of topics like Forth style and conventions, decomposition, factoring, handling data, simplifying control structures. And, to give you an idea of how these concepts can be applied, *Thinking Forth* contains revealing interviews with users and with Forth's creator Charles H. Moore. Reprint of original, 272pp.

**WRITE YOUR OWN PROGRAMMING LANGUAGE USING C++**, Norman Smith 270 - \$16

This book is about an application language. More specifically, it is about how to write your own custom application language. The book contains the tools necessary to begin the process and a complete sample language implementation. (Guess what language!) Includes disk with complete source. 108 pp.

**WRITING FCODE PROGRAMS** 252 - \$52

This manual is for designers of SBus interface cards and other devices that use the FCode interface language. It assumes familiarity with SBus card design requirements and Forth programming. Discusses SBus development for OpenBoot 1.0 and 2.0 systems. 414 pp.

### LEVELS OF MEMBERSHIP

Your standard membership in the Forth Interest Group brings *Forth Dimensions* and participation in FIG's activities—like members-only sections of our web site, discounts, special interest groups, and more. But we hope you will consider joining the growing number of members who choose to show their increased support of FIG's mission and of Forth itself.

Ask about our *special incentives* for corporate and library members, or become an individual benefactor!

Company/Corporate - \$125

Library - \$125

Benefactor - \$125

Standard - \$45 (add \$15 for non-US delivery)

**Forth Interest Group**

See contact info on mail-order form, or send e-mail to:

[office@forth.org](mailto:office@forth.org)

## DISK LIBRARY

### Contributions from the Forth Community

The "Contributions from the Forth Community" disk library contains author-submitted donations, generally including source, for a variety of computers & disk formats. Each file is designated by the author as public domain, shareware, or use with some restrictions. This library does not contain "For Sale" applications. To submit your own contributions, send them to the FIG Publications Committee.

**FLOAT4th.BLK V1.4** Robert L. Smith **C001 - \$8**

Software floating-point for fig-, poly-, 79-Std., 83-Std. Forths. IEEE short 32-bit, four standard functions, square root and log.

★★★ IBM, 190Kb, F83

**Games in Forth** **C002 - \$6**

Misc. games, Go, TETRA, Life... Source.

★ IBM, 760Kb

**A Forth Spreadsheet**, Craig Lindley **C003 - \$6**

This model spreadsheet first appeared in *Forth Dimensions* VII/1,2. Those issues contain docs & source.

★ IBM, 100Kb

**Automatic Structure Charts**, Kim Harris **C004 - \$8**

Tools for analysis of large Forth programs, first presented at FORML conference. Full source; docs included in 1985 FORML Proceedings.

★★ IBM, 114Kb

**A Simple Inference Engine**, Martin Tracy **C005 - \$8**

Based on inference engine in Winston & Horn's book on LISP, takes you from pattern variables to complete unification algorithm, with running commentary on Forth philosophy & style. Incl. source.

★★ IBM, 162 Kb

**The Math Box**, Nathaniel Grossman **C006 - \$10**

Routines by foremost math author in Forth. Extended double-precision arithmetic, complete 32-bit fixed-point math & auto-ranging text. Incl. graphics. Utilities for rapid polynomial evaluation, continued fractions & Monte Carlo factorization. Incl. source & docs.

★★ IBM, 118 Kb

**AstroForth & AstroOKO Demos**, I.R. Agumirsian **C007 - \$6**

AstroForth is the 83-Standard Russian version of Forth. Incl. window interface, full-screen editor, dynamic assembler & a great demo. AstroOKO, an astronavigation system in AstroForth, calculates sky position of several objects from different earth positions. Demos only.

★ IBM, 700 Kb

**Forth List Handler**, Martin Tracy **C008 - \$8**

List primitives extend Forth to provide a flexible, high-speed environment for AI. Incl. ELISA and Winston & Horn's micro-LISP as examples. Incl. source & docs.

★★ IBM, 170 Kb

**8051 Embedded Forth**, William Payne **C050 - \$20**

8051 ROMmable Forth operating system. 8086-to-8051 target compiler. Incl. source. Docs are in the book *Embedded Controller Forth for the 8051 Family*. Included with item #216

★★★ IBM HD, 4.3 Mb

**68HC11 Collection** **C060 - \$16**

Collection of Forths, tools and floating-point routines for the 68HC11 controller.

★★★ IBM HD, 2.5 Mb

**F83 V2.01**, Mike Perry & Henry Laxen **C100 - \$20**

The newest version, ported to a variety of machines. Editor, assembler, decompiler, metacompiler. Source and shadow screens. Manual available separately (items 217 & 235). Base for other F83 applications.

★ IBM, 83, 490 Kb

**F-PC V3.6 & TCOM 2.5**, Tom Zimmer **C200 - \$30**

A full Forth system with pull-down menus, sequential files, editor, forward assembler, metacompiler, floating point. Complete source and help files. Manual for V3.5 available separately (items 350 & 351). Base for other F-PC applications.

★ IBM HD, 83, 3.5Mb

**F-PC TEACH V3.5**, Lessons 0-7 Jack Brown **C201 - \$8**

Forth classroom on disk. First seven lessons on learning Forth, from Jack Brown of B.C. Institute of Technology.

★ IBM HD, F-PC, 790 Kb

**VP-Planner Float for F-PC, V1.01**, Jack Brown **C202 - \$8**

Software floating-point engine behind the VP-Planner spreadsheet. 80-bit (temporary-real) routines with transcendental functions, number I/O support, vectors to support numeric co-processor overlay & user NAN checking.

★★ IBM, F-PC, 350 Kb

**F-PC Graphics V4.6**, Mark Smiley **C203 - \$10**

The latest versions of new graphics routines, including CGA, EGA, and VGA support, with numerous improvements over earlier versions created or supported by Mark Smiley.

★★ IBM HD, F-PC, 605 Kb

**PocketForth V6.4**, Chris Heilman **C300 - \$12**

Smallest complete Forth for the Mac. Access to all Mac functions, events, files, graphics, floating point, macros, create standalone applications and DAs. Based on fig & *Starting Forth*. Incl. source and manual.

★ MAC, 640 Kb, System 7.01 Compatible.

**Kevo V0.9b6**, Antero Taivalsaari **C360 - \$10**

Complete Forth-like object Forth for the Mac. Object-Prototype access to all Mac functions, files, graphics, floating point, macros, create standalone applications. Kernel source included, extensive demo files, manual.

★★★ MAC, 650 Kb, System 7.01 Compatible.

**Yerkes Forth V3.67** **C350 - \$20**

Complete object-oriented Forth for the Mac. Object access to all Mac functions, files, graphics, floating point, macros, create standalone applications. Incl. source, tutorial, assembler & manual.

★★ MAC, 2.4Mb, System 7.1 Compatible.

**Pygmy V1.4**, Frank Sergeant **C500 - \$20**

A lean, fast Forth with full source code. Incl. full-screen editor, assembler and metacompiler. Up to 15 files open at a time.

★★ IBM, 320 Kb

**KForth**, Guy Kelly **C600 - \$20**

A full Forth system with windows, mouse, drawing and modem packages. Incl. source & docs.

★★ IBM, 83, 2.5 Mb

**Mops V2.6**, Michael Hore **C710 - \$20**

Close cousin to Yerkes and Neon. Very fast, compiles subroutine-threaded & native code. Object oriented. Uses F-P co-processor if present. Full access to Mac toolbox & system. Supports System 7 (e.g., AppleEvents). Incl. assembler, manual & source.

★★ MAC, 3 Mb, System 7.1 Compatible

**BBL & Abundance**, Roedy Green **C800 - \$30**

BBL public-domain, 32-bit Forth with extensive support of DOS, meticulously optimized for execution speed. Abundance is a public-domain database language written in BBL. Incl. source & docs.

★★★ IBM HD, 13.8 Mb, hard disk required

## Version-Replacement Policy

Return the old version with the FIG labels and get a new version replacement for 1/2 the current version price.



# A Forth Switchblade

Always searching, never satisfied, I tend to accumulate tools and techniques in Forth. Some are invented, some are ported, and some are outright stolen. One of my current favorites is ported from my assembly language days. It is called a *switch*.

A switch performs the function of a case statement. It is found in most languages and is quite prevalent in C usage. For instance, a very truncated example from a C program which handles Windows messages looks like the code in Listing One. This is the skeleton of a window message handler found in most C applications written for Windows 95 or NT.

A typical example of a switch in Forth is the CASE statement. The ANS-Forth-suggested case statement implementation of the above code C might look like Listing Two. The execution-time behavior of CASE and OF can be optimized until your system implementor is exhausted and performance will be similar to that of the C version.

Given this, why would anyone want to implement a new switch construct in Forth? For SwiftForth, the reason was the need for extensibility—to be able to define the base structure and to extend it at will. The traditional CASE statement does not lend itself to being extended after it is defined.

## Motivation

While building this Windows Forth, one of the biggest problems was needing to define the callback behaviors which had to respond to Windows messages. This is fine if, when you write the message handler, you know all the messages you want to deal with. However, Forth is an interactive environment and I wanted to be able to extend the message handler at any time, either in source code while compiling the system or from the keyboard while testing.

Also, the message handler needs to be defined early in a Windows program, while the behaviors associated with

the messages may not be defined until much later. To use the CASE statement would require either a mass of DEFERED definitions, another mechanism to allow the OF-branches to be added later, or a convolution of the factoring such that the message handler is at the end of the program, after all the procedures are defined.

The SwiftForth switch structure fulfills all of these requirements.

## Implementation

A switch is made up of a head and a list of clauses, each with an associated numeric key. The head consists of the name by which the list is invoked and the default behavior if no clause is matched; the list of clauses defines what will happen when it is called with each possible key value.

The head of the switch structure is built by : SWITCH and

### Listing One

```
LONG APIENTRY PolyProc(HWND hWnd, UINT wParam, WPARAM wParam, LONG lParam)
{
    switch(wParam)
    {
        case WM_CREATE:
            PolyCreateProc(hWnd);
            break;

        case WM_MOVE:
            PolyRedraw(hWnd);
            break;

        case WM_TIMER:
            PolyDrawBez(hWnd);
            break;

        default:
            return(DefMDIChildProc(hWnd, wParam, lParam));
    }
    return(0);
}
```

### Listing Two

```
: POLYPROC ( hWnd msg wParam lParam -- res )
  LOCALS| lParam wParam msg hWnd |
  msg CASE
    WM_CREATE OF   hWnd PolyCreateProc ENDOF
    WM_MOVE   OF   hWnd PolyRedraw   ENDOF
    WM_TIMER   OF   hWnd PolyDrawBez  ENDOF
    DUP        OF   hWnd msg wParam lParam DefMdiChildProc ENDOF
  ENDCASE ;
```

looks like:

```
| link | defaultxt |
```

The link of this structure points to the last clause associated with the switch. The switch clauses are built by <SWITCH and look like:

```
| link | key | matchxt |
```

where each link points to a previous link and the last link is zero.

A switch is executed by passing a value to it, and the list of clauses is traversed looking for a key that matches the value. If a match is found, the value is discarded and the associated *xt* is executed. If no match is found, the value is left on the stack and the switch's default *xt* is executed. This permits the chaining of switches, implementing a kind of inheritance of behaviors. SWITCHER traverses the list of clauses and executes appropriately:

```
: SWITCHER ( i*x n head -- j*x )
  DUP CELL+ @ >R ( save default xt)
  BEGIN
    LINK@ ?DUP WHILE ( n a)
      2DUP CELL+ @ = IF ( match)
        NIP CELL+ CELL+ @ EXECUTE
        R> DROP EXIT
      THEN
    REPEAT R> EXECUTE ;
```

A simple example might be:

```
: ONE ( -- ) ." One" ;
: TWO ( -- ) ." Two" ;
: THREE ( -- ) ." Three" ;
: MANY ( n -- ) . ." more" ;

' MANY :SWITCH NUMBERS
\ MANY is the default for switch NUMBERS

' NUMBERS >BODY
' ONE 1 <SWITCH
' TWO 2 <SWITCH
' THREE 3 <SWITCH
DROP
```

The list can be extended at any time by repeating the same pattern:

```
: FOUR ." Four" ;
: FIVE ." Five" ;

' NUMBERS >BODY
' FIVE 5 <SWITCH
' FOUR 4 <SWITCH
DROP
```

Named Forth words are not required:

```
' NUMBERS >BODY
```

```
:NONAME ." Six" ; 6 <SWITCH
:NONAME ." Seven" ; 7 <SWITCH
```

DROP

A previously defined switch may be overwritten, since the list is searched from newest entry to oldest:

```
' NUMBERS >BODY
:NONAME ." Uno" ; 1 <SWITCH
DROP
```

### Enhancements

- Obvious enhancements to the switch component include:
- Error checking during list building. In SwiftForth, a flag is left on the stack under the switch's address by [ SWITCH which is used by RUNS and RUN: to make sure that the switch clause is appended to an actual switch.
  - An optimized version of switch execution procedure. SWITCHER is presented here in high-level code for portability; any serious implementation should optimize it in native code.
  - Syntactic sugar—automatic parsing for defined words and :NONAME definitions. The SwiftForth equivalent of the above toy application would be:

```
[ SWITCH NUMBERS MANY ( n -- )
  1 RUNS ONE
  2 RUNS TWO
  3 RUNS THREE
SWITCH]

[ +SWITCH NUMBERS
  5 RUNS FIVE
  4 RUNS FOUR
  6 RUN: ." Six" ;
  7 RUN: ." Seven" ;
  1 RUNS ." Uno" ;
SWITCH]
```

This "sugar" allows very concise and simple extension of existing switch statements without a tremendous textual overhead.

[ SWITCH defines a switch with a default behavior.

[ +SWITCH extends the existing switch.

RUNS builds a switch item with a predefined action. This is most useful where a single action will be used for multiple items or where the action is complex.

RUN: builds a switch item with a :NONAME action. This is useful for simple, single-use actions.

- Housekeeping, which in SwiftForth extends the MARKER concept to allow the truncation of switch structures by the user. This is done in SwiftForth by keeping a list of all switches defined, and extending the behavior of MARKER to include pruning all items defined after a marker is declared.
- More data types for the match response than a simple *xt*. One of my favorites is to implement a switch which returns the address of a string, which makes a very nice string table.

### In Practice

In SwiftForth, all Windows message handling is done via switches. This means the user interface can be built up in

parts and extended at will. For instance,

```
MARKER FOO

: ZOT ( -- )
  HWND Z" Caught you!" Z" SwiftForth" MB_OK
  MessageBox DROP ;

[ +SWITCH MESSAGES
  WM_LBUTTONDOWN RUNS ZOT
SWITCH]
```

extends the main SwiftForth Windows message handler to respond to left mouse button presses with a message box. The behavior can be typed in at the keyboard, tested interac-

tively, and discarded by executing the marker FOO. This means that—without reloading, or patching, or anything magic—I can extend the behavior of my predefined programming environment. With this technique, I can trivially insert debug code and monitor what messages and parameters Windows is sending my application.

### Conclusions

The switch construct has been an absolute boon to my efforts at programming for Windows. With it, I can dynamically define responses to Windows messages, and monitor their effects.

The values on which a switch acts are very similar to messages being passed to objects, and we will see more of this next time.

### Listing Three

```
\ Replace LINK@ and LINK, with your favorite list building words.
\ These are the methods used by SwiftForth.

: LINK@   @REL ;
: LINK,   HERE OVER @REL ,REL SWAP !REL ;

\ -----
\ High level implementation of the switch construct

\ SWITCHER searches the linked list from its head for a match to the
\ value N. If a match is found, discard N and execute the associated
\ matched XT. If no match is found, leave N on the stack and execute
\ the default XT.

: SWITCHER ( i*x n head -- j*x )
  DUP CELL+ @ >R ( save default xt)
  BEGIN
    LINK@ ?DUP WHILE ( n a)
      2DUP CELL+ @ = IF ( match)
        NIP CELL+ CELL+ @ EXECUTE
        R> DROP EXIT
      THEN
    REPEAT R> EXECUTE ;

\ Create a code switch whose default behavior is given by XT. Leave the
\ address of the head of its list on the stack.

: :SWITCH ( xt -- addr )
  CREATE HERE 0 , SWAP , DOES> SWITCHER ;

\ Define a new clause to execute the xt when the key N is matched.

: <SWITCH ( head xt n -- head )
  2 PICK LINK, , , ;

\ -----
\ A little syntactic sugar to make switches with.

\ Define a new switch with its default. Use: [ SWITCH name default .
\ The head of the switch is left on the stack for defining clauses.
```

```

: [ SWITCH ( -- head )
  CREATE HERE 0 , ' , DOES> SWITCHER ;

\ Return the address of the given switch, leaving the head for
\ clauses to append to.

: [+SWITCH ( -- head )
  ' >BODY ;

\ Discard the switch head from the stack. Used after defining clauses.

: SWITCH] ( head -- )
  DROP ;

\ Parse for the xt of a new clause.

: RUNS ( head n -- )
  ' SWAP <SWITCH ;

\ Define a nameless clause for the given key.
\ this may be non-portable use of :NONAME

: RUN: ( head n -- )
  :NONAME [ CHAR] ; PARSE EVALUATE POSTPONE ; ( xt) SWAP <SWITCH ;

```

#### Listing Four

```

\ An example of a simple switch

: ONE ( -- ) ." One" ;
: TWO ( -- ) ." Two" ;
: THREE ( -- ) ." Three" ;

: MANY ( n -- ) ." more" ;

[ SWITCH NUMBERS MANY ( n -- )
  1 RUNS ONE
  2 RUNS TWO
SWITCH]

[ +SWITCH NUMBERS
  3 RUNS THREE
  5 RUN: ." Five" ;
  4 RUN: ." Four" ;
SWITCH]

[ +SWITCH NUMBERS
  1 RUN: ." Uno" ;
SWITCH]

```

# Point and Do

In today's PC environment, the mouse has become an integral part of the system. A pointing device can be very useful to interface the user with the intricacies of a program. To supply this function for Forth, the following definitions are presented. The definitions were written for my "embellished" Pygmy Forth<sup>1</sup> although, with minor modifications, they should be applicable to other Forth dialects.

## Description

To apply "point and do," the computer screen is divided into a number of blocks. When the mouse cursor is moved into a block, up to three functions are available. First, a movement (selection) function is invoked which can be used to highlight a command, to display pointed data, etc. (button clicks not required). Separate functions can be called with a single left or right mouse button click.

A set of functions (menu) is listed in an array. Each menu item is described with five entries: upper-left  $x$  cursor position, upper-left  $y$  cursor position, CFA of movement word, CFA of left-button word, and CFA of right-button word. A relatively unlimited number of menu arrays may be assembled. As will be shown, the menu scan always starts with the current menu variable; therefore, a button action in one menu may set a different menu for the next scan. It should be noted that each menu listing requires an appropriate display screen.

## Operation

Screen 1 describes the menu creation and variable definitions which control the point-and-do function. The  $M^X$  and  $M^Y$  variables are used to determine mouse movement. An (ITEM variable is set by a scan of the current menu array and provides pointers to the selected action words. The (MENU variable contains the address of the current menu array.

The PNT&DO word (Screen 2) fetches the mouse cursor position (pixels) and button action via M@P/S. If a button click has occurred, the CFA of the menu action item is fetched and a loop is entered to wait for the button release. During the wait loop, the CFA will be zeroed if simultaneous button action occurs. On release of the button, the fetched CFA (if non-zero) will be executed. Since the action CFA is determined when the button is first clicked, the mouse may be moved to a new location before release, thus providing a means for modified actions (drag and drop, etc.).

If a button click has not occurred, the last and current cursor position is checked for movement. The movement check may seem unnecessary, but it was included to eliminate unnecessary screen updating (possible flicker) and to minimize CPU usage. Any movement will update the last cursor position and initiate a menu scan. The menu selection

BEGIN loop begins with the lower-right item (largest cursor values) and continues while either mouse cursor  $x$  or  $y$  position is less than the menu item  $x$  or  $y$  data (above or right). On exit, the menu item address is saved (for button action reference) and the movement action CFA word is executed if non-zero.

This rather crude cursor comparison does not provide complete freedom of block location. At first, it may be difficult to comprehend the rules for block definition. The menu scan will stop when the current mouse position is between the  $x,y$  data of the current and prior item in the menu array. Therefore, the item  $x,y$  data defines the upper-left block corner, and the larger  $x,y$  data of the prior item defines the lower-right block corner.

If the blocks are arranged in a column-row format, the menu listing should start with the lower-right block, through the row items from right to left, and continue similarly with the higher row. For block arrangements that are not aligned in a column-row format, the general rule of "bottom-right to upper-left" should be followed, although some experimentation may be necessary. This may seem awkward, but I have found it to be sufficient for applications to date.

It should be noted that "dummy" row-columns may be necessary to unmark adjacent marked blocks or to provide areas of mouse inactivity.

Obviously, the PNT&DO word must be called repeatedly. It can be inserted in the keyboard query word. Since the full keyboard function is not required with the PNT&DO action, I usually define a limited keyboard function and include it with the PNT\$DO in a RUN word which may also include flag sampling for a real-time applications.

## Example

Screens 3, 4, 5, and 6 are excerpts from an application and should help demonstrate these functions. This application displays local weather data in both graphical and numerical formats, with commands to retrieve selected data from the data collection hardware.

Several weather parameters are plotted by day (in hour increments) and by hour (two hours, in minute increments). The display screen consists of a command section (top), two graphical sections (left and right center), and a numerical tabulation (bottom).

When the cursor is on the command section, various commands are highlighted as the cursor is moved. Two commands save the displayed data to disk (D.SAVE or H.SAVE) when the left button is clicked. Other commands allow a number (month, day, hour, or minute) displayed above the command to be incremented or decremented by clicking the left/right button, respectively (H+HR, H-HR, etc.).

Richard W. "Dick" Fergus • Lombard, Illinois  
Rfergus@delphi.com

A Forth user for 14 years, Mr. Fergus is heavily involved in a personal, severe weather warning project ([www.theramp.net/sferics](http://www.theramp.net/sferics)). He appreciates Forth's "interactive control and limited restrictions."

As the cursor is moved on the center sections, the numerical values for the pointed plot time (derived from the x cursor position) will be displayed (HR.PNT or DY.PNT) at the bottom of the screen. Clicking the left button on either data plot will retrieve new data (GET.DY or GET.HR), as indicated by the displayed time/date above the command words. Clicking the right button on the daily plot will retrieve and plot (GET.D>H) the minute data of the pointed hour.

The MARK and UN.MARK words are defined on screen 3. For this application, the MARK word unmarks the previous "mark," and highlights (bright white) six character positions from the x,y data of the selected menu item. This x,y position is also saved for use by UN.MARK, which will return the character attributes to normal white.

Screen 4 builds the menu item array. Since the mouse position is reported in pixels, some calculation is necessary to relate the pixel and character positions. As mentioned previously, the menu array must start with the lower-rightmost position and continue toward the upper-leftmost item. The x,y position of the last item must be 0,0 to assure the search loop will exit properly. Two menu items (lines 6 and 14) are used to unmark the area above and below the command labels. Lines 1, 4, and 15 define areas of "no action."

```
Screen # 1
0 \          POINT & DO
1
2 \ Each menu item consists of upper left x/y position, cursor
3 \ movement action, left button action, and right button action.
4 \          Menu example          ( 10 bytes per item )
5 \ CREATE xxxxxx          ( First item lower right most position )
6 \   x ,   y , ' [movement] , ' [left] , ' [right] , ( Item 1 )
7 \   .. , .. , ' ... , ' ... , ' ... ,
8 \   0 , 0 , ' [ " ] , ' [ " ] , ' [ " ] , ( Item n )
9 \          ( Last item position must be x=0 y=0 )
10 VARIABLE M^X   VARIABLE M^Y   \ Cursor position
11 VARIABLE (MENU VARIABLE (ITEM \ Current menu/item
12 : M^X@ ( --- n ) M^X @ ;      \ Get x cursor position
13 : M^Y@ ( --- n ) M^Y @ ;      \ Get y cursor position
14 : MENU! ( adr --- ) (MENU ! ; \ Set menu
15 : ITEM@ ( --- adr ) (ITEM @ ; \ Get pointed item
```

```
Screen # 2
0 \          MOUSE MENU CONTROL
1 : PNT&DO ( --- )
2   M@P/S ?DUP IF 2* 4+ ITEM@ + @ \ Click?--get routine CFA
3   BEGIN 100 MS M@P/S NIP NIP   \ Get button--drop position
4   DUP 2 > IF 2DROP 0 -1 THEN   \ Cancel if both buttons
5   0= UNTIL                      \ Wait for button release
6   ?DUP IF EXECUTE THEN 2DROP   \ If not null, do it
7   ELSE OVER M^X@ - OVER M^Y@ - \ No click--mouse moved?
8   OR IF M^Y ! M^X !           \ Save cursor position
9   (MENU @ BEGIN                \ Scan current menu
10  DUP @ M^X@ >                  \ Compare x's
11  OVER 2+ @ M^Y@ > OR WHILE    \ Compare y's
12  10 + REPEAT DUP (ITEM !      \ Save match item address
13  4+ @ ?DUP IF EXECUTE THEN    \ Movement action?
14  ELSE 2DROP THEN
15  THEN ;
```

```
Screen # 3
0 \          MARK/UNMARK
1 2VARIABLE (MARK
2
3 : UN.MARK ( --- )
4   M-CUR                          \ Cursor off
5   (MARK 2@ 6 7 ATTR$            \ Normal cursor 6 chars at xy
6   M+CUR ;                       \ Cursor on
7
8 : MARK ( --- )
9   UN.MARK M-CUR                 \ Restore prior "mark"
10  ITEM @ 8 / ITEM @ 2+ @ 16 \ Get current xy and save
11  2DUP (MARK 2! 6 15 ATTR$ \ High white attributes
12  M+CUR ;
13
14
15
```

```

Screen # 4
0 CREATE MENU ( --- )
1 0 , 320 , 0 , 0 , 0 , \ Bottom
2 600 , 4 16 * , 0 , 0 , 0 , \ Right border
3 360 , 4 16 * , ' HR.PNT , ' GET.HR , 0 , \ Hour plot
4 280 , 4 16 * , 0 , 0 , 0 , \ Plot center
5 40 , 4 16 * , ' DY.PNT , ' GET.DY , ' GET.D>H , \ Day
6 0 , 3 16 * , ' UN.MARK , 0 , 0 , \ Divider
7 66 8 * , 2 16 * , ' MARK , ' H.SAVE , 0 , \ Day save
8 61 8 * , 2 16 * , ' MARK , ' H+HR , ' H-HR , \ Hour +-hour
9 55 8 * , 2 16 * , ' MARK , ' H+DAY , ' H-DAY , \ Hour +-day
10 49 8 * , 2 16 * , ' MARK , ' H+MON , ' H-MON , \ Hour +-month
11 23 8 * , 2 16 * , ' MARK , ' D.SAVE , 0 , \ Day save
12 18 8 * , 2 16 * , ' MARK , ' D+DAY , ' D-DAY , \ Day +-day
13 12 8 * , 2 16 * , ' MARK , ' D+MON , ' D-MON , \ Day +-month
14 0 , 16 , ' UN.MARK , 0 , 0 , \ 2nd line
15 0 , 0 , 0 , 0 , 0 , \ Top line

```

```

Screen # 5
0
1 : FORM ( --- ) 7 COLOR!
2 12 2 SETCUR ." Month" 19 2 SETCUR ." Day"
3 24 2 SETCUR ." Save"
4 49 2 SETCUR ." Month" 56 2 SETCUR ." Day"
5 61 2 SETCUR ." Hour" 67 2 SETCUR ." Save"
6 7 COLOR! 2 23 SETCUR ." Temperature"
7 6 COLOR! 20 21 SETCUR ." Pressure"
8 7 COLOR! 22 23 SETCUR ." Rain"
9 5 COLOR! 39 21 SETCUR ." MPH"
10 1 COLOR! 35 23 SETCUR ." Direction"
11 4 COLOR! 55 21 SETCUR ." Red sky"
12 2 COLOR! 55 23 SETCUR ." Green sky"
13 3 COLOR! 71 21 SETCUR ." Max Ion I"
14 3 COLOR! 71 23 SETCUR ." Min Ion I"
15 7 COLOR! ;

```

```

Screen # 6
0
1 : WX.CMDS ( --- )
2 HIGPH \ Clear screen-VGA 640x480
3 MENU MENU! \ Set menu commands
4 FORM ; \ Draw screen
5
6 : RUN ( --- )
7 WX.COMDS \ Initial screen display/menu
8 BEGIN
9 PNT&DO \ Check mouse
10 KEY? 27 = \ Exit on ESC
11 UNTIL ;
12
13
14
15

```

The corresponding screen display is generated with the FORM word of screen 5. Obviously, there must be correlation between the menu *x,y* locations and the text positions on the screen.

This application is initiated with the RUN word (screen 6), which calls WX.CMDS and enters a loop with PNT&DO and KEY? (check for keypress). The routine is exited with an Esc keypress. The WX.CMDS word sets the (MENU variable and draws the initial screen. Although not used in this application, other menus can be incorporated by calling other setup words similar to WX.CMDS with a mouse-button action. The number of additional menus is limited only by the imagination and memory available.

### Summary

Although the point-and-do function has some rough edges, I have found it very useful to provide mouse action for several monitoring and data accumulation programs, with a minimum of program overhead. With a little effort, mouse action can provide program control while providing an informative menu for the user.

1. Richard W. Fergus, "Pygmy Embellishments," *Forth Dimensions* XIX.3 (Sept-Oct 1997). Also available at <http://www.theramp.net/sferics> as the file `pyg_embl.exe` in "Misc. Downloads."

# Number Conversion and Literals

## String-to-number conversion

On page 248 of *Starting Forth*, 2nd edition, there are definitions of **NUMBER?** and **NUMBER** that I take as authoritative. Here is a transcription to Standard Forth. The numeric punctuation characters have been extended to be those of *Forth Programmer's Handbook*.

In Classical Forth, the characters `, - . /` can be used freely in a double number. This lets a social security number be written `123-45-6789`; a telephone number `555-1212` or, with 32-bit cells, `1-714-546-9894`; a date `10/29/98` or `10-20-98`; a time `9:30` or `23:59:59`; an ISBN `0-201-89684-2`; and so on. `+` has been added to that, and zip+four can be written `92626+6162`.

**DPL** gives the length of the last field, or `-1` if there are no

punctuation characters. A number ending with a punctuation character will return 0. This gives the way to tell whether the number is single or double integer.

With an application, it can be used as a partial check for validity. With money, you can let whole dollar (or whatever) amounts be in single-number format and convert.

I have added several lines in lower case. They reject an empty string, a lone minus sign, a lone punctuation character, and two successive punctuation characters.

The code in *Starting Forth* accepts all of these as valid numbers. I consider that shoddy. Others think it's practical and economical.

If your implementation is case-sensitive, you may have to change the lower case to upper case.

```

1 ( Variable for decimal point location. )
2 VARIABLE DPL

4 ( Numeric Punctuation : + , - . / test. )
5 : PUNCTION? ( c -- flag ) DUP [CHAR] := SWAP [CHAR] + - 5 U< OR ;

7 ( Check that string is a number. )
8 : NUMBER? ( str len -- num . flag )
9   -1 DPL !
10  ( Reject empty string. )
11  dup 0= if false exit then
12  OVER C@ [CHAR] - = DUP >R 1 AND /STRING ( R: sign)
13  ( Reject lone minus sign. )
14  dup 0= if r> drop false exit then
15  ( Reject lone punctuation. )
16  dup 1 = if over c@ punction?
17  if r> drop false exit then
18  then
19  0 0 2SWAP ( num . str len)
20  BEGIN >NUMBER DUP WHILE
21  OVER C@ punction?
22  ( Reject successive punctuations. )
23  over dpl @ <> and
24  WHILE 1 /STRING DUP DPL !
25  REPEAT THEN
26  NIP ROT ROT R> IF DNEGATE THEN ( len num . )( R: )
27  ROT 0= ( num . flag)
28 ;

30 : NUMBER ( str len -- num . ) NUMBER? 0= ABORT" ? " ;

```

**Base-coded literals**

For cross-development, the following is a popular convention for binary numbers.

Numbers are prefixed by \$ for hex, # for decimal, @ for octal, and % for binary.

```

: NUMBER ( str len -- num . ) DUP 0= IF FALSE EXIT THEN
  BASE @ >R
  OVER C@ CASE
    [CHAR] $ OF HEX      1 /STRING ENDOF
    [CHAR] # OF DECIMAL  1 /STRING ENDOF
    [CHAR] @ OF 8 BASE ! 1 /STRING ENDOF
    [CHAR] % OF 2 BASE ! 1 /STRING ENDOF
  ENDCASE
  NUMBER?
  R> BASE !
  0= ABORT" ? "
;

```

The only one I ever can remember is \$ for hex. I think the eforth and Open Firmware approach is better. Precede all literals by **B#**, **D#**, **H#**, or **O#**. Then there can't be a conflict with a defined word or wrong base.

My own practice is to use decimal as the default base value and **H#** before each sedecimal number. Decimal is for people; sedecimal is for machines; I'm people.

```

1 ( Compile or interpret a number. )
2 : BUILD-NUMBER ( lohaf hihaf -- lohaf | lohaf hihaf )
3   DPL @ 0< IF DROP THEN
4   STATE @ IF
5     DPL @ 0< NOT IF SWAP POSTPONE LITERAL THEN
6     POSTPONE LITERAL
7   THEN
8 ;

10 ( Define word to build a number in a given base. )
11 : base# ( u "<spaces>newname" -- )
12   CREATE IMMEDIATE ,
13   DOES> @ BASE PUSH ( )( R: base)
14     BL WORD COUNT NUMBER ( num .)
15     BASE POP ( R: )
16   BUILD-NUMBER ( num | num .)
17 ;

19 ( Build binary number. )
20 2 base# B#

22 ( Build decimal number. )
23 10 base# D#

25 ( Build hex number. )
26 16 base# H#

28 ( Build octal number. )
29 8 base# O#

```

The following has been included here to balance **BUILD-NUMBER**. They and **NUMBER?** will be needed with "Simple Object Oriented Programming."

```

31 ( Compile or interpret execution token.          SOOP )
32 : BUILD-WORD          ( xt 1/-1 -- [???) )
33   0< STATE @ AND IF COMPILE,
34   ELSE                EXECUTE
35   THEN
36 ;

```

### PAD-free number display and stack dump

This was started when testing a new system before output formatting was installed. In the last three systems I've worked with, I like this format better than the system's .S format. For a very long time .. has been my favorite debug routine.

```

1 ( Recursion for PAD-free number display. )
2 : (.#)          ( n -- )
3   0 BASE @ UM/MOD ( rem quot) ?DUP IF RECURSE THEN ( rem)
4   DUP 9 > 7 AND + [CHAR] 0 + EMIT          ( )
5 ;

7 ( Display number without using PAD. Non-decimal is unsigned. )
8 : .#          ( n -- )
9   BASE @ 10 = IF
10  DUP 0< IF NEGATE [CHAR] - EMIT THEN
11  THEN
12  (.#) SPACE
13 ;

15 ( Concise stack dump bracketed by parens. )
16 : .X          ( ... -- same )
17   ." ( "
18   DEPTH BEGIN ?DUP WHILE DUP PICK .# 1- REPEAT
19   ." )"
20 ;

22 ( Destructive stack dump. Nothing printed for empty stack. )
23 : ..          ( ... -- none )
24   DEPTH 0> IF .X
25   DEPTH 0 DO DROP LOOP
26   THEN
27 ;

```

(. #) is interesting because it uses recursion. Here it is with the recursion removed.

```

: (.#)          ( n -- )
  -1 SWAP          ( -1 n ... )
  BEGIN 0 BASE @ UM/MOD
    DUP 0=
  UNTIL DROP
  BEGIN DUP 9 > 7 AND + [CHAR] 0 + EMIT
    DUP 0<
  UNTIL DROP      \ ( )
;

```

### Sedecimal output

The base I use is normally decimal. When I want to display in hex, I change the base to 16, print with *any* appropriate output word, and change BASE back to decimal. H before the output word does the base flip-flop. Because my normal base is decimal, I don't have to save the base, change to hex, print, and restore the base.

Thus, -1 H U. will give **FFFFFFF** with 32-bit cells.

H . works like the old time H. does.

H .S or H .X gives the stack dump in hex.

Of course, if the base is hex, I don't need to do this.

```

1 : H ( n -- ) S" HEX " EVALUATE
2           BL WORD COUNT EVALUATE
3           S" DECIMAL " EVALUATE
4 ; IMMEDIATE
6 ( BYE )

```

# ONLY STANDARD DEFINITIONS

This file establishes a wordlist that initially has only Standard definitions. The intent is to give you a bare system you can use to check that your application does employ just the Standard words.

To enter this mode:

```
ONLY STANDARD DEFINITIONS
```

To get out of it:

```
-1 SET-ORDER DEFINITIONS
```

The method is to put definitions into the **STANDARD** wordlist for all Standard words. It does this by setting current to **STANDARD-WORDLIST** when making the definition. Context is set to **FORTH-WORDLIST**.

A name that belongs to a word that is not immediate can usually be defined in **STANDARD** as:

```
: name name ;
```

A name that belongs to a word that is immediate can usually be defined in **STANDARD** as:

```
: name POSTPONE name ; IMMEDIATE
```

Standard words that are not defined in your system will compile, but will display "Undefined." when executed. This lets you test that the application would be compiled if the missing word were present.

The following are Tool Belt words. Eliminate the ones you already have.

```
( Bump value of a stored character. )
: C+! ( n addr -- ) DUP >R C@ + R> C! ;

( str len addr PLACE Store character string as counted string. )
: PLACE 2DUP 2>R CHAR+ SWAP CHARS MOVE 2R> C! ;

( str len addr APPEND Append character string to counted string. )
: APPEND 2DUP 2>R COUNT CHARS + SWAP CHARS MOVE 2R> C+! ;

( Convenient factor for several Tool-Belt Definitions. )
: PARAMETER BL WORD COUNT EVALUATE ;

( Conditionally compile the next word. )
: ?? S" IF " EVALUATE PARAMETER S" THEN " EVALUATE ; IMMEDIATE

( Next Word Across Line Breaks as a Character String )
( Length of string is 0 at end of file. )
: NEXT-WORD ( -- str len )
  BEGIN BL WORD COUNT ( str len)
  DUP ?? EXIT
  REFILL
  WHILE 2DROP
  REPEAT ( str len)
;
```

Wordlist to be initialized to Standard words only.

```
1 WORDLIST CONSTANT STANDARD-WORDLIST
```

```
3 ( Vocabulary for Standard wordlist. )
```

```
4 : STANDARD
```

```
5 GET-ORDER DUP 0= ?? 1 NIP
```

```
6 STANDARD-WORDLIST SWAP
```

```
7 SET-ORDER
```

Wil Baden • Costa Mesa, California  
wilbaden@netcom.com

WIL BADEN, after many years of profane language, has retired to Standard Forth. For a copy of the source for this article, send e-mail requesting Stretching Forth #22: ONLY STANDARD DEFINITIONS.

8 ;

The Standard says **POSTPONE TO** is ambiguous, so we write our own.

```
10 STANDARD-WORDLIST SET-CURRENT          ( STANDARD definitions. )

: VALUE CREATE , DOES> @ ;

: TO
  ' STATE @ IF POSTPONE LITERAL POSTPONE >BODY POSTPONE !
  ELSE          >BODY !
  THEN
; IMMEDIATE
( Counting on ` ' something` to be constant, but allowing the
( body to depend on where code has been loaded at this time. )
```

That will fail for local variables. I feel the Standard should have used **TO** for **VALUE** words and **->** for locals.

The Standard says **S"** may have only one buffer, as well as some other problems. So again we code our own.

```
FORTH-WORDLIST SET-CURRENT          ( FORTH definitions. )

12 CREATE SBUF 80 CHARS ALLOT

14 STANDARD-WORDLIST SET-CURRENT          ( STANDARD definitions. )

: S" [ CHAR] " PARSE
  STATE @ IF POSTPONE SLITERAL
  ELSE          80 MIN >R SBUF R@ CHARS MOVE SBUF R>
  THEN
; IMMEDIATE
```

Non-immediate words that do or may affect the return stack also must be postponed.

```
: >R POSTPONE >R ; IMMEDIATE
: R> POSTPONE R> ; IMMEDIATE
: R@ POSTPONE R@ ; IMMEDIATE
: 2>R POSTPONE 2>R ; IMMEDIATE
: 2R> POSTPONE 2R> ; IMMEDIATE
: 2R@ POSTPONE 2R@ ; IMMEDIATE
: EXIT POSTPONE EXIT ; IMMEDIATE

: LEAVE POSTPONE LEAVE ; IMMEDIATE

( So `ONLY FORTH` will work "normally". )

: FORTH ( -- ) STANDARD ;

: ONLY ( -- ) ONLY STANDARD ;
```

```
FORTH-WORDLIST SET-CURRENT          ( FORTH definitions. )
```

**S T R E T C H I N G   S T A N D A R D   F O R T H   -   #22**

```

16   : ORDINARY-WORD      ( str len -- )
17     S" : "              PAD PLACE
18     2DUP                PAD APPEND
19     S" "                PAD APPEND
20                               PAD APPEND ( )
21     S" ; "              PAD APPEND
22     PAD COUNT EVALUATE
23   ;

25   : IMMEDIATE-WORD    ( str len -- )
26     S" : "              PAD PLACE
27     2DUP                PAD APPEND
28     S" POSTPONE "      PAD APPEND
29                               PAD APPEND ( )
30     S" ; IMMEDIATE "  PAD APPEND
31     PAD COUNT EVALUATE
32   ;

34   : .Undefined ." Undefined. " ;

36   : UNDEFINED-WORD    ( str len -- )
37     S" : "              PAD PLACE
38                               PAD APPEND ( )
39     S" .Undefined "   PAD APPEND
40     S" ; "              PAD APPEND
41     PAD COUNT EVALUATE
42   ;

44 ( Define the words that follow into STANDARD wordlist. )
45 : CLONE-THESE-WORDS    ( --- )
46   STANDARD-WORDLIST SET-CURRENT

48   BEGIN NEXT-WORD      ( str len)
49     2DUP S" \\" COMPARE
50   WHILE 2DUP FORTH-WORDLIST SEARCH-WORDLIST DUP ?? NIP

52     ?DUP 0= IF UNDEFINED-WORD
53     ELSE 0< IF ORDINARY-WORD
54     ELSE IMMEDIATE-WORD
55     THEN THEN

57   REPEAT                2DROP

59   FORTH-WORDLIST SET-CURRENT
60 ;

```

Words are in reverse-alphabetic sequence so **WORDS** will show Standard words in order, except for specially defined words.

```
62 CLONE-THESE-WORDS
```

```

64 ] \ [ THEN] [ IF]
65 [ ELSE] [ COMPILE] [ CHAR] [ ' ]
66 [ XOR WRITE-LINE WRITE-FILE

```

S T R E T C H I N G   S T A N D A R D   F O R T H - #22

67 WORDS	WORDLIST	WORD	WITHIN
68 WHILE	W/O	VARIABLE	VALUE
69 UPDATE	UNUSED	UNTIL	UNLOOP
70 UM/MOD	UM*	U>	U<
71 U.R	U.	TYPE	TUCK
72 TRUE		TIME&DATE	TIB
73 THRU	THROW	THEN	SWAP
74 STATE	SPAN	SPACES	SPACE
75 SOURCE-ID	SOURCE	SM/REM	SLITERAL
76 SIGN	SFLOATS	SFLOAT+	SFALIGNED
77 SFALIGN	SF@	SF!	SET-PRECISION
78 SET-ORDER	SET-CURRENT	SEE	SEARCH-WORDLIST
79 SEARCH	SCR	SAVE-INPUT	SAVE-BUFFERS
80 S>D		RSHIFT	ROT
81 ROLL	RESTORE-INPUT	RESIZE-FILE	RESIZE
82 REPRESENT	REPOSITION-FILE	REPEAT	RENAME-FILE
83 REFILL	RECURSE	READ-LINE	READ-FILE
84		R/W	R/O
85 QUIT	QUERY	PREVIOUS	PRECISION
86 POSTPONE	PICK	PARSE	PAGE
87 PAD	OVER	ORDER	OR
88 OPEN-FILE		OF	NIP
89 NEGATE	MS	MOVE	MOD
90 MIN	MAX	MARKER	M+
91 M* /	M*	LSHIFT	LOOP
92 LOCALS	LOAD	LITERAL	LIST
93	KEY?	KEY	J
94 INVERT	INCLUDED	INCLUDE-FILE	IMMEDIATE
95 IF	I	HOLD	HEX
96 HERE	GET-ORDER	GET-CURRENT	F~
97 FVARIABLE	FTANH	FTAN	FSWAP
98 FSQRT	FSINH	FSINCOS	FSIN
99 FS.	FROUND	FROT	FREE
100 FOVER	FORTH-WORDLIST		FORGET
101 FNEGATE	FMIN	FMAX	FM/MOD
102 FLUSH-FILE	FLUSH	FLOOR	FLOG
103 FLOATS	FLOAT+	FLNP1	FLN
104 FLITERAL	FIND	FILL	FILE-STATUS
105 FILE-SIZE	FILE-POSITION	FEXPM1	FEXP
106 FE.	FDUP	FDROP	FDEPTH
107 FCOSH	FCOS	FCONSTANT	FATANH
108 FATAN2	FATAN	FASINH	FASIN
109 FALSE	FALOG	FALIGNED	FALIGN
110 FACOSH	FACOS	FABS	F@
111 F>D	F<	F0=	F0<
112 F/	F.	F-	F+
113 F**	F*	F!	EXPECT
114 EXIT	EXECUTE	EVALUATE	ERASE
115 ENVIRONMENT?	ENDOF	ENDCASE	EMPTY-BUFFERS
116 EMIT?	EMIT	ELSE	EKEY?
117 EKEY>CHAR	EKEY	EDITOR	DUP
118 DUMP	DU<	DROP	DOES>
119 DO	DNEGATE	DMIN	DMAX
120 DFLOATS	DFLOAT+	DFALIGNED	DFALIGN



## URLs — a selection of Web-based Forth resources

### **The MOPS Page**

<http://www.netaxs.com/~jayfar/mops.html>

The Mops public-domain development system for the Macintosh with OOP capabilities like multiple inheritance and a class library supporting the Macintosh interface.

### **Frank Sergeant's Forth Page**

<http://www.eskimo.com/~pygmy/forth.html>

Pygmy Forth and related files.

### **EE Toolbox: Software Development: FORTH Internet Resources**

<http://www.eg3.com/softd/forth.htm>

"EG3 identifies, summarizes, and organizes the wealth of Internet information available for practical electronic design."

### **The Pocket Forth Repository**

<http://chemlab.pc.maricopa.edu/pocket.html>

A haven for programs written using Chris Heilman's Pocket Forth, a freeware Forth for the Macintosh.

### **AM Research, Inc., The Embedded Control Experts**

<http://www.amresearch.com/>

AM Research has specialized in embedded control systems since 1979, and manufactures single-board computers as well as complete development systems.

### **Forth on the Web**

<http://pisa.rockefeller.edu:8080/FORTH/>

A collection of links to on-line Forth resources.

### **Laboratory Microsystems, Inc.**

<http://www.cerfnet.com/~lmi/>

The commercial site of LMI, with product information.

### **The Forth Source**

<http://theforthsource.com/>

Mountain View Press provides educational software and hardware models of Forth with documentation for students and teachers.

### **The Journal of Forth Application and Research**

<http://www.jfar.org/>

A refereed journal for the Forth community, from the Institute for Applied Forth Research.

### **COMSOL**

<http://www.computer-solutions.co.uk>

Computer Solutions Ltd. supplies Forth and other tools for embedded microprocessor designers and programmers in the U.K. and continental Europe.

### **MicroProcessor Engineering, Ltd.**

<http://www.mpeltd.demon.co.uk/>

MPE specialises in real-time and embedded systems.

### **Forth Interest Group in the United Kingdom**

<http://www.users.zetnet.co.uk/aborigine/forth.htm>

A major on-line resource for Forth in the U.K.

### **The Home of the 4th Compiler**

<http://www.geocities.com/SiliconValley/Bay/2334/index.htm>

A personal site rich in graphics and audio, as well as technical content.

### **Space-Related Applications of Forth**

<http://forth.gsfc.nasa.gov>

A large table presenting space-related applications of Forth microprocessors and of the Forth programming language.

### **FORTH, Inc.**

<http://www.forth.com>

Product descriptions, applications stories, links, announcements, and a history of Forth.

### **Forth Interest Group Home Page**

<http://www.forth.org/fig.html>

Extensive selection of links, files, education, and a members-only section.

### **Forth Information on Taygeta**

<http://www.taygeta.com/forth.html>

A selection of tools, applications, and info about the Forth Scientific Library.

### **Jeff Fox and Ultra Technology Inc.**

<http://www.dnai.com/~jfox/>

Information about Forth processors.

### **Offete Enterprises, Inc.**

<http://www.dnai.com/~jfox/offete.html>

Offete Enterprises has Forths for many systems and documentation about some public-domain systems.

### **Forth Online Resources Quick-Ref Card**

<http://www.complang.tuwien.ac.at/forth/forl.html>

Extensive list of links to Forth enterprises and personalities.

### **The Forth Research Page**

<http://cis.paisley.ac.uk/forth/>

Peter Knaggs' list of Forth resources.

### **Yahoo Page on Forth**

[http://www.yahoo.com/Computers\\_and\\_Internet/Programming\\_Languages/Forth/](http://www.yahoo.com/Computers_and_Internet/Programming_Languages/Forth/)

Some of the search engine's hits on "Forth."

### **The Open Firmware Home Page**

<http://playground.sun.com/pub/1275/>

Information published by the Open Firmware Working Group, provided as a free service.

### **American National Standard Forth Information**

<ftp://ftp.uu.net/vendor/minerva/uathena.htm>

Courtesy of Athena Programming, Inc., working documents are posted here by direction of Technical Committee X3J14, at the discretion of the X3 Secretariat.

## SPONSORS & BENEFACTORS

The following are corporate sponsors and individual benefactors whose generous donations are helping, beyond the basic membership levels, to further the work of *Forth Dimensions* and the Forth Interest Group. For information about participating in this program, please contact the FIG office ([office@forth.org](mailto:office@forth.org)).

### Corporate Sponsors

AM Research, Inc. specializes in Embedded Control applications using the language Forth. Over 75 microcontrollers are supported in three families, 8051, 6811 and 8xC16x with both hardware and software. We supply development packages, do applications and turn-key manufacturing.

Clarity Development, Inc. (<http://www.clarity-dev.com>) provides consulting, project management, systems integration, training, and seminars. We specialize in intranet applications of Object technologies, and also provide project auditing services aimed at venture capitalists who need to protect their investments. Many of our systems have employed compact Forth-like engines to implement run-time logic.

Computer Solutions, Ltd. (COMSOL to its friends) is Europe's premier supplier of embedded microprocessor development tools. Users and developers for 18 years, COMSOL pioneered Forth under operating systems, and developed the groundbreaking chipFORTH hot/target environment. Our consultancy projects range from single chip to one system with 7000 linked processors. [www.computer-solutions.co.uk](http://www.computer-solutions.co.uk).

Digalog Corp. ([www.digalog.com](http://www.digalog.com)) has supplied control and instrumentation hardware and software products, systems, and services for the automotive and aerospace testing industry for over 20 years. The real-time software for these products is Forth based. Digalog has offices in Ventura CA, Detroit MI, Chicago IL, Richmond VA, and Brighton UK.

Forth Engineering has collected Forth experience since 1980. We now concentrate on research and evolution of the Forth principle of programming and provide Holon, a new generation of Forth cross-development systems. Forth Engineering, Meggen/Lucerne, Switzerland - <http://www.holonforth.com>.

FORTH, Inc. has provided high-performance software and services for real-time applications since 1973. Today, companies in banking, aerospace, and embedded systems use our powerful Forth systems for Windows, DOS, Macs, and micro-controllers. Current developments include token-based architectures, (e.g., Open Firmware, Europay's Open Terminal Architecture), advanced cross-compilers, and industrial control systems.

The iTV Corporation is a vertically integrated computer company developing low-cost components and information appliances for the consumer marketplace. iTVc supports the Forth development community. The iTVc processor instruction set is based on Forth primitives, and most development tools, system, and application code are written in Forth.

Keycorp ([www.keycorp.com.au](http://www.keycorp.com.au)) develops innovative hardware and software solutions for electronic transactions and banking systems, and smart cards including GSM Subscriber Identification Modules (SIMs). Keycorp is also a leading developer of multi-application smart card operating systems such as the Forth-based OSSCA and MULTOS.

[www.kernelforth.com](http://www.kernelforth.com)

An interactive programming environment for writing Windows NT and Windows 95 kernel mode device drivers in Forth.

MicroProcessor Engineering supplies development tools and consultancy for real-time programming on PCs and embedded systems. An emphasis on research has led to a range of modern Forth systems including ProForth for Windows, cross-compilers for a wide range of CPUs, and the portable binary system that is the basis of the Europay Open Terminal Architecture. <http://www.mpeltd.demon.co.uk>

[www.theforthsource.com](http://www.theforthsource.com)

Silicon Composers (web site address [www.silcomp.com](http://www.silcomp.com)) sells single-board computers using the 16-bit RXT 2000 and the 32-bit SC32 Forth chips for standalone, PC plug-in, and VME-based operation. Each SBC comes with Forth development software. Our SBCs are designed for use in embedded control, data acquisition, and computation-intense control applications.

T-Recursive Technology specializes in contract development of hardware and software for embedded microprocessor systems. From concept, through hardware design, prototyping, and software implementation, "doing more with less" is our goal. We also develop tools for the embedded marketplace and, on occasion, special-purpose software where "small" and "fast" are crucial.

Tateno Dennou, Inc. was founded in 1989, and is located in Ome-city Tokyo. Our business is consulting, developing, and reselling products by importing from the U.S.A. Our main field is DSP and high-speed digital.

ASO Bldg., 5-955 Baigo, Ome, Tokyo 198-0063 Japan  
+81-428-77-7000 • Fax: +81-428-77-7002  
<http://www.dsp-tdi.com> • E-mail: [sales@dsp-tdi.com](mailto:sales@dsp-tdi.com)

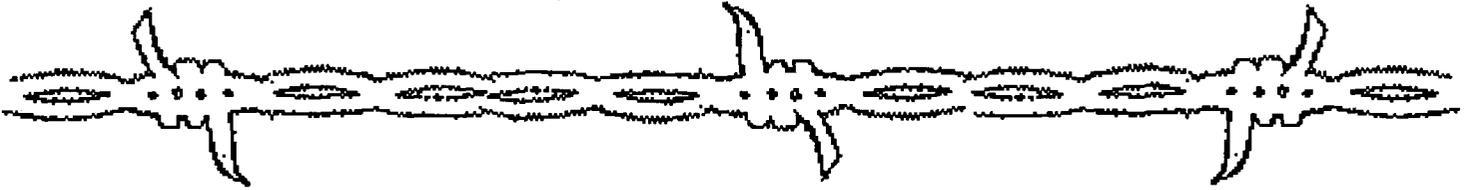
Taygeta Scientific Incorporated specializes in scientific software: data analysis, distributed and parallel software design, and signal processing. TSI also has expertise in embedded systems, TCP/IP protocols and custom applications, WWW and FTP services, and robotics. Taygeta Scientific Incorporated • 1340 Munras Avenue, Suite 314 • Monterey, CA 93940 • 408-641-0645, fax 408-641-0647 • <http://www.taygeta.com>

Triangle Digital Services Ltd.—Manufacturer of Industrial Embedded Forth Computers, we offer solutions to low-power, portable data logging, CAN and control applications. Optimised performance, yet ever-increasing functionality of our 16-bit TDS2020 computer and add-on boards offer versatility. Exceptional hardware and software support to developers make us the choice of the professional.

### Individual Benefactors

Makoto Akaishi  
Everett F. Carter, Jr.  
Edward W. Falat  
Michael Frain  
Guy Grotke  
John D. Hall  
Guy Kelly  
Zvie Liberman

Marty McGowan  
Gary S. Nemeth  
Marlin Ouverson  
John Phillips  
Thomas A. Scally  
Werner Thie  
Richard C. Wagner



# WANTED



BY THE FORTH INTEREST GROUP

## Articles

The author of any Forth-related article published in a periodical or in the proceedings of a non-Forth conference is awarded one year's membership in the Forth Interest Group, subject to these conditions:

- The membership awarded is for the membership year following the one during which the article was published.
- Only one membership per person is awarded in any year, regardless of the number of articles the person published in that year.
- The article's length must be one page or more in the magazine in which it appeared.
- The author must submit the printed article (photocopies are accepted) to the Forth Interest Group, including identification of the magazine and issue in which it appeared, within sixty days of publication. In return, the author will be sent a coupon good for the following year's membership.
- If the original article was published in a language other than English, the article must be accompanied by an English translation or summary.

**"Silicon Slick" (an alias)**

**...and any and all Forth programmers and other SOFTWARE RENEGADES roaming the range in pioneer territories...**

**...to write articles about their DISCOVERIES & TECHNIQUES, PERILOUS MISADVENTURES, and MYSTIFYING ENCOUNTERS with STRANGE CHARACTERS and with FORTH FEATURES obvious and subtle.**

# REWARD

**To recognize and reward authors of Forth-related articles, the Forth Interest Group (FIG) has adopted the following Author Recognition Program.**

The fastest, most convenient way for us to receive your material is via e-mail (a vast improvement over the telegraph, a.k.a "talking wire") to the editor@forth.org address. Binary (e.g., formatted text) files must be uuencoded to be sent as e-mail, but ASCII files can be sent as-is.

## Letters to the Editor

Letters to the editor are, in effect, short articles, and so deserve recognition. The author of a Forth-related letter to an editor published in any magazine except *Forth Dimensions* is awarded \$10 credit toward FIG membership dues, subject to these conditions:

- The credit applies only to membership dues for the membership year following the one in which the letter was published.
- The maximum award in any year to one person will not exceed the full cost of the FIG membership dues for the following year.
- The author must submit to the Forth Interest Group a photocopy of the printed letter, including identification of the magazine and issue in which it appeared, within sixty days of publication. A coupon worth \$10 toward the following year's membership will then be sent to the author.
- If the original letter was published in a language other than English, the letter must be accompanied by an English translation or summary.

