$10 Volume XIX, Number 5 January 1998 February

FORTH DIMENSIONS

Forth Profiling Utility

Easy Target Compilation

Adventures in Debugging a Mix of New Hardware and Software

Manipulating Input Source Contexts in ANS Forth

The beginning of a new year is a great time to set forth (okay, somebody had to say it!) resolutions and goals you plan to live up to in that year. Also, someone once said to me that when you do have a set of goals, they're easier to achieve than when you don't. Here are the resolutions and goals of the Forth Interest Group's Business and Administration office.

1998 Resolutions and Goals for the FIG Business and Administration Office

1. Process membership renewals and new member requests within a week of receiving them. As we hit the road running last year and tried to learn everything as we went along, this was not always possible. However, in 1998, the new office is now better organized—it isn't so new anymore and the systems in place are helping us to handle renewals and new member requests faster and more efficiently. Now that doesn't mean that occasionally one won't slip through the cracks, but we've definitely got a better handle on it this year.

2. Increase our cash flow. To do this, we need your help! (You see, that's the other thing about publishing our goals: if we're lucky, we might get your help along the way.) There are several ways we can increase cash flow; we need:

a. Members (both standard and benefactor status). The Forth Interest Group survives on your generous membership dues and contribution. Your membership and contribution goes to finance the editing, printing, and distribution of *Forth Dimensions*, in addition to helping pay the overhead of the business and administrative office.

b. Corporate/company members. We are proud of this new category of membership. It means that businesses making their living using Forth are willing to make known their support of the Forth Interest Group. We'd like to see more—we know there are more of you out there. Please help to support us if you can.

c. Additional advertising in *Forth Dimensions*. If we have more corporate advertising, it will help to defray the costs of bringing this wonderful source of Forth information to you. A magazine should be able to finance itself through its advertising. We hope to see more advertising this coming year.

d. Sales of products from our mail-order catalog. When was the last time you took a good look at the mail-order catalog? Is there a back volume you need? Have you considered that they may not always be available? We're very low on some years, and also of the *FORML Proceedings*. Each year we print a small number—once those are gone, we don't reprint them. If there is something you've been wanting, you might want to get it now, because another goal of mine for 1998 is:

e. Reduction of back inventory. As we try to run with a leaner overhead, reducing our need for storage of back inventory will help. The word for 1998, if you see it in the Mail Order Catalog and you've always wanted it, get it now!

3. Semi-weekly postings to comp.lang.forth. Nothing big, just general information about memberships, office business, and anything that happens a little out of the ordinary. Tidbits that will keep the Forth Interest Group mentioned in places where we might attract new members. (See 2a and 2b!)

4. Volunteer job list. This idea came up last year: often there are members with time and ambition, and we want to be able to utilize that time for the best of all. With a volunteer job list, we may be able to do that. Keep watching the FIG web site for this.

5. Increased participation in FORML. This year, we had three sponsors of FORML: FORTH, Inc.; Taygeta Scientific, Incorporated; and, as an individual benefactor sponsor, John D. and Jae H. Hall. Thank you to each for your increased financial support. Next year, we'd like to see even more!

This list can go on and on. We have a great organization here, with the foundation to be able to grow and do more things. The reason we have an organization is because of the support and determination of our members and our Board of Directors. The reason we can grow is because of the donations of time and materials by these same people.

1998 is our year to grow and prosper—may it be the same for all of you! Good health, good wealth, and take good care.

Cheers,
Trace Carter
Forth Interest Group
100 Dolores Street, Suite 183
Carmel, California 93923

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

CONTENTS

DEPARTMENTS

EDITORIAL

Beginner's Mind

I've been engaged in an ongoing quest for knowledge of a particular kind. To obtain it requires me to ask fundamental questions. This seems especially appropriate when trying to understand the barriers to Forth's acceptance generally and for specific projects to which it is eminently well suited. Some reasons have been expressed, good ones, but I don't think I've heard The Reason and maybe there isn't one. Maybe the search for a meta-rationale, or just the continual pressure to deliver goods and services, can blind us to the importance of the small things we are so accustomed to doing (or to doing without) and lead us to greatly underrate their importance to the uninitiated.

A colleague recently explained at length why single-steppers aren't needed in Forth—you know: bottom-up, incremental development of well-tested modules eliminates the need; the challenge of providing them in certain environments; etc. It all made sense and I left with head nodding. But after a couple of weeks focusing on the concerns of people new to Forth, I wasn't so sure any more. And when I read Randy Leberknight's contribution to this issue, I saw that even a sophisticated Forth development team might include such tools in its arsenal.

So I revisited that earlier conversation with a different ear. It's an exercise I recommend to anyone trying to move Forth. The explanations still make sense, but they often aren't as convincing. Like when a potential customer approaches a sales counter and explains what he is looking for; if the clerk says, "No, you don't need that, you really need this," the customer is most likely to reply, "uh-huh" and leave to find a store willing to sell what he wants. We could instead meet the new user's initial expectations and let him find that, as one's aptitude increases, reliance on the add-on decreases. It seems to me that the alternative is to require them to commit to a whole new way of programming before they wet their feet in Forth waters (or, at least, before they become willing to pay in money or time to do so), even though most people only appreciate the Forth approach *after* some time spent using it and being exposed to good examples.

This is a micro-topic in a macro-discussion but, not having found The Reason, we should focus on reasons and solve them. Skip Carter's FORML paper, reprinted in our preceding issue, delineates causes of resistance to Forth's use for large-scale projects and I hope the community will address his points. Marcel Hendix's article in this issue may provide a good start. That it is related to these overall concerns is demonstrated by one of his concluding remarks, "The new generation of users may ask for new features and their *must-haves* will be different from present-day requirements."

That might be a tactful way of saying, "Adapt or go extinct." Fortunately, Forth's malleability will allow it to make whatever transition is required, and without losing its fundamentally important characteristics. I have some personal objections about the direction programming in general has taken over the last decade or so, but as Forth perseveres—even if it must do so as a wolf in sheep's clothing—it will be ready when conditions are right and when we address the needs of the wider marketplace as the marketplace perceives its own needs to be.

Errata

Anton Ertl (anton@mips.complang.tuwien.ac.at) reports, "There is a serious bug in my structures package in *Forth Dimensions* XIX.3, pages 13–16. It can be fixed by replacing the definition of CREATE-FIELD with:

```
: create-field ( align1 offset1 align size "name" --  align2 offset2 )
    create swap rot over nalign dup , ( align1 size align offset )
    rot + >r nalign r> ;
```

The author thanks Jack Brien for discovering this bug.

Marlin Ouverson
editor@forth.org

Forth Dimensions
Volume XIX, Number 5
January 1998 February

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

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.

LETTERS TO THE EDITOR

A CASE for avoiding defining words

In regard to Randy Leberknight's article, "Transportable Control Structures" in *Forth Dimensions* XIX.1, I would like to point out an alternative approach for extending CASE. This alternative does not require the use of any compiling words. (This approach surfaced during a rather heated argument at one of the ANS Forth meetings some years ago, in response to a criticism that CASE was too limited.)

The basic idea is to use OF as-is, preceding it with a new word whose output is fed to OF.

For example, suppose you want to test for inclusion within a range of numbers, as with WITHIN; i.e. you would like to be able to write:

```
: FOO  ( selector -- )
   CASE
      3          OF  this       ENDOF
      5 9 RANGE  OF  that       ENDOF
      1          OF  the-other  ENDOF
   ENDCASE
;
```

Recognizing that OF will execute its predicate if and only if the two numbers on top of the stack are the same, the stack diagram for RANGE must be:

```
RANGE  ( selector low high -- selector x )
```

where *x* is the same as *selector* if the selector is within range, and something else otherwise.

Here is one solution:

```
: RANGE  ( selector low high -- selector x )
   2>R  DUP DUP  2R>  WITHIN
   0=  IF  INVERT  THEN
;
```

This technique can, of course, be applied to other kinds of tests in addition to inclusion within a range. It depends only upon the semantics of the particular CASE statement, and not upon its implementation details.

—Mitch Bradley (wmb@FirmWorks.com)

The author replies:

I like it! One part of the definition of elegance is more bang for less buck, and this certainly has that. I am especially attracted to how easy is would be to include this definition any time you felt the need. I might be reluctant to add the case statement with the compiling words if it was only going to be used once; it would feel like I was adding more complication than I was removing. However, this approach is simple enough to justify its addition with just one use.

Nice to hear from you!

—Randy Leberknight (RandyL@phx.mcd.mot.com)

An addendum from the correspondent:

I forgot to point out one other advantage: the non-compiling version doesn't confuse decompilers.

The compiling version is probably marginally faster at run time but, as you point out, small scalar differences in execution speed rarely make much difference in this era of CPUs with five-nanosecond cycle times attached to I/O buses with 400 nanosecond (or more) access times.

—Mitch

The author's final response:

I didn't want to get too crazy with the philosophical stuff, however, since you point out that decompiler issue...

It is, of course, common knowledge that one of the good things about Forth is the ability to extend the compiler. This can become the proverbial combined blessing and curse. The advanced user might consider it a routine exercise to extend the compiler, and think nothing of it. However, it is good to keep in mind that extending the compiler is really a different class of operation (as opposed to just adding a word). Therefore, there is a new class of issues to consider, such as the effect on a decompiler.

As usual, there are tradeoffs. If you demand speed, you can add the compiling word, code its run-time behavior, and extend the decompiler. I doubt that all that is needed in most cases.

Vocabulary vs. wordlist — what's in a name?

In a footnote to his "Working Comments (long)" article in *Forth Dimensions* XIX.1, Julian Noble poses the question:

> *It [VOCABULARY] is now called a WORDLIST in ANS Forth, for reasons that I cannot fathom— what was wrong with VOCABULARY?*

The answer is almost given in the rationale section of the standard, where it says:

> *Search-order specification and control mechanisms vary widely. The FIG-Forth, Forth-79, polyFORTH, and Forth-83 vocabulary and search order mechanisms are all mutually incompatible....*

In particular, many or most pre-ANS Forth systems already had a word named VOCABULARY. Existing VOCABULARYs all addressed the same problem, but were mutually incompatible. I believe that we identified at least five different behaviors of words named VOCABULARY. The committee tried many times to come to an agreement on precise semantics for VOCABULARY, but always ran afoul of the problem that any choice we tried angered 75% of the contingents, rendering their systems and their existing programs non-standard.

As in other areas of similar controversy (e.g., NOT), the committee was able to achieve consensus only by picking a new neutral name. It is my personal opinion that, from a practical standpoint, this is a fine approach, allowing peaceful coexistence between old and new programs. It is relatively easy to add ANS Forth extensions to an existing system if the names don't conflict.

—Mitch Bradley
ANS Forth Committee Member

"I Love the Smell of Ozone in the Morning"

Adventures in Debugging a Mix of New Hardware and Software

Debugging new software or new hardware can be a challenge at any time. However, it is particularly challenging when the software is the firmware for the new hardware. In many cases, the presence of new hardware necessitates new firmware, and we must test them both at once. Some of the features of Open Firmware which help us deal with these challenges are discussed here.

Introduction

As the scene opens, we see a lab bench on which lays the patient—a freshly minted board which is an early version of the Viper. Various cables, reminiscent of I.V. lines, link it to power, a serial port, a keyboard, and a monitor. The flash socket contains the new firmware, which is supposed to work with this new hardware. The part in the socket has a hand-written label with today's date on it, in the barely legible scribble used by the software engineer who just carried the part into the lab. In fact, both the hardware and the firmware are quite new, and we don't actually know yet if either of them work.

Power is applied...

The CPU fan starts up, various people hold their collective breaths...

Don't you wish you were there? I, for one, am sure that most engineers live for that magical moment when a new piece of hardware meets a new piece of software, and the deathly silence coming from the apparently lifeless machine is more than made up for by the voices of the various people pointing fingers around the room...

In reality, I have not heard many finger-pointing arguments here at Motorola Computer Group. However, I have been involved in a number of occasions when we needed to know why something wasn't working, and it wasn't clear if the trouble was due to hardware, software, or both. I would like to describe an occasion on which that occurred, to demonstrate some of the features of Open Firmware which help us to debug a wide variety of problems.

In the scene above, we were not greeted with total dead silence. Instead, the system displayed a cryptic (to the uninitiated) message before it died. "Tried to access instance specific data with no current instance" was the helpful tidbit we were offered before the machine took a left turn into the weeds. At first, we had no idea who was saying it, or why. In an hour or so, we were able to point to a particular bit of a memory device which was stuck, causing the mysterious message. A connector with a bent pin was replaced, and the system worked fine.

In order to explain how we were able to do this, a little background is needed. First, it would be helpful to know some of the responsibilities of the firmware, and how the firmware fulfills them. Then we can talk about some of the debugging capabilities which are built into the firmware, and how we used them to solve this problem.

Initialization of devices

One of the primary responsibilities of the firmware is to initialize various devices such as memory, I/O devices, and bridge chips. For our purposes at this moment, we just need to know that the first part is done by machine-language code, and the second part is done by high-level code.

Tools

The first debugging tool is a flag called `Stand-init-debug?` which can be turned on at compile time. This flag causes in-line assembly of routines which emit a single character out a serial port at strategic times during the system power-up. This is a delicate time in the life of a system, because there are not many resources available for reporting problems. However, if we see a string of characters normally displayed, such as `$#%`, and in the problem case the system stops after #, we know the trouble occurred after some primitive I/O initialization, and before we finished the routine which initializes certain system data structures. This tool is handy for giving a clue about where we are during the early parts of the initialization, but it has two obvious limitations. First, it has to be activated at compile time, so a new ROM must be placed in the system being debugged. Second, it's not very descriptive about where the trouble occurred. The programmer must peruse the source to find out who printed the last character, and who should have printed the next one. Still, since trouble doesn't occur very often at this level, this tool serves its purpose.

The second interesting tool is a progress report like the first, but uses high-level Forth and runs after we have routines in place which can conveniently check `Stand-init-debug?` and print whole strings if it is true. This happens during an initialization process called `stand-init`. It is a chain of routines. The first one just does its job. The second routine calls the previously existing routine and, on returning, does the initialization it wants to do. The third one calls the second (which calls the first...) and when the first two are done, the third does its job. The purpose of this chain is to allow a programmer writing code for a device to write the initialization code with the rest of the driver code, and invoke a compiler routine which automatically hooks the init code into a chain that will happen at power-up.

There is a debugging feature associated with this chain. The programmer can specify a string at the time the code is hooked into the chain. At run time, if `Stand-init-debug?` is true, the string will be printed. Here is what is seen on the

Randy Leberknight • Tempe, Arizona
randyl@phx.mcd.mot.com

For 19 years, Randy Leberknight has been working in areas where hardware and software mix, most recently at Motorola Computer Group working on Open Firmware for PowerPC-based systems.

I/O device when this occurs:

```
Type 'i' to interrupt stand-init sequence
First stand-init:
Calibrate
CIF buffers
memory node
Instruction cache on
Decrementer
Enable machine check exceptions
Set Memory Map
Client memory allocator
MMU
Real mode CIF
Root node
Data cache on
Fast CPU mode
PCI host bridge
CPU nodes
interrupt controller
isa
Power
SuperI/O
SIO Real-Time Clock
Audio chip
Probing memory
Toolbox Flash ROM
```

You can see here 23 steps, during which we could encounter trouble—with software or hardware—and crash, or print an error message. Knowing what init routine was running at the time is a great help in narrowing down the reason for the problem. This information alone is often enough to enable us to find and fix a bug. However, much more can be done. Note the first line, "Type 'i' to interrupt stand-init sequence." By pressing the i key on the serial port before, or just after, this message appears, we can get to the Forth interpreter which is built into Open Firmware. The interpreter gives us a host of options, including running routines one at a time by typing their names, or setting breakpoints and invoking the high-level, single-stepping debugger.

The process

Breakpoints & single-stepping

The first routine to be called after the stand-init chain is called `startup`. `Startup` probes devices for FCode drivers, sets the default console device, prints a banner, looks for a key-chord, and either boots the operating system or invokes a user interface. These are all places where trouble can occur. Therefore, if the system prints the stand-init chain and then stops, we usually "press i to interrupt," set a breakpoint at startup, and tell the system to resume. We can then step through each routine `startup` call, optionally nesting into any high-level routines we encounter.

In the problem case described above, after we installed a ROM with the debug option turned on, we saw all the stand-init chain announcements. We therefore set a breakpoint at `startup`. By stepping through `startup`, we found that the system was issuing the error message during a routine called `probe-rom`.

The decompiler

We used the decompiler to see `probe-rom`:

```
ok see probe-rom
: probe-rom
   " /rom" " probe" execute-device-method drop
;
ok
```

The device tree

This means go to the `/rom` node and execute the method there called `probe`. The scoping mechanisms make it so that the routines which are in the individual device drivers are not normally visible outside the driver. However, we have ways of accessing them. What we do here is:

```
ok dev /rom
ok debug probe
Stepper keys: <space> Down Up Continue Forth
Go Help ? See $tring Quit
ok device-end
ok
```

This means go to the `/rom` node and set a breakpoint on `probe`. Then exit the device-node context. When we enter `resume`, the system will continue booting, but will stop when it gets to the `probe` method we just marked. Upon arriving at the marked code, the debugger was invoked, and announced that it was in `probe`, about to execute `open`. By pressing the space bar, we told it to go ahead and run that routine. Each time we press the space bar, the next routine runs, and the contents of the stack are printed, so we can see what parameters are passed between routines.

Here is a section of what we see when debugging `probe`:

```
: probe       ( ff00b6d0 )
open          ( ff00b6d0 ffffffff )
drop          ( ff00b6d0 )
rom-base      ( ff00b6d0 ff000000 )
/mac-rom      ( ff00b6d0 ff000000 400000 )
bounds        ( ff00b6d0 ff400000 ff000000 )
?do           ( ff00b6d0 )
i             ( ff00b6d0 ff000000 )
fcode?        ( ff00b6d0 0 )
if            ( ff00b6d0 )
1000          ( ff00b6d0 1000 )
+loop         ( ff00b6d0 )
i             ( ff00b6d0 ff001000 )
fcode?        ( ff00b6d0 0 )
if            ( ff00b6d0 )
1000          ( ff00b6d0 1000 )
+loop
```

This told us that we were running a loop which would search through four megabytes of ROM space checking for the presence of FCode tokens at 4K boundaries. We didn't feel like single-stepping though 1024 loops waiting for trouble, so we hooked the serial output to an exterm window with a big buffer, and told the debugger to continue printing, without pausing (the command is c). When we looked at the result, we found that the system found FCode, but the FCode seemed to contain bad commands. The question then became: is the toolbox ROM bad, or are we reading it incorrectly?

Mini-programs—interactively

The next step was to write a couple lines of code at the interpreter which would dump the contents of the ROM to the screen, in the suspect area. We don't even have to leave the debugger to do this. Just enter f, and the interpreter is at our fingertips. Type `resume`, and we are debugging again.

The answer revealed

We saved the dump results to a file, moved the Toolbox to a different system and dumped it from there as well. A file comparison showed that the results were different. Close examination showed that a particular data bit was stuck in one position. When the connector for the Toolbox was examined, bent pins were found. After it was replaced, the board worked fine.

New features

Since this debugging session took place, we have enhanced the debugging features of the system in two ways. First, we can use a non-volatile RAM (`NV-RAM`) configuration variable to tell the system that we want the `Stand-init-debug?` flag turned on, allowing us to interrupt the system without giving it a new flash part. This does not turn on the single-character progress reports which come from the earliest initialization code, so we still need a special PROM to turn those on, because they are assembled conditionally.

Second, there is a sort of dead-man (dead-board?) switch built into the system now. A flag is set in `NV-RAM` before the stand-init chain is called. This flag means, "I died last time." The flag is reset after we successfully negotiate most of the initialization code. Therefore, if we wake up and find the flag set, we know the system hung on a previous boot attempt. This signals us to turn on the debug switch. This is very helpful, because there can be cases where the system is not healthy enough for us to manually turn it on before the system dies.

Conclusion

While this case turned out to be a hardware problem, we have also used these techniques to ferret out software bugs as well. The combination of the progress reports, the interrupt feature, the single stepper, and the Forth interpreter provide a great deal of flexibility when a machine sticks its feet in the air, or when we detect that familiar smell of ozone.

Easy Target Compilation

Custom macro languages are easy to create in Forth. Building on techniques from my last article (*FD* XIX.3, "Approaching CREATE DOES>"), I will demonstrate how simple it is to write custom compilers and assemblers using Forth. Best of all, the new languages will retain the unique interpretive and compiling characteristics of Forth.

A common definition of a macro language is one that is built into a program, such as a spreadsheet or word processor, that allows it to be automated in some way. Microsoft Word and AutoCAD are some well-known applications that contain built-in scripting languages. Forth itself can be linked into a C application and used to interactively call the application functions; PFE for UNIX and Until for DOS/UNIX are two Forth systems written in C that can be dropped into an application or used by themselves for software development.

In this article, I will be describing a macro language in the context of embedded systems, specifically related to host/target communication and development. By this, I mean the ability to use a host computer to interactively develop programs on a target computer that may have a completely different instruction set. The target instruction set will be defined on the host Forth and, when executed by the host, will build the program on the target through some kind of communication link, such as a serial port. The target program can then be initiated by the host.

This trick can be used to provide a scripting interface to an embedded processor or remote computer that has some kind of externally accessible command set. If the target contains at least a Forth inner interpreter and virtual machine, the level of remote control is constrained only by the physical resources of the target. More often, the system will have a debugger or set of operating commands that talks through a serial link.

To demonstrate this, I am going to present an example of a macro language for an embedded device that has a limited command set and some means to load binary programs into memory and execute them. A language for this imaginary device will be defined that compiles commands and data into a buffer that can be transmitted into the device memory for execution. I am using this simple example to expose some of the raw Forth techniques that are used to develop a custom language.

Spawning little languages in Forth

Compiler design is typically considered a very advanced topic in computer science. How then, can it be so easy in Forth? Part of the answer is that the source code for a Forth custom language is executed by Forth; each token in the new language is itself a Forth word and, when executed, it performs the act of compiling. Another reason is that the Forth outer interpreter is already available to parse the language source stream and execute it. The means to interpret the source from text files, or interactively, is part of Forth—you don't have to spend any time developing it. In a couple of dozen lines of code, you can write a special-purpose scripting language that can interpret and execute new programs from ASCII source files.

An assembler compiles mnemonics that represent machine instructions and data into a file that can be executed by a computer; a compiler compiles high-level language source code into machine instructions and data that can also be executed. An assembler or compiler can be written in Forth merely by creating defining words that allow production of words that compile instructions or data into a target memory structure. Forth interprets source text using the word [(named "left-bracket"), which puts Forth in interpretation mode. To write a custom compiler, CREATE DOES> is used to produce target compiling words, and [is used to interpret those words. A custom compiler is simply [used in a definition with some other words that set up the memory structure and other details particular to the target. When the source stream ends, Forth returns to compilation mode through the word] ("right-bracket").

Essentially the new compiler will look like this :

```
: BEGIN-TARGETCOMPILING
  CREATE-TARGET-PROGRAM-AND-BEHAVIOR [ ] ;

: END-TARGETCOMPILING
  CLEANUP-DATA-STRUCTURES ;
```

This may sound a little confusing—Forth entering interpretation mode to perform target compilation and returning to compilation mode when finished. Forth interprets the source code to your new language, whose symbols compile data into a target buffer.] is used in the definition of the target compiler when it is being compiled by Forth.

The confusion can be a path to greater understanding of the language. Forth models the English language so closely that, at times, one will have flashes of insight that can both thrill and disturb. You can reach a new level of Forth knowledge and feel as if you really don't get it at all. It is difficult to write about these abstractions, because a correct statement of a Forth concept can often sound like it is being defined by the concept itself.

Enough Zen. Plod forward and become a compiler writer so you can brag about it at the coffee machine. A couple of good one-liners for this situation are :

"You can write a compiler in three lines of Forth code,"

followed by a thoughtful supping noise at your coffee mug.

Dave Taliaferro • Corrales, New Mexico
dtaliaf@Rt66.com

Dave Taliaferro is a software engineer developing real-time embedded systems. He encountered Forth while working with industrial robotics, and was never able to shake his attraction to this language.

Or,

"A Forth program is a language model of the application problem domain."

Here is a nutshell method for writing a compiler :

1. Define the target memory data structures and location pointers:
   Create a buffer or storage area that will hold the compiled program.
   Create memory access words to adjust location pointers in the target buffer.

2. Define the language:
   Create defining words that allow production of language symbols.
   Create a set of conditional, loop, and branch instructions such as IF, WHILE, BRANCH, and DO ... LOOP.

3. Define the compiler:
   Create a compiler that can execute the language symbol and conditional stream from a text file or other interface.

For simplicity, I am glossing over a couple of finer points regarding dictionaries and literals. To complete a target compiler, a separate dictionary needs to be created to hold target word definitions that may be duplicates of host word definitions.

Figure One

```
: TARGET-ROUTINE
  CREATE , DOES> @ COMPILE>TARGET ;

: CREATE-TARGET-PROGRAM-AND-BEHAVIOR
       CREATE ( associate data with program )
       DOES> ( associate action with program )
;

: T:  CREATE-TARGET-PROGRAM-AND-BEHAVIOR
  INIT-TARGET-MEM [ ] ;

: T; CLEANUP-TARGET-MEM ;
```

Figure One is a bare-bones example of a target compiler in Forth. TARGET-ROUTINE is a defining word used to produce target compiling words. Given a buffer in memory and pointers to access the buffer, COMPILE>TARGET will take the data in a TARGET-ROUTINE word and compile it into the buffer. The compiling takes place when a TARGET-ROUTINE word is executed. Here are some example TARGET-ROUTINES for a make-believe embedded system that has routines in ROM whose addresses begin at the example hex addresses:

```
HEX

A000 TARGET-ROUTINE LOAD-PROGRAM
A01C TARGET-ROUTINE RUN-PROGRAM
A02B TARGET-ROUTINE STOP-PROGRAM
```

In CREATE-TARGET-PROGRAM-AND-BEHAVIOR, we are defining programs in the new language. These programs will themselves be executable words in the host Forth dictionary. The behavior of the "program" depends on the application. For an interactive target compiler linked to an embedded Forth nucleus, we could cause the contents of the target buffer to be transmitted to the target Forth dictionary during target compilation. This would give transparency to program downloading, always a bane when writing embedded code. When the program word is executed on the host, it could transmit a code address to the target Forth which would call the newly embedded routine.

T: and T; implement the compiler. Programs in the new language are written just like Forth:

```
T: MY-PROGRAM
                TARGETBUFFER B600 LOAD-PROGRAM
                RUN-PROGRAM
                BEGIN EKEY? UNTIL
                STOP-PROGRAM
T;
```

With some additional programming, one could write a language whose syntax is like traditional prefix procedural languages.

Notice that, not only are we compiling into the target buffer, we are using host Forth words (BEGIN EKEY? UNTIL) in our new routine. A Forth target compiler can mix host and target words in the same definition. When MY-PROGRAM executes on the host, it will load a program into B600 on the target and execute it, then loop until a key is pressed on the host, causing the target program to terminate. We have moved the user interface from the target to the host, where the full power of the host PC is available for data collection and testing.

Example: A tiny target compiler

I encountered a need for a custom compiler during my last contract, when I discovered that the engineers were hand assembling machine sequence opcodes for a proprietary embedded controller. Basically, the embedded controller would receive a table of hex instructions for port read and write commands, followed by a second table of command offsets in the first table to use for branching and sequencing. This would allow a limited amount of programmability in the controller for test automation. This method was a little weird, and to write a sequence program one had to hand assemble the table, count byte offsets, and manually enter the table data into a program to calculate the CRCs, which was then appended to the data using a text editor. I was able to write a quick target compiler in Forth to automate this with a natural language interface. Of course, if they had used Forth in the embedded controller in the first place...

I was going to use that program for this article, but decided the branching and sequencing scheme was a bit too convoluted to use, so I have written an example macro language that demonstates a couple of the tricks. The example is somewhat useless, since it merely assembles hypothetical opcodes and data into a target memory structure. The opcodes could be machine instructions for a hypothetical processor, or I/O instructions for a specialized controller.

Defining the target memory data structures and location pointers:

The structure of a target buffer and the buffer-access words depends on the application. For this example, a small byte array, TARGETBUFFER, is suitable—the language symbols will compile 16-bit values into the buffer.

Choice of location pointers is also application dependent; there are a number of things you may want to do with the compiled data—such as transmitting it to the target or making a ROM image—that would require additional pointers. In the example, to keep track of the "location counter" for the next free buffer element that will be compiled into, a pointer called THERE is defined (meaning "target HERE"). This is initialized to the value of TARGETBUFFER, and is incremented by two bytes (the value of TARGETCELL) each time a compiling word executes.

Memory-access words are not always necessary, but are convenient. To compile a two-byte value into the location pointed to by THERE, the word THERE!++ stores the value and increments the pointer.

I defined a couple of pointers that keep track of the target's memory space, corresponding to the image we are building on the host: TORG and TWHERE. TORG is the beginning address of target code memory that the compiled data will be loaded into. TWHERE is the target's equivalent to THERE, the location counter.

Defining the language:

The example supposes a target controller that has routines in ROM, along with an interpreter to execute those routines if given the address. We will also suppose that we can build a table of these addresses that can be loaded into the target to be interpreted. A defining word to implement the language only needs the target routine code address and THERE!++ to create a language symbol to compile the code address into TARGETBUFFER. The definition for this defining word is:

```
: TARGET-ROUTINE
  CREATE , DOES> @ THERE!++ ;
```

Creation of conditional, loop, and branch instructions may seem a little tricky, but in Forth it is very simple. For a loop instruction, we need to have an address to loop back to when we reach an end-loop symbol. This is the address in the target buffer where we encountered a start-loop symbol. By saving this address when we hit a start-loop symbol, and compiling it into the buffer when we hit an end-loop symbol, forward referencing is accomplished. We are also assuming that a target branch routine exists, which in the example is called UBRANCH. Its behavior is to branch to the address contained in the next memory location.

```
: START-LOOP THERE?
  TARGETBUFFER - TORG @ + START-ADDRESS ! ;

: END-LOOP
  UBRANCH START-ADDRESS @ THERE!++ ;
```

Our IF: symbol works in a similar fashion. A test instruction is also assumed on the target interpreter that can test the value at an address and branch to a forward location if it false. This is called TARGET-IF. Observe the byte dump of the buffer when the macro language program, MYPROGRAM, executes :

```
E000 : C010 1000 B610 B647 0170 C040 E014 0000
E008 : C020 2000 0045 C040 E01C 0000 C030 E000

TARGET START : E000 TWHERE : E01C
```

The forward references have been taken care of. Note the E014 and E01C addresses compiled after C040, the address for TARGET-IF. These addresses are the values of the location counter when the ENDIF: symbol is interpreted by Forth. Note also that the last 16-bit value in the buffer is E000, preceded by C030, the address for UBRANCH, which causes it to loop back to the beginning of the program. If it still seems confusing, study the hForth source for its conditional and branch instructions—most are only one line of code.

Finally, to define the compiler:

A defining word is needed to name a program in the new language and associate some behavior with it. For our example, the behavior is simply to dump the contents of the target buffer.

```
: DEFINE-PROGRAM-NAME
  CREATE THERE? , DOES> @ BYTEDUMP ;
```

Our custom compiler is just a word to start the interpretation of our new language-symbol-compiling words. It first calls DEFINE-PROGRAM-NAME to give a name to our custom language program, and then enters interpretion mode. When the input stream is exhausted, Forth returns to compilation mode.

```
: BEGIN-PROGRAM
  DEFINE-PROGRAM-NAME [  ] ;
```

To end the process, we may need a word to clean up the target memory buffer on the host, transmit the new definition to the target, or some other housekeeping. For our example, no action is necessary, so we define a dummy word for appearances:

```
: END-PROGRAM ;
```

Well, not bad: a custom language in about two pages of code, including an example program in the new language. Forth experts will notice that some deeper issues have been omitted, but these require little additional code. I will touch upon them in the next article.

Links

Here are some links to more detailed treatments of meta-compilation and target compilation. "Meta" and "target" are used interchangeably by different authors; most imply that it is a matter of personal taste.

Forth assemblers are examples of target compilation. *Starting Forth,* by Leo Brodie, contains an example Forth assembler, and shareware Forths such as Win32Forth and hForth come with assembler sources that are useful for study. Leo also discusses custom languages in *Thinking Forth;* both his books are available from the Forth Interest Group (FIG).

Jeff Fox has a Forth metacompilation tutorial on his web site (http://www.dnai.com/~jfox/meta.html). I think it was Jeff's eForth source code (about five pages) for an MuP21 metacompiler that caused me to embark on writing my own target compiler.

Brad Rodriguez wrote a three-part series for *Forth Dimensions* (volume XIV) titled, "Principles of Metacompilation." Reprints can be ordered from FIG.

The registered version of Pygmy by Frank Sergeant contains source code and tutorial for a metacompiler. Pygmy costs only $15, and would make an excellent platform for a DOS-based remote target compiler (http://www.eskimo.com/~pygmy/forth.html).

I wrote a poor man's Motorola 56002 DSP target compiler using the techniques described here. My system used the Motorola macro assembler to create the target nucleus and for interactive development of assembly object words. A real Forth target compiler would include a Forth target assembler that could be used to generate the embeddable target nucleus, as well as a complete debugging and ROMing toolset. If you are interested in the source code, send me an e-mail request.

Forth, Inc. (http://www.forth.com) and MPE, Ltd. (http://www.mpeltd.demon.co.uk/) sell commercial-quality target compilers for embedded firmware development.

Towards a target compiler for an embedded DSP

My next article will describe the 56002 target compiler. To get it working, I had to solve a number of interesting problems that are common to remote target development systems. Because Forth makes it easy, solving these issues and being able to devise other tools is within the reach of any programmer.

In other words, if I can do it, you can do it.

```
\ typing ".." will reinterpret this file
S" FTASK" DROP 1- FIND NIP [ IF] FTASK [ THEN] MARKER FTASK
: .. [ '] FTASK EXECUTE S" TINYTCOM.F" INCLUDED ;

\ ----------------------------------
CLS CR .( A Tiny Target Compiler ) CR
\ ----------------------------------

CR .( 1> define the target memory data structures )

2 CONSTANT TARGETCELL

CREATE TARGETBUFFER 320 ALLOT       \ a buffer to compile into

VARIABLE THERE          \ target here - target buffer location pointer
VARIABLE TWHERE         \ start of target program
VARIABLE TORG           \ start of target code in rom

VARIABLE START-ADDRESS
VARIABLE IF-ADDRESS

VARIABLE TMP

\ ----------------------------------------
CR .(    define the target memory access words )
\ ----------------------------------------

: THERE? THERE @  ;
: THERE! THERE !  ;

\ store stack item at there, increment there 2 cellS
: THERE!++ THERE? ! THERE? TARGETCELL + THERE! TWHERE @ TARGETCELL + TWHERE ! ;


\ ------ utility routines -------------

\ display location pointers on terminal...
: WELL? CR ." TARGET START : " TORG @ . ." TWHERE : " TWHERE @ . 2 SPACES CR ;

\ initialize target buffer
: INIT-T TWHERE ! TARGETBUFFER DUP 320 0 FILL THERE! ;

\ ------------------------------------------------------------------------
CR CR .( 2> create the defining words that allows production of language symbols )
```

```
\ ------------------------------------------------------------------------------

: TARGET-ROUTINE CREATE , DOES> @ THERE!++ ;

\ --------------------------------
CR .(    define the language symbols )
\ --------------------------------

HEX

C010 TARGET-ROUTINE READPORT
C020 TARGET-ROUTINE WRITEPORT
C030 TARGET-ROUTINE UBRANCH
C040 TARGET-ROUTINE TARGET-IF
B600 TARGET-ROUTINE STOP
B610 TARGET-ROUTINE CALCTEMP
B647 TARGET-ROUTINE DISPLAY
B69A TARGET-ROUTINE CALCERROR

\ ---------------------------------------------
CR .(    define conditional, branch, and loop instructions )
\ ---------------------------------------------

\ store an address to branch to from an END-LOOP
: START-LOOP THERE? TARGETBUFFER - TORG @ + START-ADDRESS ! ;

\ compile the START-LOOP address into target memory
: END-LOOP UBRANCH START-ADDRESS @ THERE!++ ;

: IF: TARGET-IF THERE? DUP IF-ADDRESS ! 2 + THERE ! ;

: ENDIF: THERE? TARGETBUFFER - TORG @ +  IF-ADDRESS @ ! ;

\ create a target constant defining word...
\ when the child word is executed it compiles the constant value
\ into target memory space

: TCONSTANT CREATE , DOES> @ THERE!++  ;

\ -------------------------------
CR CR .( 3> define the compiler ) CR
\ -------------------------------
\ this just displays the buffer contents; in a real application one might
\ transmit a code address to the target associated with the created name
\ on the host
: BYTEDUMP CR TORG @ TMP !
           THERE? SWAP - 2 / TARGETBUFFER SWAP
           8 / 0 DO
                TMP @ DUP . ." : " 8 + TMP !
                8 0 DO DUP C@ >R DUP 1 + C@ R> SWAP  \ ...remove swap if not intel
                8 LSHIFT SWAP OR 0 <# # # # # # #> TYPE SPACE 2 + LOOP CR
           LOOP DROP ;

: DEFINE-PROGRAM-NAME CREATE THERE? ,
         DOES> @ BYTEDUMP ;

\ compiler
: BEGIN-PROGRAM DEFINE-PROGRAM-NAME [   ] ;

: END-PROGRAM ;

\ ------------------------------------------------------
```

```
CR .( Compile an example program in the new language ) CR
\ ------------------------------------------------------

E000 DUP TORG ! INIT-T

BEGIN-PROGRAM MYPROGRAM

                1000 TCONSTANT TEMPSENSOR
                2000 TCONSTANT HEATER1
                175  TCONSTANT SETPOINT
                170  TCONSTANT LOWTEMP
                45   TCONSTANT TIMEOUT

                START-LOOP
                   READPORT TEMPSENSOR
                   CALCTEMP
                   DISPLAY

                   LOWTEMP IF: CALCERROR WRITEPORT HEATER1
                           ENDIF:

                   TIMEOUT IF: STOP
                           ENDIF:

                END-LOOP \ compiles jump instruction to addr from start-loop

END-PROGRAM


CR .( Execute the target program : ) CR

MYPROGRAM WELL? CR
```

Forth Profiling Utility

LPROFILER counts the number of times a source code line is executed. Although not measuring the exact run time of a program line, LPROFILER provides a good start when hunting for performance bottlenecks. Once the most promising candidates for optimization are known, the word .TIME" can be used to time the execution performance of individual Forth phrases. A fringe benefit of LPROFILER is that it shows lines of code that are not visited at all: it points out incompletely tested applications.

Because LPROFILER works like a text filter, it is not necessary to edit or modify source code in order to have it profiled.

The code is tested with Gforth and iForth. LPROFILER is not strictly ANS, in that it requires a system variable pointing to the source line currently being interpreted. Text in screen files cannot be profiled.

Use of the profiling utility is demonstrated with several rewrites of a prime number filter program.

Introduction

This is a utility I wrote in 1993 to introduce ANS Forth to readers of *Het Vijgeblad*, the periodical of the Dutch FIG. LPROFILER is inspired by "Column 1: Profilers" from Jon Bentley's book *More Programming Pearls, Confessions of a Coder* (ISBN 0-201-11889-0). Bentley worked at AT&T Bell Laboratories and started his famous columns for *Communications of the Association for Computing Machinery* (CACM) in 1986. In his columns, he discussed the daily practice of high-octane programming. As Bentley himself put it in the introduction to his book:

> *"Computer programming is fun. Sometimes programming is elegant science. It's also building and using new software tools. Programming is about people too: What problem does my customer really want to solve? How can I make it easy for users to communicate with my program? Programming has led me to learn about topics ranging from organic chemistry to Napoleon's campaigns. This book describes all these aspects of programming, and many more."*

The first columns from *MPPCC* describe common programming techniques. One of them is analyzing the dynamic execution behavior of programs. My translation of this process can be defined as: How many times, and under which circumstances, does code described in a certain part of the source program execute?

What is profiling?

When discussing run-time efficiency of programs, it is common to (hesitantly) admit that current non-commercial Forths are not implemented very efficiently. Although Forth used to be ten times faster than BASIC, current BASICs have steadily improved while Forth, as we all know, only gets reinvented. A test I did while researching this article shows that MS-DOS QBASIC 1.0 (1991) is about five times slower than Win32Forth 3.4 (1997) on the *Sieve of Eratosthenes* benchmark. So far, so good. However, when the source is fed to good old BASCOM (1987), the BASIC Sieve suddenly runs seven times faster than Win32Forth. Without changing one single character in the source.

The standard reply to this rather unpopular fact is to state that in Forth it is very easy to find the words responsible for the bulk of the run time. These words can then be rewritten in assembler (converted to CODE words).

The resident Forth assembler is, of course, a big asset. But it is hard to believe that to have a significant effect (like a factor of seven speedup), it will suffice to rewrite *just one* of the kernel words with the assembler. To begin with, these words will be written in assembly language already (and if, for instance, F-PC is a typical case, it will be very smartly written assembly language indeed). What we can hope for is that our source code contains one or more strings of serially connected, simple CODE words. Although each of the component words is written optimally, it will have the full overhead of the inner interpreter—ranging from insignificant in an optimizing subroutine-threaded Forth to gruesome in a token-threaded implementation.

The implicit assumption, that in every larger program only very few statements are responsible for the bulk of the run time, is seldom proved. Jon Bentley gives examples that make it likely for C code. Let's see if it is true for Forth.

Source code overview

This article provides a tool to search for efficiency bottlenecks in existing Forth programs. This is the actual profiler. A second tool is able to measure the execution time of given Forth words: Word *A* executing a million times but taking only a microsecond each time is okay, while word *B* executing 10,000 times and taking 10 milliseconds is a disaster.

Only the source code for the above two utilities is shown. Several different approaches to implementing a prime number sieve show that interesting results can be arrived at. (For in-depth treatment, the interested reader is advised to study Jon Bentley's book.)

The programs were tested with two very different ANS Forth implementations: iForth (subroutine-threaded and optimizing) and Gforth (direct-threaded). The results of the profiling sessions will be markedly different on your own system. The timings given in this article are true for iForth 1.07 running on an Intel Pentium-166 under Windows NT 4.0.

Marcel Hendrix • Eindhoven, The Netherlands
mhx@IAEhv.nl

Marcel is a senior developer for Philips Lighting, designing power supplies for gas-discharge lamps, including the power electronics, magnetic components, lamp physics and modeling, and digital control theory.

Run-time speed of a Forth word

To measure the execution speed of single Forth words, e.g., DROP, I originally came up with the following user interface:

```
.TIME" DROP"   ( 0.12 microseconds/iteration  )
```

In practice, this is not okay. To time DROP, it will have to execute at least once. Because, normally, the parameter stack is empty, the likely result of the test will be a stack underflow error. The only way to prevent the stack error is to give .TIME" knowledge about the stack effect of each tested word and to push or pop sufficient numbers to keep the stack manager happy. This is too complex and extremely inelegant.

A second problem is that executing the test words only once gives very inaccurate results. Most system timers won't have sufficient resolution to time sub-microsecond events.

The final implementation requires that the phrase to be tested has a neutral stack effect. The above example becomes:

```
.TIME" 1 DROP"   ( 0.30 microseconds/iteration )
```

Obviously, the code to prepare and clean up the stack is also being timed, decreasing the accuracy of the measurement. In practice, this is only a problem when testing kernel or CODE words (as demonstrated above).

The resolution problem is solved by making a temporary definition where the tested phrase is put inside a DO LOOP. This temporary definition is executed a variable number of times, until at least a full second of run time has elapsed. After correcting the elapsed time for DO LOOP overhead (I don't do that in practice), we divide by the number of loop iterations.

When the tested phrase does not have the required neutral stack effect, the internal DO LOOP makes sure nasty things will happen.

An interesting side-effect of the chosen solution is that it is valid to test the following phrase:

```
.TIME"  2   0 IF 1+ ELSE 2- THEN  DROP"
```

This is so because the string inside the quotes is compiled before it gets executed.

Let's proceed to the implementation. The temporary definition for the above reads:

```
:NONAME ( -- )
TIMER-RESET
          /checks 0 DO
                2  0 IF 1+ ELSE 2- THEN  DROP
           LOOP
       TIMER-STOP ; ( -- xt ) TO *secret*
```

The execution token for the nameless definition is saved in a secret VALUE. The timer word sets up /checks calls to EXECUTE with the secret *xt*, trying to make sure at least one second goes by:

```
: CHECK-OUT
  10 TO /checks          \ initialize /checks
  BEGIN
    *secret* EXECUTE     \ do phrase
    TIMER-READ 1000 <    \ busy for 1000 msecs?
  WHILE                  \ no, ten times more
    /checks 10 * TO /checks
  REPEAT
  .RESULT ;     \ time in milliseconds..
                \ ..divide by /checks and print
```

The definitions of TIMER-RESET, TIMER-READ, and TIMER-STOP might be a problem for some Forth systems. This can be solved with the ANS Forth word TIME&DATE, which reports times with a one-second granularity. To get sufficient resolution, the running time of the test phrase must be increased to at least 100 seconds. (Fortunately, one of Gforth's developers helped me write a new primitive to make timing practical for this Forth. Waiting 100 seconds for a result is a real burden for someone as impatient as me.)

Finally, the approach to building a temporary definition. I implemented this by splicing together three strings and EVALUATEing the result. String1 is

```
:NONAME TIMER-RESET /checks 0 DO
```

String2 is the phrase to be tested, and string3 is

```
LOOP TIMER-STOP ; TO *secret* CHECK-OUT
```

When the Forth system has string-manipulation words, this is a simple operation, e.g., in F-PC one can use PLACE and PLACE+. I used an approach in which the strings are copied character-by-character to the nowbuf buffer. With nowbuf COUNT EVALUATE we then ask the Forth compiler to build the temporary code. When finished with .TIME", the compiled code is automatically removed by the word FORGET-TEMP.

The profiler

A design requirement for the profiler was that it shouldn't be necessary to manually change already working and tested code. Existing and well-known tricks to extend : with code to first increment a counter are not suitable, because it does not allow one to profile each line of a definition—unless it is a one-liner.

In the end, I chose the solution in which each line of the source file is read in, prefixed with a special immediate Forth word, and is copied out to a temporary disk file. This special Forth word is called ^ and its execution-time action is to do nothing. During compilation, however, ^ compiles code to increment a specific counter in an array of counters. The index of the chosen counter *n* corresponds with the line in the file currently being interpreted that caused ^ to start compiling. In order for this to work, it is necessary that ^ can ask the Forth system to tell it this line number. iForth and Gforth provide such a word. By carefully engineered miracle, both call it SOURCELINE# (-- n).

Using the profiler, therefore, requires preparing a file. This is done with PROFILE <name> . The new file is always called !!!!!!!!.$$$. Immediately after building this file, PROFILE reads it again and compiles it.

PROINIT can be used to reset the counter array. This is not always a useful action, therefore PROFILE doesn't do it automatically.

After normally executing the prepared program, the results can be studied with .PROFILE. This word prints the source file text, indicating the execution count of a line in the left margin. [See Figure One.]

The word ?FILE is a standard ANS ior return value handler. A possible implementation is:

```
: ?FILE ABORT" File error" ;
```

WAIT? tests for keypresses. If yes, the key is read. When it is the Esc key, WAIT? returns a true flag; if not, it waits for the

Figure One.

```
        | -- P1 ------------------------------------------------------------
 109890 + : prime          local n  TRUE                    \ <n> -- <bool>
 109890 +                  n 2 ?do
8582420 +                           n i mod
8582420 +                           0= if
  91410 +                                 invert leave
        |                            then
8491010 +                     loop ;
```

Figure Two.

```
        | -- P1 ------------------------------------------------------------
 109890 + : prime          local n  TRUE                    \ <n> -- <bool>
 109890 +                  n 2 ?do
8582420 +                           n i mod
8582420 +                           0= if
  91410 +                                 invert leave
        |                            then
8491010 +                     loop ;
        |
        |
    110 + : P1             /n 1+ 2 ?do                     \ <> -- <>
 109890 +                             i prime if
  18480 +                                       i .result
  18480 +                             endif
 109890 +                       loop ;
```

Figure Three

```
        | -- P2 ------------------------------------------------------------
 109890 + : prim2          local n  TRUE                    \ <n> -- <bool>
 109890 +                  2  n iroot > if exit then
 109670 +                  n 2 ?do
 599940 +                           n i mod
 599940 +                           0= if
  91410 +                                 invert leave
        |                            then
 508530 +                           i n iroot > if
  18260 +                                          leave
        |                                    then
 490270 +                     loop ;
        |
        |
    110 + : P2             /n 1+ 2 ?do                     \ <> -- <>
 109890 +                             i prim2 if
  18480 +                                       i .result
  18480 +                             endif
 109890 +                       loop ;
```

next keypress and tests for Esc again. This obscure textual description simply means that pressing Esc stops the listing action of `.PROFILE` immediately. Any other key, presumably the space bar, starts and stops the listing temporarily.

The following line:

```
1 [  SOURCELINE# ] LITERAL probes+!
```

compiles the counter code. `'probes` is a 2000-cell array with line counters. Larger files than this should not be very common in regular Forth practice.

`PROFILE` expects to find the full name of the file to be profiled in the input stream. The name is fetched with `BL WORD`. After creating `!!!!!!!!.$$$`, we call `EDIT-FILE`, close both files, and read `!!!!!!!!.$$$` for interpretation. The call to `EDIT-FILE` is embedded in a `CATCH THROW` construct to take care that, after an internal error, the two files are closed correctly.

`EDIT-FILE` reads a line from the file with the source code to be profiled into a buffer with the fixed prefix text `" ^ "`. This complete buffer is copied to `!!!!!!!!.$$$`. When the new line gets interpreted (through `INCLUDED` in `PROFILE`), the `IMMEDIATE` word `^` will, when we're compiling, generate

Figure Four.

```
        | -- P3 ------------------------------------------------------------
1108890 + : prim3          local n  TRUE                    \ <n> -- <bool>
1108890 +                  n iroot 1+
1108890 +                    2 ?do
5869680 +                           n i mod
5869680 +                           0= if
 922410 +                                  invert leave
        |                            then
4947270 +                     loop ;
        |
        |
   1110 + : P3             /n 1+ 2 ?do                     \ <> -- <>
1108890 +                             i prim3 if
 186480 +                                        i .result
 186480 +                                  endif
1108890 +                        loop ;
```

code to increment the correct counter in `'probes`.

`.PROFILE` reads the still-existing file `!!!!!!!!.$$$` line by line, and removes the `" ^ "` strings. Instead of the prefix, we print the contents of the `'probes` array that corresponds to the current line (that's why there is a line# index in `(.PROFILE)`). We print as much of the source text as will fit on the remainder of the line (`C/L` gives the number of columns the terminal can handle). Again we use a `CATCH THROW` to close the file correctly, in case of an I/O error.

Primes, an example

All of Jon Bentley's profiler examples concern finding prime numbers less than a predefined value. The first naive implementation is the word *P1* (Jon obviously is a C programmer). Shown together with the profiler output, we get the results in Figure Two.

P1 tests all odd numbers less than or equal to *n* for primeness. The word that does the actual testing is called *prime*. This word simply computes *n* modulus all prime candidates. By definition, this modulus will never be zero if *n* is prime. (A prime number can only be divided by itself or by one. Note that two is prime.)

Division, and thus also `MOD`, is a very slow operation. Using `.TIME" P1"` I found a run time of 55 milliseconds with *n* = 1000. We see that `MOD` is called a whopping 8,582,420 times (`/checks` seems to be 110).

The first improvement, *P2*, tests only up to a maximum of $\sqrt{n}$, because a possible divider of *n* will never be any greater [see Figure Three].

Indeed, the number of `MOD` calls falls to a mere 599,940 (fourteen times less). Unfortunately, we also need 109890+508530 calls to *iroot*, which is a floating-point word. In Jon Bentley's example, the root extraction proved to be abysmally slow, and *P2* was much slower than *P1*. Using iForth (hardware floating-point), I found a run time of 10.6 milliseconds per iteration: *P2* is five times faster than *P1*.

In the next improvement, the *iroot* call is moved outside the loop in *prim3* [see Figure Four].

The result is, again, quite a lot faster: `.TIME" P3"` reports 5.5 milliseconds per iteration.

We can do even better by pre-testing for divisibility by 2, 3, or 5 before starting the main loop [see Figure Five].

It is seen that in *prim4* the number of loops goes down dramatically, with nice results: `.TIME" P4"` says 3.24 milliseconds per iteration, which is 70% faster with respect to *P3*. This is an unexpected result (at least to me), which shows that loops should be avoided at all costs. The "sweet spot" in P4 is the fragment `n I mod 0= if` where the "if" part is almost never taken. Rewriting the fragment with the resident assembler is not difficult, and may lead to between three and five times faster execution.[1]

The last program, *P5*, blows our bubble: a new algorithm, *the Sieve of Eratosthenes*, shrinks execution time by a factor of four to a mere 0.945 milliseconds per iteration. Be sure to do your homework before starting to code... [see Figure Six].

Our profiler tells us the phrase `DUP size U<` runs very often. The whole `BEGIN WHILE REPEAT` loop needs attention. It begs to be rewritten as a `DO +LOOP`. Again, we'll leave this as an exercise to the reader.

Concluding remarks

A profiler is a useful tool for the serious Forth programmer. Of course, at the moment Forth is not trying to compete head-on with languages like Pascal or C for system-level programming. However, as ANS Forth now allows us to write such programs, comparisons will inevitably be done. The new generation of users may ask for new features and their *must-haves* will be different from present-day requirements.

I hope to have shown that it is not difficult to build one's own profiling toolkit. You won't need to bug your friendly Forth vendor for it, given system documentation that has at least a certain minimum standard.

Code follows, and can be downloaded by FTP from ftp://ftp.forth.org/pub/Forth/FD/1998/Profiler.zip

1. I could not test this because Gforth for Linux does not have a resident assembler and iForth already generates optimal machine code for the above fragment.

Figure Five.

```
        | -- P4 ------------------------------------------------------------
1108890 + : prim4          local n                          \ <n> -- <bool>
1108890 +                  TRUE  n 2 =  n 3 = or  n 5 = or  if
   3330 +                                                       exit
        |                                                  then
1105560 +                  INVERT ( false )
1105560 +                  n 2 mod 0=  if exit endif
 551670 +                  n 3 mod 0=  if exit endif
 367410 +                  n 5 mod 0=  if exit endif
 294150 +                  INVERT ( true )
 294150 +                  n iroot 1+  7 max
 294150 +                    7 ?do
1698300 +                           n i mod
1698300 +                           0= if
 111000 +                                 invert leave
        |                             then
1587300 +                  2 +loop ;                        \ only test the odd
        |
        |
   1110 + : P4             /n 1+ 2 ?do                      \ <> -- <>
1108890 +                            i prim4 if
 186480 +                                       i .result
 186480 +                                  endif
1108890 +                        loop ;
```

Figure Six

```
        | -- P5 ------------------------------------------------------------
        | -- search primes between 2 and 2 * size  Note: 2 is prime!
        |
  11110 + : P5              /n 2/ local size                 \ <> -- <>
  11110 +                   size 2+ chars allocate ?allocate
  11110 +                   ( addr) local flags
  11110 +                   2 .result                        \ 2 is a prime
  11110 +                   flags size 1 FILL
  11110 +                      size 0 DO
5555000 +                               flags I +
5555000 +                               C@ IF
1855370 +                                     I DUP +  3 +
1855370 +                                     DUP .result
1855370 +                                     DUP  I +
1855370 +                                      BEGIN
9676810 +                                        DUP size <
9676810 +                                      WHILE
7821440 +                                        0 OVER flags + C!
7821440 +                                        OVER +
7821440 +                                      REPEAT
1855370 +                                     2DROP
1855370 +                               ENDIF
5555000 +                         LOOP
  11110 +                   flags free ?allocate ;
```

Listing One. Lprofile.frt

```
\ Some small changes to Gforth to unify the source code...

0 CONSTANT native ( 1 == iForth, 0 == Gforth )

native
  [ IF] ( iForth )

: TIMER-START TIMER-RESET ;                \ <> --- <>
: TIMER-STOP  (.T0) 2DROP ;                \ <> --- <>
: READ-TIMER  diff0 @  ;                   \ <> --- <ms>

[ ELSE]

\ Gforth, ref. Jens Wilke
\ add the next 7 lines to file "primitives", below ms (tabs are significant):
\
\ timeusec    -- nusec nsec new
\ struct timeval tv;
\ struct timezone zone1;
\ gettimeofday(&tv,&zone1);
\ nusec=tv.tv_usec;
\ nsec=tv.tv_sec;

2VARIABLE tstart
VARIABLE diff0
: TIMER-START timeusec tstart 2! ;
: TIMER-STOP  tstart 2@ timeusec rot - -rot swap -
              dup 0< IF 1000000 + swap 1- swap THEN
               ( sec usec )
               1000 / swap 1000 * +  diff0 ! ;
: READ-TIMER  diff0 @ ;                    \ <> --- <ms>

: PRIVATE ;
: DEPRIVE ;
: ?ALLOCATE   THROW ;
: ?FILE       THROW ;
: --          POSTPONE \ ; IMMEDIATE

  9    CONSTANT ^I
CHAR ^ CONSTANT '^'
CHAR " CONSTANT '"'
CHAR . CONSTANT '.'
 27    CONSTANT ESC

: BREAK?      KEY ESC = ;          \ <> --- <bool>   accepts ESC only

: WAIT?              KEY? DUP IF DROP BREAK?     \ <> --- <bool>
                         DUP 0= IF DROP BREAK?
                            ENDIF
                     ENDIF ;

: S>F  ( n -- ) ( F: -- r ) S>D D>F ;
: F>S  ( -- n ) ( F: r -- ) F>D DROP ;
: 2+   2 + ;

FORM CONSTANT C/L  DROP

: NEEDS              POSTPONE \ ; IMMEDIATE
: PRIVATES    POSTPONE \ ; IMMEDIATE
: REVISION    POSTPONE \ ; IMMEDIATE


: SCAN-$      2>R                          \ <addr> <cnt> --- <>
              BEGIN   BEGIN  BL WORD COUNT DUP 0= ( eol or eof..)
                     WHILE  2DROP REFILL  0= IF 2R> 2DROP EXIT
                                          ENDIF
                     REPEAT
                     2R@ COMPARE 0=
              UNTIL
              2R> 2DROP ;
```

```
: (*          S" *)"      SCAN-$ ; IMMEDIATE
: DOC         S" ENDDOC"  SCAN-$ ; IMMEDIATE

[ THEN]

-- Below this line the code is almost standard --------------------------------
(*
 * LANGUAGE     : ANS Forth
 * PROJECT      : Forth Environments
 * DESCRIPTION  : Inspired by Jon Bentley's ``More Programming Pearls''
 * CATEGORY     : Tools
 * AUTHOR       : Marcel Hendrix
 * LAST CHANGE  : May 26, 1997, Marcel Hendrix general butchering for publication
 * LAST CHANGE  : September 8, 1995, Marcel Hendrix removed ARRAY
 * LAST CHANGE  : September 8, 1993, Marcel Hendrix redefined :
 * LAST CHANGE  : March 8, 1993, Marcel Hendrix
 *)

       NEEDS -miscutil

       REVISION -lprofile "fff Forth Line Profiler Version 1.12 fff"

       PRIVATES

DOC Line Profiler
(*
 Profiling?
 ----------
 Sometimes it is useful to know where a program is spending its runtime. Although schemes exist
 where : and ; get redefined to compile counters, there is no direct link to the source code
 with this solution.

 Editing in a special word in the source is very flexible -- it limits output to just the words
 and constructs you're interested in. However, sometimes the exact troublespot is unknown.
 Furthermore, some programmers hate it to have to modify the source code by hand after it is
 finished and debugged.

 The solution presented here is to have the profiler read in the source and write a modified
 version of it to a temporary file. The latter is then included. The modifications made allow
 one to list the original source with an execution count in the left margin.

 Regrettably the idea will not work on all ANS Forth systems. The main stumbling block will be
 the availability of the variable #LINES , counting  the lines compiled.

 There could be problems with the use of TAB's and the IBM-PC character set
 but they should be easy to solve.

 Implementation
 --------------
 The source file is read in line by line. Each line is prepended by the string "^       " (The
 caret character plus a TAB). The word '^' is immediate and does nothing in execute mode. How
 ever, when compiling it compiles code to increment a counter in the array PROBES , at the
 position corresponding to the line where it executes (is: compiles). The modified lines are
 copied to the file  !!!!!!!!.$$$ , which is subsequently included (in that way executing /
 compiling ^ for every line).

 With .PROFILE the file !!!!!!!!.$$$ is read in and displayed without the prepended "^       "
 string. Instead of this string the contents of the corresponding counter in PROBES are displayed.
 This is of course only meaningful when the words in this file have been executed at least once.

 The counters can be reset with the word PROINIT (not automatic!).
*)
ENDDOC

DOC Timing
(*
 Timing individual words
 -----------------------
 Apart from its execution frequency, the speed of execution of a word is important. The words
 TIMER-RESET and .ELAPSED are almost always sufficient for this task, however some (kernel)
 words are so fast that you will appreciate the word  .TIME" string " which times with microsec
 ond resolution. It does this by placing  string  in a loop and executing the result a suffi
 cient number of times (sufficient for the wanted resolution).
```

```
 The words to be tested may not change any stack. This means words must be added to string to
 assure this. Likewise, loop overhead is not automatically subtracted out as the optimizer makes
 this overhead difficult to predict. But you can do this yourself easily. Let's see how you
 would test DUP :

  .TIME" 4  DUP 2DROP"  <cr>  xxx  microseconds / iteration
  .TIME" 4 4    2DROP"  <cr>  yyy  microseconds / iteration

 Subtracting xxx and yyy gives a reasonable approximation to the DUP speed.

 CAREFUL!
 """""""""
 Do not time words from a file that is being profiled. Nothing will break, but the words are
 MUCH slower than without the profiler code, so completely wrong conclusions could be drawn.
*)
ENDDOC


BASE @ DECIMAL

              ( The timing tool )


0 VALUE *secret*   ( can not be private )

CREATE nowbuf PRIVATE         257 CHARS ALLOT

: clear.NB    0 nowbuf C! ; PRIVATE  clear.NB

: c>NOW              nowbuf COUNT + C!           \ <char> --- <>
              nowbuf C@  CHAR+  DUP nowbuf C!
              254 >= ABORT" NOW buffer overflow" ; PRIVATE

: $>NOW              0 ?DO  COUNT c>NOW  LOOP DROP ;    \ <c-addr> <u> --- <>
              PRIVATE


\ Forget the temporary definition, -secret is a MARKER
: FORGET-TEMP S" -secret" EVALUATE ; PRIVATE

0 VALUE /checks      ( cannot be invisible, see :NONAME )

: .RESULT     READ-TIMER 1000      /checks */MOD
              BASE @ >R DECIMAL
                0 .R '.' EMIT
              1000 /checks */ . ." microseconds / iteration."
              R> BASE ! ; PRIVATE


\ The string to be timed MAY NOT HAVE any stack effects.

: CHECK-OUT   10 TO /checks
              BEGIN  *secret* EXECUTE
                     READ-TIMER 1000      U<
              WHILE  /checks 10 * TO /checks
              REPEAT
              .RESULT FORGET-TEMP ;

: .TIME"      clear.NB
              S" MARKER -secret :NONAME  TIMER-START  /checks 0 DO " $>NOW
              '"' WORD COUNT $>NOW
              S"  LOOP TIMER-STOP ; TO *secret*  CHECK-OUT " $>NOW
              nowbuf COUNT EVALUATE ;


              ( The profiler tool )

2000 CONSTANT /maxlines      PRIVATE      \ maximum number of lines in source file
/maxlines CELLS ALLOCATE ?ALLOCATE CONSTANT 'probes

: probes@  ( ix -- n )      CELLS 'probes + @  ; PRIVATE
: probes+! ( n ix -- )      CELLS 'probes + +! ; PRIVATE
: PROINIT  ( -- )    'probes /maxlines CELLS ERASE ;  PROINIT
```

```
native [ IF] :NONAME ( pfa -- )     DROP 'probes FREE DROP ; IS-FORGET probes@  [ THEN]


\ User marker: probe this line if compiling, else do nothing.
\ Note that ':' must be redefined too, but let's delay that ...

: ^           STATE @ 0= IF EXIT ENDIF
                1      POSTPONE LITERAL
              SOURCELINE# 1- DUP /maxlines U> ABORT" array bounds exceeded"
                       POSTPONE LITERAL
              POSTPONE probes+! ; IMMEDIATE

\ The strings to type start with "^      ", which we'll throw away.

: TTYPE              2 /STRING  0 LOCALS| pos |  \ <addr> <u> --- <>
              0 ?DO
                    COUNT DUP  ^I = IF DROP 8  pos 8 MOD  - DUP  SPACES
                                 ELSE EMIT 1
                                ENDIF pos + TO pos
                    pos C/L 14 - U> IF LEAVE
                               ENDIF
              LOOP DROP ; PRIVATE


CREATE "^"buf PRIVATE  '^' C, ^I C, 256 CHARS ALLOT

: EDIT-FILE   LOCALS| hof hif |          \ <infile> <outfile> --- <>
              BEGIN  "^"buf 2+  256  hif READ-LINE  ?FILE
              WHILE  "^"buf SWAP 2+  hof WRITE-LINE ?FILE
              REPEAT DROP ; PRIVATE


\ Instead of INCLUDE name , IN name , S" name" INCLUDED etcetera

: PROFILE     BL WORD COUNT  R/O OPEN-FILE       \ #<filename># --- <>
              ?FILE LOCALS| handle-if |
              S" !!!!!!!!.$$$" W/O CREATE-FILE
              ?FILE LOCALS| handle-of |
                handle-if handle-of
                [ '} EDIT-FILE CATCH IF 2DROP ." oeps!"
                                ENDIF
              handle-of CLOSE-FILE ?FILE
              handle-if CLOSE-FILE ?FILE
              S" !!!!!!!!.$$$" INCLUDED ;


: (.PROFILE)  0 LOCALS| line# handle |    \ <handle> --- <>
              BEGIN  PAD 256 handle READ-LINE ?FILE
                     WAIT? 0= AND
              WHILE  CR line# probes@ DUP 0> IF 9  .R    ."  * "
                                          ELSE 9 SPACES ."  | "
                                               DROP
                                         ENDIF
                     PAD SWAP TTYPE
                     line# 1+ TO line#
              REPEAT DROP ; PRIVATE


: .PROFILE    S" !!!!!!!!.$$$" R/O OPEN-FILE     \ #<name># --- <>
              ?FILE DUP LOCALS| handle |
              [ '] (.PROFILE) CATCH IF DROP ." oeps!"
                                ENDIF
              handle CLOSE-FILE ?FILE ;

: :           :  POSTPONE ^ ; IMMEDIATE

: ABOUT       CR ." A file to be profiled must be loaded with PROFILE name"
       CR ." Execute PROINIT and the main word, then type  .PROFILE  for a listing."
       CR ." Type .TIME" '"' EMIT ."  string "  '"' EMIT ."  to time <string>"
       CR
       CR ." Note that <string> may NOT have any lasting stack effects."
       CR ." Example:  .TIME" '"' EMIT ."  9 iroot drop "  '"' EMIT ."  (t1)"
       CR ."           .TIME" '"' EMIT ."  9 -opt  drop "  '"' EMIT ."  (t2)"
```

```
       CR ."            Real elapsed time = t2-t1" ;

       ABOUT
       DEPRIVE

BASE !
                                (* End of Source *)
```

Listing Two. Ppearls.frt

```
(*
 * LANGUAGE     : ANS Forth
 * PROJECT      : Forth Environments
 * DESCRIPTION : Inspired by Jon Bentley's ``More Programming Pearls''
 * CATEGORY     : Publishable Forth
 * AUTHOR       : Marcel Hendrix
 * LAST CHANGE : March 7, 1993, Marcel Hendrix
 *)

       REVISION -ppearls "fff Publishable Forth   Version 0.01 fff"

-- ANSI Forth programs to print all primes less than /n (1000), in order ----

decimal

1000 value /n                              \ the number of primes to print
   0 value /count                          \ how many primes are there?

defer .result

: .resprint   cr . ;                       \ </primes> --- <>
: .rescount   drop  /count 1+ to /count ; \ </primes> --- <>

              ' .rescount IS .result

: iroot              s>f fsqrt f>s ;                  \ <n> --- <root_n>

-- P1 ------------------------------------------------------------------------
: prime              locals| n | TRUE            \ <n> --- <bool>
              n 2 ?do
                      n i mod
                      0= if
                            invert leave
                       then
                 loop ;


: P1          /n 1+ 2 ?do                  \ <> --- <>
                        i prime if
                                  i .result
                              endif
                    loop ;

-- P2 ------------------------------------------------------------------------
: prim2              locals| n | TRUE            \ <n> --- <bool>
              2  n iroot > if exit then
              n 2 ?do
                      n i mod
                      0= if
                            invert leave
                       then
                      i n iroot > if
                                   leave
                               then
                 loop ;


: P2          /n 1+ 2 ?do                  \ <> --- <>
                        i prim2 if
                                  i .result
                              endif
                    loop ;
```

```
-- P3 ------------------------------------------------------------------------
: prim3              locals| n | TRUE           \ <n> --- <bool>
             n iroot 1+
               2 ?do
                     n i mod
                     0= if
                          invert leave
                      then
                loop ;


: P3         /n 1+ 2 ?do                 \ <> --- <>
                      i prim3 if
                                i .result
                          endif
                    loop ;

-- P4 ------------------------------------------------------------------------
: prim4              locals| n |                \ <n> --- <bool>
             TRUE  n 2 =  n 3 = or  n 5 = or  if
                                              exit
                                           then
             INVERT ( false )
             n 2 mod 0=  if exit endif
             n 3 mod 0=  if exit endif
             n 5 mod 0=  if exit endif
             INVERT ( true )
             n iroot 1+  7 max
               7 ?do
                     n i mod
                     0= if
                          invert leave
                      then
             2 +loop ;                   \ only test the odd ones!


: P4         /n 1+ 2 ?do                 \ <> --- <>
                      i prim4 if
                                i .result
                          endif
                    loop ;

-- P5 ------------------------------------------------------------------------
-- search primes between 2 and 2 * size  Note: 2 is prime!

: P5         /n 2/ locals| size |        \ <> --- <>
             size 2+ chars allocate ?allocate
             ( addr) locals| flags |
             2 .result                   \ 2 is a prime
             flags size 1 FILL
                size 0 DO
                          flags I +
                          C@ IF
                               I DUP +  3 +
                               DUP .result
                               DUP  I +
                                BEGIN
                                  DUP size <
                                WHILE
                                  0 OVER flags + C!
                                  OVER +
                                REPEAT
                               2DROP
                          ENDIF
                    LOOP
             flags free ?allocate ;


: ABOUT      cr ." Type P1 | P2 | P3 | P4 | P5 to print all primes below " /n dec. ;

      ABOUT CR

                           (* End of Source *)
```

Forth Programmer's Handbook

Forth Programmer's Handbook by Edward K. Conklin and Elizabeth D. Rather is the greatest book on Forth to appear in several years. Well, it's the only book on Forth to appear in several years, but it's still great.

It "provides a detailed technical reference for programmers and engineers who are developing software using ANS-compliant versions of Forth provided by FORTH, Inc. or other vendors." It shows how the major supplier of Forth has adapted to Standard Forth, and has adapted Standard Forth.

As a programmer I'm interested in the language, and as a Forth programmer in the words. And so I plunge into Appendix B: Index to Forth Words.

This is supposed to be an alphabetical index to the Forth words appearing in the glossaries in the book. In the first printing the sequence is not quite right, which is a minor nuisance in finding some words.

As the index is obviously based on the Standard words, I look for what's different.

The obsolescent words, the Locals word set, and the basic Search-Order words except for **DEFINITIONS** are not there. All the Search-Order Extension words *are* there, as well as **VOCABULARY**.

COUNT and **DECIMAL** are also missing, which I suspect is an oversight—they are required words. **HEX**, **WITHIN**, and **[COMPILE]** are also missing. **COUNT**, **DECIMAL**, and **HEX** are used in the body of the book, but not in a glossary. I can easily do without **WITHIN** and **[COMPILE]**. And **WITHIN** would be easy to define.

A definition of **COUNT** is given in an example. It has an environmental dependency, using **1+** rather than **CHAR+** in the definition. A later example defines a different **COUNT** as a constant.

In the later example **500 ' COUNT !** is used to change the value of the constant. In another place **[']** without **>BODY** is used to change the value of a 2constant. Thus ticking a constant does not return an execution token.

BYE is also missing. It doesn't make sense in a dedicated environment.

Now for the good stuff—words not in the Standard.

VOCABULARY is one. It has a definition that is agreeable with the result of the definition in the Standard's Rationale. I hope this will effectively standardize the meaning of **VOCABULARY**.

Another "new" word is **NOT**, equivalent to **0=** — the only sensible meaning for today's optimizing Forths.

Other old favorites that have re-appeared are **C+!**, **M-**, **M/**, **T***, and **T/**. **M/** has an ambiguous definition—is it equivalent to **SM/REM NIP** or **FM/MOD NIP** or should dividend and divisor have the same sign for portability?

We can all adopt **[DEFINED]** and **[UNDEFINED]**. (My definitions)

```
: [ DEFINED]    ( <name> -- flag )
  BL WORD FIND NIP 0<> ; IMMEDIATE

: [ UNDEFINED]  ( <name> -- flag )
  BL WORD FIND NIP 0= ; IMMEDIATE
```

It stands to reason that since Forth words can't have blanks in them, Forth source filenames shouldn't have blanks in them, and some systems don't give you a choice. This lets us write **INCLUDE filename** instead of **S" filename" INCLUDED**, just like old times.

2+ and **2-** are back.

CONTEXT and **CURRENT** are given as names of addresses used in manipulating word lists by the Search-Order words. There isn't anything a user can do with them in Standard use. Up to eight word lists may exist at any one time.

.' attempts to identify the definition in which an address occurs. Thus used after an address, it returns the name of the nearest definition before the address and the offset of the address within that definition.

(In the example the execution token of a word is the address of the beginning of the definition. In general this doesn't have to be so.)

The Standard word **SEE** decompiles or disassembles the following word. Given an address, **DASM** decompiles the code there.

Also for debugging, if the compiler encounters an error and aborts, you can go directly to the block (or file) and line at which the error occurred by typing **L**. **LOCATE** will call up the source code for a command. **WHERE** (*a.k.a.* **WH**) followed by a name will give all the places where the name is used.

CVARIABLE is provided for one-byte variables. It is typically available only on embedded systems.

DEFER declares an execution variable. **TO** is used to assign a meaning. This is an extension of the Standard **TO** for value words.

For code routines the formal endings **NEXT**, **END-CODE**, and **INTERRUPT** are specified. Code routines also have the basic control-flow words.

/LOOP is like Standard **+LOOP** but requires the increment to be positive. It will be much faster because the test to continue the loop is so much simpler—a test for less.

Now for environmental dependencies—the politically correct way to adapt the Standard.

Ever since 1983 I have felt the Forth-83 and now Standard **+LOOP** to be a ridiculous monstrosity. Sure, it gets the

Continued on page 29

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

Wil Baden is a professional programmer with an interest in Forth.

Iterated Interpretation

Iterated interpretation is the most useful tool, after Simple Macros, in the Tool Belt. With it, you insert items or phrases into a longer phrase, and interpret or compile the longer phrase for each item or phrase inserted into it.

The syntax is:

```
// <the-beginning> | <the-end> | <item-or-phrase> ... \\
```

Examples

Declare three variables.

```
// VARIABLE | | Larry Moe Curly \\
```

The result is:

```
VARIABLE Larry  VARIABLE Moe  VARIABLE Curly
```

A function to initialize the variables:

```
: Init-Stooges  // FALSE | ! | Larry Moe Curly \\ ;
```

That becomes:

```
: Init-Stooges  FALSE Larry !  FALSE Moe !  FALSE Curly ! ;
```

Define some constants for a calendar program.

```
0 // DUP CONSTANT | 1+ | SUN MON TUE WED THU FRI SAT \\ DROP

0 // 1+ DUP CONSTANT ||                                  ( month#)
   January February March     April   May      June
   July    August   September October November December
\\                                                        DROP
```

Make a table of sines for 0 to 90 degrees.

```
ALIGN HERE // | , |                                      ( addr)

   0  175  349  523  698  872 1045 1219 1392 1564
1736 1908 2079 2250 2419 2588 2756 2924 3090 3256
3420 3584 3746 3907 4067 4226 4384 4540 4695 4848
5000 5150 5299 5446 5592 5736 5878 6018 6157 6293
6428 6561 6691 6820 6947 7071 7193 7314 7431 7547
7660 7771 7880 7986 8090 8192 8290 8387 8480 8572
8660 8746 8829 8910 8988 9063 9135 9205 9272 9336
9397 9455 9511 9563 9613 9659 9703 9744 9781 9816
9848 9877 9903 9925 9945 9962 9976 9986 9994 9998 10000
\\

: SINE ( degs -- 10000*sin ) CELLS OUTSIDE LITERAL + @ ; DROP
```

Spell the digits in a number.

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

Wil Baden is a professional programmer with an interest in Forth. For a copy of the source for this article, send e-mail to requesting Standard Forth Tool Belt #3: Iterated Interpretation.

```
ALIGN HERE  10 CELLS ALLOT                                  ( addr)

DUP // HERE BL STRING | OVER ! CELL+ |
    Zero One Two Three Four Five Six Seven Eight Nine
    \\
DROP

: .UNIT  CELLS OUTSIDE LITERAL + @ COUNT TYPE SPACE  ; DROP

: .UNITS  0 10 UM/MOD ?DUP ?? RECURSE  .UNIT ;
```

It can be used interactively to check the values of an expression you have defined.

Iterated interpretation uses Agenda as the area for a string. Agenda is Agenda-Limit characters long. Material between // and | is placed at the beginning of the area. Material between | and | is moved to the end of the area. Then each following word or phrase up to \\ is moved one at a time to the end of the beginning part, the end part is moved to the end of this, and the area up to there is evaluated. The end part is moved back to the end of the area to make room for the next word or phrase. The length of the end part is kept on the return stack.

Phrases are delimited by ^ before them and ^ at the end.

```
 1 ( Iterated Interpretation )

 3 ( Tool Belt )
 4 : PLACE  ( a1 n1 a2 -- ) 2DUP 2>R  CHAR+ SWAP MOVE  2R> C!;
 5 : STRING ( char "ccc<char>" -- )
 6          WORD COUNT HERE OVER 1+ CHARS ALLOT PLACE
 7 ;
 8 MACRO ?? " IF \ THEN "

10 : NEXT-WORD                  ( -- caddr k )
11     BEGIN   BL WORD COUNT        ( caddr k)
12             DUP 0=
13     WHILE   REFILL
14     WHILE   2DROP
15     REPEAT  THEN
16 ;

18 160 CONSTANT Agenda-Limit
19 CREATE Agenda   Agenda-Limit CHARS ALLOT

21 MACRO Agenda-End      " Agenda Agenda-Limit R@ - CHARS + "
22 MACRO Agenda-Switch   " Agenda COUNT CHARS + "

24 ( // do-before-each | do-after-each | word-or-^phrase^ ...\\ )
25 : //                                 ( ... -- ??? ) -
26     [CHAR] | PARSE Agenda PLACE
27     [CHAR] | PARSE >R Agenda-End R@ MOVE    ( R: k2)
28     CR
29     BEGIN   NEXT-WORD                       ( a k)
30             DUP
31     WHILE   2DUP S" \\" COMPARE
32     WHILE   2DUP S" ^" COMPARE 0=
33                 IF  2DROP [CHAR] ^ PARSE  THEN
34             DUP Agenda C@ + R@ + Agenda-Limit < NOT
35                 ABORT" Agenda-Limit is too small. "
```

STANDARD FORTH TOOL BELT - #3

```
36              TUCK Agenda-Switch SWAP MOVE    ( k)
37              Agenda-Switch OVER CHARS +
38                  Agenda-End SWAP R@ MOVE
39              Agenda COUNT ROT +              ( a k+k1)
40              2DUP CHARS +  R@  SWAP >R  +    ( a k+k1+k2)( R: k2 a2)
41                  EVALUATE                    ( )
42              R> Agenda-End R@ MOVE           ( R: k2)
43      REPEAT  THEN                            ( a k)
44      R> DROP                                 2DROP  ( R: )
45 ; IMMEDIATE
```

In Starting Forth OUTSIDE is suggested as a way to handle one-time tables.

Here is a definition of OUTSIDE that I think will work for all Forths, regardless of where the control-flow stack is or the size of control-flow stack elements or *colon-sys*. Any compiler security is maintained. Take care to define OUTSIDE when the stack is empty.

```
: OUTSIDE [  DEPTH ] LITERAL PICK ; IMMEDIATE
```

Larry, Moe, and Curly work as stooges for local variables. I use one of them in OUTSIDE to guarantee the value of the literal.

```
47 DEPTH Larry !
48 : OUTSIDE  [  DEPTH Larry @ - ] LITERAL PICK ; IMMEDIATE

50 (
51 --
52 Wil Baden    Costa Mesa, California
53 )
```

STRETCHING STANDARD FORTH - #17

Continued from page 26

job done, but in a complicated way like no human thought process. Sometimes I have wondered at the sanity of whoever proposed it. The handbook improves performance by requiring the increment to be evenly divisible into the range of the loop and thus a simple test for equality can be made. (Presumably **1 0 DO 2 +LOOP** will run forever.)

The definition of **BEGIN** is given:

```
: BEGIN    HERE  ;     IMMEDIATE
```

It is explained that "**BEGIN** is simply an **IMMEDIATE** version of **HERE**."

This shows that (1) the data stack is used for the control-flow stack, (2) control-flow stack elements are one cell wide, and (3) code is compiled into data-space.

The following examples from pages 107–108 show that compiler security is not checked.

```
CREATE TENS  1 , 10 , 100 , 1000 , 10000 ,
: 10**   ( n1 n2 -- n)   CELLS  TENS + @ * ;
HERE   2 ,  4 ,  8 ,  16 ,  32 ,  64 ,
: 2**    ( n n -- n)    CELLS  LITERAL + @ * ;
```

The Handbook says that "32-bit versions of Forth use the circular model" for number representation. This means that "less than" could be defined **: < - 0< ;** and 2,000,000,000 is less than -2,000,000,000. **/LOOP** range is limited from anywhere to halfway around the number circle.

Besides **.** anywhere, punctuation characters for numeric input are **,**, **+**, **-**, **/**, and **:**.

As observed above, ticking a constant does not yield an execution token.

There are some minor errors, which are corrected in new printings.

Conclusion

The extensions (and restrictions) are very attractive and, I think, necessary for present-day Forth applications. It presents a Forth that I would be happy to work in. I'm pleased that Stretching Forth articles are compatible with it.

Manipulating Input Source Contexts in ANS Forth

This paper presents a method of manipulating contexts, a technique which may be useful for programmers who have to switch contexts, e.g., when binding together two languages. The particular problem solved in this paper is to change the current input source parameters, having no special construct to do this or to establish a new input source context with the desired parameters.

Briefly, the following expedients were used:

1. **Making an entity executable** (converting data to an executable format), which permits establishing a context around it.
2. **Changing** the parameters of a **context from inside** it when we either cannot create a context with the desired parameters or have to neutralize the effect of some action.
3. **Temporary changes** in a entity **to make some action**; the changes are undone when the action is complete.
4. **Create** an auxiliary context **and export it** outside its original scope, instead of changing the parameters of an existing context.
5. Use of the context **save and restore** operations **to export a context** outside of its original scope.

Motivation

One of the pitfalls of ANS Forth is that there is no standard tool to redirect the input stream to an arbitrary string in memory (which would enable us to process the string using the standard parsing routines). The word `EVALUATE ( addr len -- )` helps with this only if the first word in the string specified by *addr* and *len* is a Forth word that processes the rest of the string.

In this paper, we shall show how this pitfall may be compensated, using a program with a negligible environmental dependency (the first character of the string to which the input stream must be directed has to be in RAM). At the end of the paper, we also mention some interesting properties of input stream manipulations and contexts. The code is given in Listing One.

A sensible example of usage

Now we can define defining words that get the name of the new word from the stack rather than from the input stream. The symbol *$addr* denotes the address of a counted string.

```
: $create    ( $addr -- )
    >r save-input
    r> count source!
        create
    restore-input drop ;

: $const     ( n $addr -- )
    $create , does> @ ;
```

Now we can define a constant, e.g., as

```
5 c" five" $const
```

(provided that `C"` leaves a counted string's address on the stack in interpretation mode). If the name `five` was followed by a space and some other text, the text would be ignored, because `CREATE` consumes only one name from the input stream.

The testing code, along with the output, is given in Listing Two. (The word `? IO ( errcode -- )` reports an error if *errcode* is not 0.)

Caution: This `EVALUATE`-based implementation of `SOURCE!` sets `SOURCE-ID` to -1 (which means that the input source is a string), so `SOURCE!` cannot be used to *restore* the input source specification. Listing Three contains an example of code in which an attempt is made to save the input stream parameters via `>IN @` and `SOURCE`, and to restore them via `SOURCE!` and `>IN !`. Execution of such code leads to an error in the text interpreter, which tries to read the next line using the current `SOURCE-ID` (which is expected to contain a file identifier but is set to -1 by `SOURCE!`).

We need the word `SOURCE-ID!` to restore the input source parameters this way but, so far, there is no standard definition of `SOURCE-ID!`. (As a consequence, a standard program cannot reuse the word `REFILL` to process an arbitrary text file.) So use `SAVE-INPUT` and `RESTORE-INPUT` to save and restore the input source state.

Must `SOURCE!` modify `SOURCE-ID`? This depends on what we want this word to do. If we want it to establish a new source input context, it must. If we want it to modify the existing context, it must not. And it must not if we want to use it in a word that creates a new source input context: the best choice is to have a separate word (e.g., `SOURCE-ID!`) which sets `SOURCE-ID` to any desired value.

How the code works

The word `SOURCE!` performs the following steps:

1. Auxiliary data manipulations (required to get *addr* and *len* above the unpredictable number of elements produced at step 2).
2. Save the current search order on the stack (the number of one-cell values in the search order specification is unpredictable in the general case).
3. Set a search order in which the auxiliary word `|` may be found.
4. Prepare a string located at *addr* to be processed by `EVALUATE` (this string contains the word `|` and is of length 1).
5. Rearrange the parameters on the stack.
6. `EVALUATE` the auxiliary word `|` in the string of length 1

M.L. Gassanenko • St. Petersburg, Russia
mlg@forth.org

M.L. Gassanenko defended his Ph.D. dissertation at the St. Petersburg Institute of Informatics and Automitazation at the Russian Academy of Sciences (SPIIRAS), where he is now a researcher.

Listing One

```
\ SOURCE! ( addr len -- )
\ make (addr,len) the current source; store -1 to SOURCE-ID
\
\ This implementation of SOURCE! modifies memory at addr,
\    and on some ROMed systems this probably will not work.
\ The character at addr is temporarily modified, even if len=0; therefore:
\    changing the character at addr must not crash the system;
\    the character at addr must be in RAM.
\ The variable #TIB gets modified (this is unusual, but
\    the standard does not forbid it).
\ The systen is assumed to store the length of the input buffer
\    to #TIB for any kind of input source.

wordlist constant ___source!___

GET-CURRENT

    ___source!___ SET-CURRENT
    : | ( order-spec c #tib -- input-specification )
        #tib !              \ restore #TIB
        0 >in !
        source drop c!      \ restore char at tib=addr
        SET-ORDER           \ restore search order
        SAVE-INPUT          \ leave the input parameters for the string at addr
        [ compile] \        \ ignore the text in the string
    ;  immediate
SET-CURRENT

: source!   ( addr len -- )
    >r >r                   ( R: len addr )

    GET-ORDER               ( order-spec )
    ___source!___ 1 SET-ORDER

    r@ c@                   ( order-spec c )        \ a copy of char at addr
    [ char] | r@ c!                                 \ the word at (addr,1) will be |
    r> r> swap 1            ( order-spec c len addr 1 )

    EVALUATE                                        \ evaluate the word | at (addr,1)
                            ( input-spec )          \ input parameters for:
                                                    \ SOURCE = (addr,len),
                                                    \ >IN = 0, SOURCE-ID = -1
    RESTORE-INPUT drop
;
```

located at *addr*, thus setting TIB to *addr*.

7. The word | sets #TIB to *len*, >IN to 0, and restores the original value of the character at *addr*. Now the current input source context is what we wanted to have: the string at *addr* of length *len*, and >IN points to its beginning.
8. The word | restores the original search order.
9. The word | executes SAVE-INPUT which leaves the parameters of the current input source (the string specified by *addr* and *len*) on the stack.
10. The word | executes the word \ to make EVALUATE ignore the rest of the string.
11. RESTORE-INPUT is executed, which sets the input stream to the beginning of the string specified by *addr* and *len*.

Note that this will work even with *len* = 0.

The problems, and how they were solved

1. The visibility of an auxiliary word depends on the search order.

A special wordlist is used to contain the auxiliary word, and the search order is set to this wordlist before the auxiliary word is EVALUATEd.

2. The standard does not provide a word to change the value of TIB.

EVALUATE is used, and special care is taken to neutralize the effect of processing the string by the text interpreter.

3. The string specified by *addr* and *len* does not begin with

Listing Two. The testing code and its output.

```
\ ---------------------- testing words --------------------
\ Output: ( addr len ) "contents-of-TIB"
: x   cr ." ( " source swap . . ." ) " [char] " emit source type [char] " emit cr
;

\ Output: ( addr len ) "contents-of-TIB"
\         f
\ where f is the flag returned by RESTORE-INPUT
: t1  save-input s"  " source! x restore-input . ;
: t2  save-input s" test#1" source! x restore-input . ;
: t4  save-input s" a" drop 0 source! x restore-input . ;
: t5  save-input s" a" source! x restore-input . ;

: $create >r save-input r> count source! create restore-input ?io ;
: $const    $create , does> @ ;

\ --------------------- testing ----------------------------
t1 1 . 2 .

( 24000023 1 ) " "
0 1 2
t2 1 . 2 .

( 24000041 6 ) "test#1"
0 1 2
t4 1 . 2 .

( 24000065 0 ) ""
0 1 2
t5 1 . 2 .

( 24000087 1 ) "a"
0 1 2

5 c" five qwe" $const
five .
5
```

a word that can process the rest of the input stream.

The name of such a word is written to the beginning of the string at *addr*. The original contents of the string at *addr* is restored when the auxiliary word executes. This imposes a restriction: *addr* must not be in ROM.

4. The name of the auxiliary word must be separated by a space from the rest of the string (that is, one more character must be written to the beginning of the string, if *len* > 1).

The length of the string that is passed to EVALUATE is exactly 1, and the delimiting space is not needed. The proper input buffer length is stored to #TIB by the auxiliary word, *after* it has been parsed and called.

5. It is difficult to pass the procedural context (that is, the rest of the procedure's code and the return stack state) to the auxiliary word being EVALUATEd in the string. We can establish the required input source context (with EVALUATE), but we cannot import the procedural context into the scope of this input source context.

Instead, we return the input source specification from the auxiliary word.

6. When EVALUATE finishes, the original input source specification (that is, the one that was in effect before EVALUATE) is restored, while we want it to be changed.

The word within the string being EVALUATEd performs SAVE-INPUT, and when EVALUATE finishes, RESTORE-INPUT is performed.

Note that the things that cannot be done "from outside" a context are done by an auxiliary word working "from inside" this context.

Why so complex

Indeed, if we did not want to write an ANS Forth standard program, we could write

```
: SOURCE!   ( addr len -- )
  #TIB ! 'TIB !  0 >IN ! ;
```

assuming that TIB is defined as

```
: TIB         ( -- addr )       'TIB @ ;
```

So, 21 lines of formatted standard code have been required to express what can be done by one line of non-standard code. As we mentioned in the beginning, this is a pitfall of the standard.

What is good

The *raison d'etre* of this paper is to accentuate some interesting properties of manipulating contexts—in particular, of manipulating the input source context with the restriction that the nesting of procedural contexts cannot be changed.

1. We have done this by evaluating the text.

We could either dynamically define the first character in the string as a function (which would pose several other problems), or change the first character to a predefined value. In both cases, we make our data executable at run time.

2. We changed the data to let it be interpreted, and later reversed the change.

3. We have done this by *exchanging* the contexts.

Indeed, we call EVALUATE which first establishes its procedural context and then its input stream context; then we modify and save the input stream context; then we leave the EVALUATE procedural context along with its input stream context, and finally restore the saved input stream context.

4. Really, we *exported* the input stream context from within the procedural context.

5. The operations of context save and restore may be used to export a context from within another context.

Indeed, we save the input source context from within EVALUATE and restore it when EVALUATE finishes.

6. We had to modify *context*.

Indeed, everything is done via global variables, the aggregate values of which form the input stream context. It is easier to think in terms of contexts than in terms of arbitrary sets of parameters. The word "context" means that its components are inter-related. Usually contexts are thought of as things that are created automatically; sometimes programming tools provide operations for saving/restoring them, while support for manipulating contexts as a whole is weak.

7. In our case, the most useful thing would be an operation of *creation* of a context with required parameters.

We have emulated such operation via non-elementary operations of:

(a) creation of two contexts (procedural and input source), interpretation, and restoring the previous two contexts;
(b) saving the input source context;
(c) restoring that context.

This resembles the use of a *cut* statement in Prolog to emulate an *if*: in both cases, simple things are expressed via complex things. The good news is that this is possible. The bad news is that simple notions are noticed when compound notions are compared.

The final note

I have tried these definitions on Win32For ver. 3.5 and GForth ver. 0.3.0, and it did not work in either. Win32For's RESTORE-INPUT does not leave the flag required by the ANSI standard; GForth's RESTORE-INPUT throws an exception, "argument input source different than current input source," although (1) there is no reason to do this in cases of string (via EVALUATE) and terminal input, and (2) this situation must be indicated by the flag which "is true if the input source specification cannot be so restored" (cited from the standard). To correct this, I had to redefine RESTORE-INPUT; but if the end goal is to implement SOURCE!, it is easier to write a system-dependent definition.

This is one more important lesson: even if you write a standard program, there is no guarantee that the implementors of ANS Forth systems read the standard enough carefully to understand all its dim places. We can only wish the next standard to be *easily readable* (if it was a program, would you call it readable?), otherwise such situations will happen.

Listing Three. Test code, and its output, which shows that execution of EVALUATE-based version of SOURCE! cannot restore the input stream specification.

```
: t3
   >in @ source
   s" test number 3" source!       x
   source! >in !
   ." source-id=" source-id .      x
;

t3 1 . 2 .

( 2400011B D ) "test number 3"
source-id=-1
( D000FD4 A ) "t3 1 . 2 ."
1 2
i/o error 6h, i/o routine xt=242A name= READ-LINE
```

Part One

Adaptive PID

Skip Carter • Monterey, California
skip@taygeta.com

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

Introduction

If you have been following the thread of this column, you will recognize that we now have all the background we need to create an adaptive PID controller. With this installment we will proceed with its design. Writing this installment presented me with a difficult problem. Some of you are aware that I have been very gentle with the mathematics behind the topics I have covered in the past. From the feedback I have received, it would seem that, in spite of my efforts to go easy, this column has a reputation for being a bit challenging. No matter how much you try to avoid or hide it, understanding adaptive controllers involves a lot of math.

As I looked at how to present adaptive controllers, I came to the conclusion that if you consider where these controllers get used (typically machinery, frequently *dangerous* machinery), one shouldn't be doing this as if it was from a cookbook. You really should know the math if you are doing this stuff. So be forewarned, there is a bit of math in this installment. There is enough here that we will be presenting the adaptive controller in two parts; this time we will do the math, and next time we will look at an implementation.

The importance of being linear

While there is a good deal of mathematics behind adaptive controllers, it's not particularly hard mathematics. The reason for this is that traditionally most controllers are *linear*. We can take advantage of this linearity to make the equations relatively easy to manipulate. Let's first consider what is means for a system to be linear. Essentially, linearity means that the system obeys a superposition principle. Suppose that $f(.)$ represents our system and, further, that A and B are two valid but otherwise arbitrary solutions to f. Then *if* it is true that,

$$f(A + B) = f(A) + f(B) \quad (1)$$

then the system is said to be linear. Many familiar systems have this property. It is equation (1) that allows us to decompose a periodic signal into frequency bands and calculate a power spectrum. Potential fields (electric, magnetic, and gravitational) are also linear. The main reason why linear systems are so familiar, is *not* because they are so ubiquitous (in fact, one author has pointed out that dividing nature into linear and nonlinear systems is like having "nonelephant" biology as a special subfield, and missing the fact that most systems are not linear), but because equation (1) makes linear systems solvable. Many nonlinear systems are handled by making them approximately linear, e.g.:

$$f(A + B) = f(A) + f(B) + \text{a little bit extra} \quad (2)$$

The work then primarily concentrates on how small that little bit actually is, and under what circumstances it stays small. For many nonlinear systems, (2) is a practical approach that gives useful answers. Systems that can be analyzed this way generally get described with phrases like "small amplitude," a dead giveaway that something like (2) was used. A simple example of this is the ordinary pendulum. A pendulum is actually a nonlinear system, but for small amplitude excursions (say ten degrees), the nonlinear effects are extremely small and can be ignored for typical applications. Some nonlinear systems *cannot* be broken down to something like (2) without completely missing the real solutions. Any system that has chaotic behavior is like this, the chaos comes from the nonlinearity; there are no chaotic systems that are linear.

Many methods have been invented for dealing with linear systems; one that we will find useful here is the *Laplace transform*. For the system $F(t)$ the Laplace transform $\mathcal{L}\{F(t)\}$ is defined by,

$$\mathcal{L}\{F(t)\} = f(s) = \int_0^\infty e^{-st} F(t)dt \quad (3)$$

(Strictly speaking this applies only for $t > 0$.) Two properties make the Laplace transform particularly suited to our problem:

- Like the Fourier transform, it converts a linear differential equation into a polynomial. (You might not have realized this, but we can transform—Fourier or Laplace—an *equation*, not just a stream of data).
- Unlike the Fourier transform, it treats transients efficiently. The Laplace transform is, in fact, the impulse response function for a system.

It's a bit tedious to do the integrations required to do either a forward or an inverse transform by hand, so the Laplace transform is often done with the help of symbolic integration software or the use of tables in a handbook. Table One gives the Laplace transform for several useful mathematical functions. Combining this with some general transformation properties, given in Table Two, gives us the ability to determine the Laplace transform of a large number of useful functions without the need to explicitly solve (3). The forward Laplace transform is not too difficult to do numerically, but calculating the *inverse* transform numerically leads to problems with the numerical stability of the calculation; this is not a problem with software that is capable of doing the in-

Table One

$F(t)$	$\mathcal{L}\{F(t)\}$	
1	$\frac{1}{s}$	$s > 0$
t	$\frac{1}{s^2}$	$s > 0$
t^n	$\frac{n!}{s^{n+1}}$	$s > 0$
e^{at}	$\frac{1}{s-a}$	$s > 0$
sin at	$\frac{a}{s^2+a^2}$	$s > 0$
cos at	$\frac{s}{s^2+a^2}$	$s > 0$
sinh at	$\frac{a}{s^2-a^2}$	$s > 0$
cosh at	$\frac{s}{s^2-a^2}$	$s > 0$

Table Two

$\mathcal{L}\{F(t)\}$	$f(s)$
$\mathcal{L}\{F(at)\}$	$\frac{1}{a} f\left(\frac{s}{a}\right)$
$\mathcal{L}\{c_1 F_1(t) + c_2 F_2(t)\}$	$c_1 f_1(s) + c_2 f_2(s)$
$\mathcal{L}\{F'(t)\}$	$s f(s) - F(0)$
$\mathcal{L}\{e^{at}F(t)\}$	$f(s-a)$
$\mathcal{L}\{F(t-a)\}$	$e^{-as}f(s)$
$\mathcal{L}\{\int_0^t F(u)du\}$	$\frac{f(s)}{s}$

verse transform symbolically.

We will use the Laplace transform to work out how the controller will respond to its inputs. We need to do be able to do this because there is no unique way to set up an adaptive controller—a motor speed controller that uses a shaft angle encoder will be quite different from one that uses a tachometer.

The controller equations

Recall from "Closing the Loop" (*FD* XVIII No. 5) that the equation for a *proportional-integral-derivative* (PID) controller is:

$$z(t) = K_p\varepsilon(t) + K_i\int_0^t \varepsilon d\tau + K_d\frac{d\varepsilon(t)}{dt} \qquad (4)$$

where K_p is the proportional gain, K_i is the integral gain, and K_d is the differential gain.

The quantity e is an error signal that is the difference between the commanded input, x, and the output of the controlled system y. In our earlier investigations, we were not much concerned with the internals of the controlled system (often called the *plant*), all we needed was the output signal. Figure One shows a generic schematic of our controller and plant.

For an adaptive system, we need to have an additional equation that describes how the plant behaves so that we can properly adjust the parameters of the controller. So now we have two equations to consider, the controller and the plant. The design of the adaptiveness depends upon the form of *both equations*. For our example plant, we will use the second order differential equation:

$$F(x) = \alpha\frac{d^2x}{dt^2} + \beta\frac{dx}{dt} + \gamma x \qquad (5)$$

This is a pretty generic model; as an example, this can be thought of as a damped mass-spring system where a is the mass, b is the friction, and g is the spring constant. *F(x)* would represent the imposed external forces on the system (the input), and the solution to the equation would give the plant's response to it.

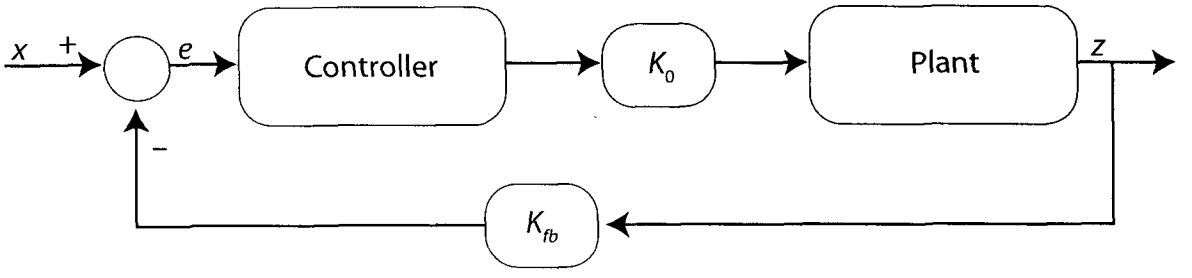
Analyzing the adaptive controller

We will start with the Laplace transform of the plant equation (5), since it's a little simpler to do. The first step is to use the linearity property of Laplace transforms, which is given as the third rule in Table Two. This property allows us to do the transform of (5) by doing the transform of each term separately,

$$\mathcal{L}\{F(x)\} = \mathcal{L}\{\alpha\frac{d^2x}{dt^2}\} + \mathcal{L}\{\beta\frac{dx}{dt}\} + \mathcal{L}\{\gamma x\} \qquad (6)$$

and further the linearity property allows us to move the constants outside of the transform operation,

Figure One



$$\mathcal{L}\{F(x)\} = \alpha\mathcal{L}\{\alpha\frac{d^2x}{dt^2}\} + \beta\mathcal{L}\{\frac{dx}{dt}\} + \gamma\mathcal{L}\{x\}$$
$$= \alpha s^2\mathcal{L}\{x\} + \beta s\mathcal{L}\{x\} + \gamma\mathcal{L}\{x\} \quad (7)$$

Now representing L{x} as *f(s)*, we get,

$$\mathcal{L}\{F(x)\} = \alpha s^2 f(s) + \beta s f(s) + \gamma f(s) \quad (8)$$

The transfer function is the ratio of the output response *f(s)* to the input forcing L{*F(x)*}, which we get by rearranging the above equation to get,

$$G_p(s) = \frac{1}{\alpha s^2 + \beta s + \gamma} \quad (9)$$

Now that we see how we go about doing this, it's a straightforward matter to do the same for the PID controller,

$$\mathcal{L}\{z(t)\} = \mathcal{L}\{K_p\varepsilon(t)\} + \mathcal{L}\{K_i\int_0^t \varepsilon d\tau\} + \mathcal{L}\{K_d\frac{d\varepsilon(t)}{dt}\}$$
$$= K_p\mathcal{L}\{\varepsilon(t)\} + K_i\mathcal{L}\{\int_0^t \varepsilon d\tau\} + K_d\mathcal{L}\{\frac{d\varepsilon(t)}{dt}$$
$$= K_p f(s) + K_i\frac{1}{s}f(s) + K_d s f(s) \quad (10)$$

(we needed the derivative rule, number four, and the integral rule, number seven, from Table Two as well as the linearity rule). Now we have to be careful here, the transfer function is the ratio of the output to the input. For the PID controller the error signal, e, is the *input*, the output is the quantity *z*.

So the transfer function for the PID controller is,

$$G_c(s) = K_d s + K_p + \frac{K_i}{s}$$
$$= \frac{K_d s^2 + K_p s + K_i}{s} \quad (11)$$

Now we need to couple the controller and the plant. We will connect the output of the controller into the plant by taking the controller output, multiplying it by a plant gain factor K_0, and using that as the plant input. This is represented mathematically by multiplying the controller response by the plant gain and the plant response,
$G_c \times K_0 \times G_p$

We will connect the plant output into the controller at the negative side of the summing node (to get an error signal) after multiplying it by a feedback gain factor K_{fb}. The plant and feedback gains do not really change anything, they just give us more opportunities to adjust things. The real change is the fact that the input to the controller is now reinterpreted as the output of the summing point (before it was just "the input," now we care how it relates to the rest of the system). So now, the system output becomes a portion of the system input,
$x - K_{fb}z$

We get the full equation by going step at a time through the diagram (Figure One). At the output of the summing node we have,
$x - K_{fb}z$

then after the controller we get,
$(x - K_{fb}z)G_c$

and so, after the plant we have,

$$z = (x - K_{fb}z)\, G_c K_0 G_p \quad (12)$$

The response function is the ratio of the output *z* to the input *x* which we can determine by reorganizing the above equation to get,

$$G(s) = \frac{z}{x} = \frac{G_c K_0 G_p}{1 + K_{fb} G_c K_0 G_p} \quad (13)$$

now we expand this using (9) for G_p and (11) for G_c and simplify,

$$G(s) = \frac{\left(K_d s^2 + K_p s + K_i\right) K_0}{\alpha s^3 + \left(\beta + K_{fb} K_0 K_d\right) s^2 + \left(\gamma + K_{fb} K_0 K_p\right) s + K_{fb} K_0 K_i} \quad (14)$$

This is the response function of our PID controller with the plant defined by equation (5). Notice that we managed to go from a description of how the controller is interconnected (basically Figure One) all the way to its input response function with nothing more than polynomial manipulations. If we had not used Laplace transforms, getting the response function by the direct manipulations of the integro-differential equation would have been much harder to do.

It is important to recognize that the details of what we have done are dramatically dependent upon the forms of equations (4) and (5). However the *method* we used will apply as long as the two equations (and how they are coupled) are linear.

Now that we know how the adaptive controller will respond, how do we adapt it? First we need to understand what it means for the controller to be optimally adapted. To do this, we need to consider when the denominator of equation (14) is zero.

$$\alpha s^3 + \left(\beta + K_{fb} K_0 K_d\right) s^2 + \left(\gamma + K_{fb} K_0 K_p\right) s + K_{fb} K_0 K_i = 0 \quad (15)$$

This equation is called the *characteristic equation* for the system. For the moment we will combine the coefficients,

$$As^3 + Bs^2 + Cs + D = 0 \quad (16)$$

The locations of the solutions to (16) in the complex plane can be used to predict the behavior of the controller. For a given set of coefficients, there will be three solutions called the *roots*. If the roots are in the negative half of the plane, the controller will be stable. If the roots are real but unequal, the system is overdamped (that is, it will never quite recover from a suddenly imposed step input). If the roots are imaginary, the system is underdamped (which will "ring" when it gets a step input). The optimal response to an imposed step is the critically damped case; this will happen if the roots are real and equal.

So our goal is to adjust the gains (the various K values) so that the characteristic equation always has real, equal, negative roots. It turns out that we can achieve all these constraints provided that in equation (16), A and B have opposite signs and that

$$C = \frac{B^2}{3A} \quad (17)$$

$$D = \frac{B^3}{27A^2} \quad (18)$$

Clearly, we cannot optimally adaptively control an arbitrary plant. If a and b don't have opposite signs, the controller won't even be stable. Also note that there are not enough equations to give us independently all five gains. The feedback and plant gains K_{fb} and K_0 are either going to have to be defined to have fixed (known) values or they will need to be absorbed into the other gains.

Conclusion, Part One

The next step is to use our knowledge of least-squares methods to find the minimizing solution to the mean value of z^2. We need to do this while still satisfying the constraints imposed by the characteristic equation. The stage is set, we know what to do, but it's going to take several more pages to do it. Consequently, I will continue this next time.

Feedback

Wil Baden sent me a copy of his version of my least squares estimator program from last time (Listing One). His version uses his formula translator, which he has described in his column. This nice thing about his version is that you can read the algorithm directly from the expressions in the program, leaving expansion of the equations into "traditional" Forth to the translator.

Please don't hesitate to contact me through *Forth Dimensions* or via e-mail if you have any comments or suggestion about this or any other Forthware column.

See Listing One on next page.

References

Boyce, W.E, and R.C. DiPrima, 1969; *Elementary Differential Equations*, John Wiley & Sons, New York.

Kaufman, H., I. Barkana, and K. Sobel, 1998; *Direct Adaptive Control Algorithms Theory and Applications*, Springer Verlag, Berlin. ISBN 0-387-94884-8

McKerrow, J.P., 1991; *Introduction to Robotics*, Addison-Wesley, Sydney, ISBN 0-201-18240-8

Listing One

```
// FVARIABLE || sumx sumxz sumz sumx2 \\

: Lsq-Init                ( -- )
    0 n !
    // let | = 0: | sumx sumxz sumx2 sumz \\
;

: Calc-Det                ( F: -- d )
    let {n @ S>F}*sumx2 - sumx*{FDUP}:
;

: Estimate                ( F: -- b a )
    let x = {Calc-Det}:

    \ Calculate b and a
    let (sumx2*sumz - sumx*sumxz) / x, ({n @ S>F}*sumxz - sumx*sumz) / x:

;

: Lsq                     ( --<infile>-- )
    Lsq-Init

    next_file                    ( str len)

    R/O OPEN-FILE ABORT" Unable to open input data file. "
        TO fin                   ( )

    CR

    fin get-int  DUP n !         ( n)

    0 DO                         ( )
            I .
        let x = {fin get-float}:  \ Get X point
            x F.
        let sumx = sumx + x:
        let sumx2 = sumx2 + x*{FDUP}:

        let z = {fin get-float}:  \ Get Z point
            z F.
        let sumz = sumz + z:
        let sumxz = sumxz + x*z:
            CR
    LOOP

    fin CLOSE-FILE DROP

    Estimate                ( b a)
    ." slope (a) : " F.
    ." intercept (b) : " F. CR  ( )
;
```

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).

Corporate Sponsors

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.

Digalog Corp. (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, 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) 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

An interactive programming environment for writing WindowsNT and Windows95 kernel mode device drivers in Forth.

www.theforthsource.com

Silicon Composers (web site address 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

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 Incoporated • 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

Everett F. Carter, Jr.
Guy Grotke
John D. Hall
Zvie Liberman
Gary S. Nemeth
Marlin Ouverson

FORTH SECRETS REVEALED!

FORTH TECHNOLOGY AND ITS APPLICATION

Call for Papers

18th Rochester Forth Conference
June 24th – 27th, 1998
University of Rochester
Rochester, NY

Hosted by the
Institute for Applied Forth Research, Inc.

Sponsored by
FORTH Interest Group
Dash, Find & Associates
Taygeta Scientific, Inc.
Diversity Dynamics

For more information:

Rochester Forth Conference
Lawrence P. G. Forsley, Conference Chairman lpgforsley@aol.com
P.O. Box 1261 Annandale, VA 22003 716-235-0168