$10 Volume XIX, Number 3 September 1997 October

FORTH DIMENSIONS

NOW BEING SERVED

For the past two years the Forth Interest Group has been maintaining several e-mail based special interest sections. These groups allow its members to exchange information about special Forth topics directly with other members who share the same interest.

This can be much more efficient than using network newsgroups for asking questions about, say Win32For, by sending e-mail to fig-win32for@forth.org because fig-win32for is a list of people who are particularly interested in Win32For.

Any FIG member can join any of the special interest mailing lists by filling out the form at:

http://www.forth.org/fig/sigs.html

We have recently added an automated mailing list server called Majordomo. This new service allows you to subscribe, unsubscribe, and learn other information about any of the lists purely by e-mail.

You still mail to, for example, fig-win32for@forth.org to exchange messages with other members of that list. But you use majordomo@forth.org to send messages about the management of the list (all of the lists, not just fig-win32for).

To get help on using the Majordomo mailserver, send e-mail to majordomo@forth.org with "help" as the message content. Majordomo will receive your request and e-mail its help file back to you.

You can join a list by sending the message:

subscribe <listname>

Sending the message "lists" will cause you to receive a list of all the special interest sections.

Details on these and other commands are given in the help file, so sending a help request is the recommended first step.

OFFICE NEWS

Autumn is beautiful in Carmel! We hope we'll be seeing many of you at FORML '97 this year. The Monterey peninsula (Carmel, Monterey, Pacific Grove, Pebble Beach, Sand City, and Seaside) is a fabulous place to "get away to" in late November.

Did you know that FORML '97 Early Registration will give you an additional 10% off the Conference fee? And it helps us to plan—please take advantage of it! The Early Registration cut-off is November 1st. Registrations taken after that date will be at the full fee. Also, there are a limited number of rooms, another reason to register early!

This year we also are offering Corporate Sponsorships. If you work for a corporation, or own one, and would like to know more about sponsorship, please contact the office and we'll be happy to provide the details.

As an additional reminder, please let us know at your earliest convenience the title of your talk for FORML '97. You can use FORML@forth.org to get in touch with Guy Kelly, Conference Chair.

The Forth Interest Group, as you know, is a member-supported, non-profit organization. This past year, FIG has undergone many changes in both personnel and office procedures. We're currently in the process of revising the Chapter Kit to better serve your needs to connect with each other and to interact with the FIG office. And, of course, you've no doubt noticed the new look and design of *Forth Dimensions*. On the drawing board, we have a Membership Drive to increase FIG's membership, as well as a planned Donation Drive and Fund Raiser. More information about these activities will be in future issues of *Forth Dimensions*. There are even plans for the office to start posting weekly information about our activities on comp.lang.forth.

All in all, these activities are designed to keep you better informed about your Forth Interest Group. Naturally, if you have any questions, don't hesitate to contact me at the office.

Thank you all for your continued support!

Cheers!

Trace Carter
Forth Interest Group
100 Dolores Street, Suite 183
Carmel, California 93923
voice 408-373-6784 • fax 408-373-2845

CONTENTS

DEPARTMENTS

EDITORIAL

Our Biggest Failure

A telling phrase shows up in this issue's report from the annual euroForth Conference. The author comments that there appear to be more applications and users of Forth than even its adherents believe.

This mirrors my own experience. As the (perhaps interminable) editor of this publication, one of my jobs through the years has been to find articles about interesting, and hopefully inspiring, uses of Forth. But finding application stories has turned out to be one of the more challenging aspects of this job, while some years have found debates over CASE statements and the implementation details of object orientation filling our pages. The latter kind of low-level discussion is, of course, both appropriate and welcome; if the situation were reversed, I'd lament the lack of such material.

The fact is, our community has never excelled at putting its best face toward the spotlight. I remember when a group of FIG personalities were photographed for *Rolling Stone*, replete with FIG t-shirts and, if memory serves, wielding figForth listings. How more appropriate, and what a different public perception would have been achieved, at least among those readers, if the article had included a shot of a Forth-controlled laser light show. (Presuming, of course, that *Rolling Stone* intended a straight story; but you get my drift.)

Of course, Forth applications abound, both large and small, embedded and not. It is in cars, runs an international airport, puts the fun in amusement parks, and even operates the national telephone network of a country in the Far East. There are other big apps, and countless smaller ones, but we hear about very few. Most I only know about because of my position with *Forth Dimensions*, and most of the insiders at such projects don't write about them for us or for any of the industry trade publications.

The reasons, I've been told, are many. The programmers are too busy working on their next project. The language used is a company secret. Writing doesn't pay enough. The team's scarce narrative skills are conserved for code comments. The egos involved can deliver functional, maybe brilliant, code (because they must and they can, and because it is judged on familiar, defensible terms) but shy from exposing their prose to public criticism.

It isn't only application stories that we have been poor at telling. Perhaps as a side effect of (usually unnecessarily) avoiding specific discussion of performance metrics, our collective voice has never, to my own way of thinking, learned to concisely, consistently, and persuasively articulate the importance of things like development costs and the performance/resource ratio.

The unfortunate result is that, like our European correspondent implies, the perception grows that Forth isn't used much. In the absence of knowledge, in some quarters the belief might even take hold that Forth isn't *good* for much. Now, it is easy to counter such attitudes when they surface; but they rarely surface in the presence of someone capable enough to dispel them—more often they lurk among the unspoken and subjective prejudices that influence which solutions will even be *considered* for a project. It is for this reason, not for our own general interest and morale, that Forth success stories need to be broadcast in as many venues as possible.

It has been pointed out to us that application stories do not sell products. But that is the difference between sales and marketing, and what I am talking about is marketing—the biggest failure of the Forth community as a whole. Marketing is the groundwork without which every sale, every contract, is an exercise in persuasion and fortitude. Without it, we have to stick our foot in the doorway; with it, the door is open and waiting for our call.

Not interested in marketing? Then get interested in your own Forth work, or that of your colleagues. Budget into your estimates, as necessary overhead, the time that will be required to document your projects in the best prose you can produce. Extend your attention span and write. Write for us, write for others—either way, such a collective practice will accrue benefits to you (publicity), to the Forth language (marketing), and to your next negotiation (sales).

Marlin Ouverson
editor@forth.org

Forth Dimensions

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 ($53 Canada/Mexico, $60 overseas air). For membership, change of address, and to submit items for publication, the address is:

Forth Interest Group
100 Dolores Street, suite 183
Carmel, California 93923
Administrative offices:
408-37-FORTH fax: 408-373-2845
office@forth.org
www.forth.org/fig.html

The Forth Interest Group

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

FORTH DIMENSIONS (ISSN 0884-0822) is published bimonthly for $45/53/60 per year by 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.

ROCHESTER '97 REPORT

The 1997 Rochester Forth Conference on "Portable Software" has come and gone. It was good to be there again, back in Rochester, a chance to mingle with others in your field of work and get exposed to new and interesting ideas others are working on. A chance to catch up with annual friends.

Larry Forsley, the conference organizer, opened the smallest conference gathering in years with the general warmth and enthusiasm that is his nature. There were roughly 30 attendees, and (according to the schedule) 21 talks in all.

Thursday and Friday were filled with talks followed by working groups. On Friday, we wound up everything at a wonderful Thai restaurant for supper. Saturday saw the 4K run, walk, stroll, sleep. I elected for the 4K sleep, where the unit of measure was not meters but seconds.

On Saturday, there were four one-hour focus sessions on different topics. The closing of the conference brought together the stragglers (or, as Larry likes to call them, the survivors) back at the Forth Institute for a barbecue.

The Talks

Thursday

Skip Carter opened with a talk on how his oceanic robot (maybe it's an *obot*) evolved from smart instruments.

Allan Anway updated us about his continuing success and growth using FORTH, Inc.'s Express software to control a lime kiln.

I gave a talk (available on the web at http://www.compusmart.ab.ca/rc/Timbre/AWritableComputer.html) about my research with writeable computers over the last year with a malleable model of a stack processor.

Richard Haskell described how Whyp was used to create embedded systems, and demonstrated a digital, handheld computer compass.

Elizabeth Rather gave a quick overview of where FORTH, Inc. plans to take their cross-compiler technologies now and in the future. They have some nice products, with report cards to attest to the gains they've made with their next-generation compiler technologies.

I gave a talk (available on the web at http://www.compusmart.ab.ca/rc/Timbre/TimbreUnitII.html) on Timbre, introducing the ten forms of Timbre and the Tu of Timbre.

Peter Knaggs entertained us with his experiences of learning Perl and then comparing it to Forth.

Randy Leberknight painted a picture of how Open Firmware creates a nice, portable infrastructure for hardware systems.

Stephen Pelc talked about a portable open software architecture for industry.

John Rible, a man I consider one of the true miners of Forth, treated us to an update on his gem, the QS2116 BRISC Microprocessor and its hardware threaded-code interpreter.

Penio Penev concluded the talks for the day with an active discussion about mapping the Forth virtual machine in a RISC environment.

In the last part of the afternoon, Larry conducted a working group focused on: What are the year 2000 problems? This brought forth a lively discussion from the audience, with various war stories, predictions, and some knowledge. I discovered that my PowerBook rolled over a two-digit date at the year 2019 to 1920, or something like that. It's probably when the warranty expires.

Friday

Friday morning brought us back together to hear Ken Gifford talk about Ion Implanters for Dummies.

Olive Shank followed with Software Life Cycles.

Steve Kunz described how video image analysis can be done with Forth.

Next was a talk on the Essential Service Nodes (ESNs) being developed within NASA towards plug-and-play spacecraft systems.

Peter Knaggs described some work he'd been involved with in marrying Tcl/Tk and ProForth, and how it was used as a bilingual scripting environment.

Skip Carter updated us with Forth Script II: Why Isn't Everyone Using It? (We are, but in different dialects...)

We were then treated to a swift life story by Martha Chernoch, and how she grew with the computing age.

Warren Bean gave a talk on growing hardware from hardware.

The last talk was from a conference regular, Brad Rodriguez (by now a Ph.D. graduate): Toward a Distributed, Object-Based Forth.

The afternoon consisted of four working groups held so that each person could attend at least two of them.

Saturday

The conference wrapped up with Saturday as an open day for drop-in visitors and four one-hour tutorials.

I ran the first tutorial, on Timbre Tools and Applications. I talked about how Timbre is used, and showed a kernel compiler, complete with optimizer, implemented in Timbre.

Elizabeth Rather ran the second tutorial, and spoke about FORTH, Inc.'s cross-compilers.

I ran the third tutorial, after lunch, going much deeper into the work I'd done with writeable computers. I explained the architecture, and demonstrated it with statistical addition.

The fourth tutorial was a focus group on the *Journal of Forth Application and Research*.

In Closing

The papers from these talks will, of course, be available in the conference proceedings from the Forth Institute.

A Tribute to Larry

I've got to give the Forsleys (Brenda, Larry, Alex, and Amara) lots of credit. They delivered the conference right on time, in the year it was supposed to happen, with real-time defect removal and software co-routines for those other problems.

Rob Chapman • Edmonton, Alberta, Canada
rc@compusmart.ab.ca

Rob Chapman is a design engineer with New Micros, working on embedded and real-time systems. His research includes translation (Timbre) and mutable hardware.

euroFORTH '97 REPORT

euroFORTH '97 was held at St. Anne's College (Oxford, England) on September 26–28, with the main topic of "Embedded Communications." A copy of the proceedings is available from MPE Ltd.

This year's conference had more delegates and visitors than ever before, and demonstrated that Forth is more widespread than even its practitioners believe. The applications handled by Forth are increasing in size and complexity, and some of the papers discuss methodologies to handle complexity in software—in terms both of code size and of team programming—that can be applied to modern Forth systems.

The theme of this year's conference was Embedded Communications, and I am very pleased that we had several papers on this topic, including the use of the IX1 stack, networking papers covering the CAN fieldbus, and a TCP/IP stack in Forth. As ever, the actual topic of a euroForth conference emerges when we recieve the papers. This year, one of the emerging themes was safety and certification, particularly as part of the problems of managing large software projects. This is related to an interest expressed before the conference by several delegates in source documentation and management.

Apart from our thanks to all delegates for their enthusiasm and their papers, our thanks also go to those who organised the conference. Peter Knaggs and his committee looked after the refereed papers. Joan Perham and Sarah Windless performed the administration of the conference, and its successful running is due to them. The conference sponsors enabled money to be available for student sponsorship.

This information is part of a report offered by Peter Knaggs on the euroForth web page at http://www-cis.paisley.ac.uk/forth/euro/ef97.html, where more details of the event can be found.

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.

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.

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.

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.

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

Individual Benefactors

John D. Hall

Writing a Macintosh Application with Pocket Forth

Pocket Forth by Chris Heilman is a small, freeware Forth for the Macintosh. It is wonderful in its simplicity. This article will outline how to use Pocket Forth to write simple Macintosh applications. I claim no expertise apart from that gained by having done it before. Pocket Forth is not a complete development system, and needs to be treated differently than other Mac development systems, including the much more powerful Mops/Yerk pair.

Most people learn best by example. Therefore, we will create an example: a simple drawing program called MouseDraw. Along the way, certain Mac features will be described, but only so far as they relate directly to the task at hand. If you want to learn the ins and outs of Mac programming buy the *Inside Macintosh* books. Then come and teach me. To those who shiver at the sight of the word "Macintosh," I urge you to forge ahead. Using Pocket Forth is more a hacker's thing than most Mac programming is. You just might like it!

Pocket Forth

PF is small, weighing in around 18K, including all the resources. It is also reasonably fast. It has no editor, no assembler, and the dictionary size is 26K. When run, you are given a simple 64 × 16 character interpreter window and an `ok` prompt. Source files can be read from disk via `--> filename`, `open`, or the File menu. The last two display a standard Macintosh open dialog. PF's native types are signed 16-bit integers and 10-byte floats with a shared data stack. PF has no built-in disk access, but makes up for this by its ability to access all of the Macintosh Toolbox. In essence, PF is the minimum programming language for the Macintosh. It provides complete access to the whole machine, with the power of Forth as the means to tie it all together. The dictionary can be saved and a startup word defined (more on that later), so it is capable of creating fully compiled, standalone applications.

In PF Toolbox access, this is handled via assembly language traps. By design, PF's return stack is the system stack, allowing necessary parameters to be passed to the return stack just before a Toolbox routine is called. For example:

```
: pensize ( height width -- )
  >r >r  ,$ A89B ;
```

`Pensize` takes two 16-bit arguments, passes them to the return stack, and calls the QuickDraw function to resize the drawing pen (trap $A89B). In this way, any Toolbox function can be used.

The goal

The goal is to create a standalone application. In order to accomplish this, PF must be transformed. This is a two-step process: first, load your code, replacing PF's handlers with your own; second, use a resource editor to alter PF's resources to complete the application. This is one step beyond the usual turnkey approach.

Why two steps? All Macintosh files can contain a resource fork and a data fork. Usually, applications are all resource fork while everything else is all data fork. The Macintosh accesses the application's resources as needed. When an icon is to be displayed, the icon is read from the resource fork. When a new window is to be created, the necessary data is read from the resource fork. This is why most Macintosh applications are completely contained within a single file. A resource editor is absolutely essential to programming the Macintosh in any language; ResEdit by Apple is as good as any other. The PF resources we will be concerned with here are those that deal with menus, icons, dialogs, and windows.

The example

This is the goal: a simple program that will let us draw with the mouse in a small window. For thrills we can add a "button" for clearing the screen, and some menu options for adjusting the size of the drawing pen. We also want the screen to properly update itself when it needs to.

The events

There is one thing to remember: GUI systems, like the Mac, are *event driven*. Basically, applications sit around and wait for the user to do something, then they react to that event. Most Mac applications deal with this via a Toolbox routine called WaitNextEvent. Pocket Forth deals with this by providing three words that, as a side-effect, handle events for us. They are all related to the keyboard: `key`, `expect`, and `?terminal`. These operate in the usual Forth way, but they also handle events, such as mouse clicks and menu choices via command-key sequences. So, our application will contain an inner loop that listens for events:

```
: eventLoop ( -- )
  ( event loop, listen for events via KEY )
  initialize
  begin
   10000 10000 !pen ( move pen off screen )
    key drop  ( get key presses & ignore )
  again  ;
```

The word `initialize` sets up the application, followed

Ronald T. Kneusel • Milwaukee, Wisconsin
rkneusel@post.its.mcw.edu

Ron is a Systems Specialist for the Medical College of Wisconsin, and is a part-time biophysics graduate student.

by the event loop, which never exits. At first glance, it appears this is a meaningless infinite loop which gets characters from the keyboard and throws them away. This is exactly what it is. The secret lies in the fact that, while `key` is waiting for a keypress, it is also listening for events and responding to them via PF's built-in event handlers. Moving the pen off the edge of the screen removes the cursor from view.

Whenever the user does something, it triggers one of PF's event handlers. Most of these have default values and need not be changed. Some, of course, will be changed to place the application's response under the programmer's control. Handlers are stored as addresses in a table of *unnamed variables* accessed via offsets to a base word `+md`. The default handler's address is replaced with the new handler's address:

```
' myButton  16 +md !
```

This causes the word `myButton` to execute each time a mouse-button-down event is encountered. The PF documentation has a complete list of available handlers. For our sample application, we will use the following: menu one, items one, two, and four; the screen update handlers; the mouse button handler; and the startup word.

While it is possible to get PF to create new menus, it is easiest to simply appropriate the two existing menus and reassign handlers. The PF File menu contains eight items (three of which are separator lines); the Edit menu contains another six items. The menu handlers are stored in a list structure (Figure One). The words `18 +md` return the address of a handle, which in turn points to the beginning of the menu structure. The first element of the menu structure (two bytes) points to a list containing the addresses of the menu item handlers. So, to set the first menu's first item handler, we would use the cryptic:

```
' +pen  18 +md @ @ !
```

The remaining handlers are set in a similar fashion:

```
' -pen  18 +md @ @ 2 + !
( 2nd File menu item )

' bye   18 +md @ @ 6 + !
( 4th File menu item )
```

Notice that the third menu item handler has not been reset. We will be altering the MENU resource to change the names of the menu items. In the process, we will set item three to be a separator line (which cannot be selected, so there is no need to reset its handler).

The heart of our application is the word `mouse`, which is to be executed whenever the user holds down the mouse button. So we need to set the mouse-down handler:

```
' mouse  16 +md !  ( mouse button handler )
```

Next, in order to update the screen properly, we need to set the window update and activate handlers:

```
' myAct  12 +md !  ( new activate handler )
' myUpd  14 +md !  ( new update handler   )
' myVer  24 +md !  ( new version handler  )
```

Finally, we need to set the startup word so we can turnkey the application:

```
' eventLoop 26 +md !  ( startup word )
```

Where are we now? We are working with a copy of Pocket Forth which we will transform into our application. We have an event loop which will process events while waiting for keystrokes. We have decided which event handlers to update and have written words which implement our new handlers. Before proceeding further, let's take a look at the code itself.

The code

The complete source to MouseDraw, with line numbers added, is in Figure Two.

Lines seven through 42 define several Mac-specific utility words for changing fonts, drawing rectangles, and storing the application window for updates. *Rects* are rectangles, defined by their top, left, bottom, and right coordinates. There are several toolbox calls for drawing rects, erasing rects, and for determining if a point lies within a rect. Fonts are chosen by font number. The word `wsize` is used to resize the application window on startup; an alternative would be to change the default window size in the application's WIND resource (see below). The series of words in lines 23–42 are used to create a picture in memory and then copy it to the current window. This works for updating, because we set the handlers to store a copy of the window when we click off the application, and it is that copy that will be returned when the application is active again (i.e., made the front running application—the application will continue to run in the background).

The main program starts with line 47, which defines a word to alter the drawing pen size, as seen above. Next, the drawing

Figure One. Menu list structure

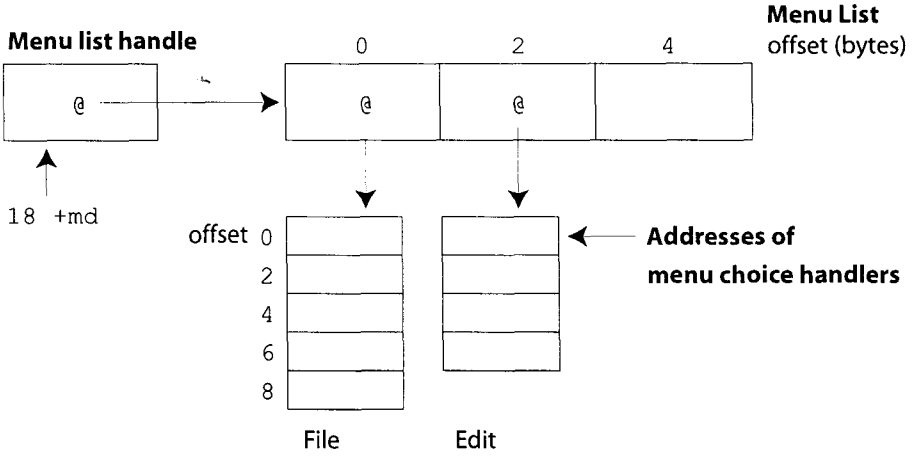

Figure Two. Complete source to MouseDraw.

```
001  \
002  \ A Simple Macintosh Application using Pocket Forth
003 \
004  \ RTK, 07-Feb-97, last update: 09-Feb-97
005  \
006
007  ( font, rect and widow management words.. C. Heilman, PF examples )
008
009  : wsize ( h v -- )  2dup 8 +md 2! 0 +md 2@ 2>r 2>r 256 >r ,$ A91D ;
010
011  : !FONT ( n -- ) >r ,$ A887 ; macro  ( _TextFont ) ( set font )
012  : !FSIZE ( n -- ) >r ,$ A88A ; macro  ( _TextSize ) ( set size )
013  : monaco ( -- )  4 !FONT  9 !FSIZE ;
014  : chicago ( -- ) 0 !FONT 12 !FSIZE ;
015
016  : rect ( create a named rect )  variable 6 allot ;
017  : fRect ( rect -- ) a>r ,$ A8A1 ;  ( _FrameRect )
018  : eRect ( rect -- ) a>r ,$ A8A3 ;  ( _EraseRect )
019  : pRect ( rect -- ) a>r ,$ A8A2 ;  ( _PaintRect )
020  : !rect ( t l b r rect -- )  >r swap r 4 + 2! swap r> 2! ;
021  : ?in ( h v rect -- b )  0 >r rot rot 2>r a>r ,$ A8AD r> ; ( _PtInRect )
022
023  4 +md constant WRECT  ( addr of window's rect )
024  : WINDOW ( -- window.pointer ) 0 +md 2@ ;
025  : WPICT ( -- dhandle ) ( the window picture's handle )
026     0 0 2>r  window 2>r ,$ A92F 2r> ;  ( _GetWindowPic )
027  : KPIC ( d -- ) 2dup or IF 2>r ,$ A8F5 ELSE 2drop THEN ;
028  : PICTURE ( rect -- dhandle ) ( open a picture leave its handle )
029     0 0 2>r  a>r  ,$ A8F3 2r> ;  ( _OpenPicture )
030  : PCLOSE ( -- ) ,$ A8F4 ; macro  ( _ClosePicture )
031  : PKILL ( addr -- ) 2@ kpic ; ( _KillPicture at addr )
032  : WPASSIGN ( handle -- ) ( ASSIGN a Picture to Window )
033     window 2>r  2>r  ,$ A92E ;  ( _SetWindowPic )
034  : BCOPY ( rect -- ) ( copy window bitmap to window )
035     window  2 0 d+ 2dup 2>r 2>r  ( window bits = source, destination )
036     dup a>r  a>r  0 >r  ( source rect, destination rect, mode )
037     window 24 0 d+ dl@ 2>r  ( mask to port visrgn )
038     ,$ A8EC ;  ( SrcCopy mode,  _CopyBits )
039  : WSAVE ( -- ) ( save the screen for updating )
040     wpict kpic  ( _KillPicture )
041     0 0  window 148 0 d+  dl!  ( zero window picture in window record )
042     wrect picture  wpassign  wrect bcopy  pclose ;
043
044
045  ( Start of MouseDraw code )
046
047  : pensize ( height width -- )  ( change the drawing pen size )
048    >r >r  ,$ A89B ;
049
050  rect CLEAR   250 5 265 52 CLEAR !rect  ( CLEAR box )
051  rect DRAW    20 0 246 1024 DRAW !rect  ( drawing field )
052
053  variable x  ( last position )
054  variable y
055  variable p  ( current drawing pen size )
056
057  : initialize ( -- )  ( setup screen )
058    400 275 wsize  ( set window size )
059    1 p !  ( set pensize )
060    page  3 15 !pen chicago ." MouseDraw"
061   monaco  ."   A simple drawing program" cr
```

```
062    CLEAR fRect 7 261 !pen ."  CLEAR"   ( draw CLEAR button )
063  ;
064
065  : +pen ( -- )  ( increment pen size by 1 )
066    1 p +!  p @ 10 > if 10 p ! then
067    p @ dup pensize ;
068
069  : -pen ( -- )  ( decrement pen size by 1 )
070    -1 p +!  p @ 1 < if 1 p ! then
071   p @ dup pensize ;
072
073  : ?click ( -- t|f )  ( true if mouse button clicked )
074    ?button if begin ?button 0= until -1 else 0 then ;
075
076  : draw_segment ( x' y' -- )  ( draw a segment from last point )
077       x @ y @ !pen  2dup -to  y ! x !  ;
078
079  : clear_box ( -- )  ( check if CLEAR clicked )
080     ?click IF  initialize  THEN ;
081
082  : mouse ( -- )  ( mouse button handler )
083       @mouse 2dup DRAW ?in IF
084          y ! x !                ( set initial position )
085       ELSE 2drop THEN
086       BEGIN
087        ?button                  ( while button still down )
088       WHILE
089         @mouse                  ( get mouse position )
090         2dup DRAW ?in IF        ( if in drawing rect )
091           draw_segment          ( draw the segment )
092         ELSE
093           CLEAR ?in IF
094             clear_box           ( otherwise, see if CLEAR clicked )
095           THEN
096         THEN
097         ?terminal drop          ( process an event )
098       REPEAT
099  ;
100
101  : eventLoop ( -- )  ( event loop.. listen for events via KEY )
102    initialize
103    begin
104      10000 10000 !pen  ( move pen off the screen )
105      key drop          ( get key presses and ignore )
106    again  ;
107
108  ( update and activate - for restoring the window )
109
110  : myAct  if null else WSAVE then ;
111  : myUpd  WSAVE ;
112  : myVer  WSAVE [  24 +md @ compile ] ;
113
114  ( setup handlers )
115
116  ' +pen  18 +md @ @ !        ( 1st file menu item )
117  ' -pen  18 +md @ @ 2 + !    ( 2nd file menu item )
118  ' bye   18 +md @ @ 6 + !    ( 4th file menu item )
119
120  ' mouse  16 +md !   ( set up mouse button handler )
121  ' myAct  12 +md !   ( new activate handler )
122  ' myUpd  14 +md !   ( new update handler   )
123  ' myVer  24 +md !   ( new version handler  )
124
125  ' eventLoop 26 +md !  ( startup word )
```

and clear rects are defined. All drawing will take place within the drawing rect. The clear rect will act as a button for erasing the drawing rect. The word `initialize` sets up the window and draws the text. It is called once by `startup`, which then goes into the `eventLoop` to listen for events. The handlers are set up at compile time, in lines 116–125.

The application has a single menu, with three options: increase pen size, decrease pen size, and quit. These correspond to the words `+pen`, `-pen`, and `bye`, respectively. The function of `+pen` and `-pen` is straightforward. The current pen size is stored in `p`, and is increased or decreased as needed.

The word `?click` returns *true* if the mouse button has been clicked (i.e., the button was held down and then released). PF's own word, `?button`, returns *true* when the mouse button is down. `Clear_box` determines if the mouse button has been clicked within the clear rect. `Draw_segment` draws a line from the last place the mouse was to the place it is now.

`Mouse` is where the actual drawing happens. While it is true that `mouse` could be more completely factored, I chose to leave it as is to make it, perhaps, more readable (no flames, please!) Structurally it is:

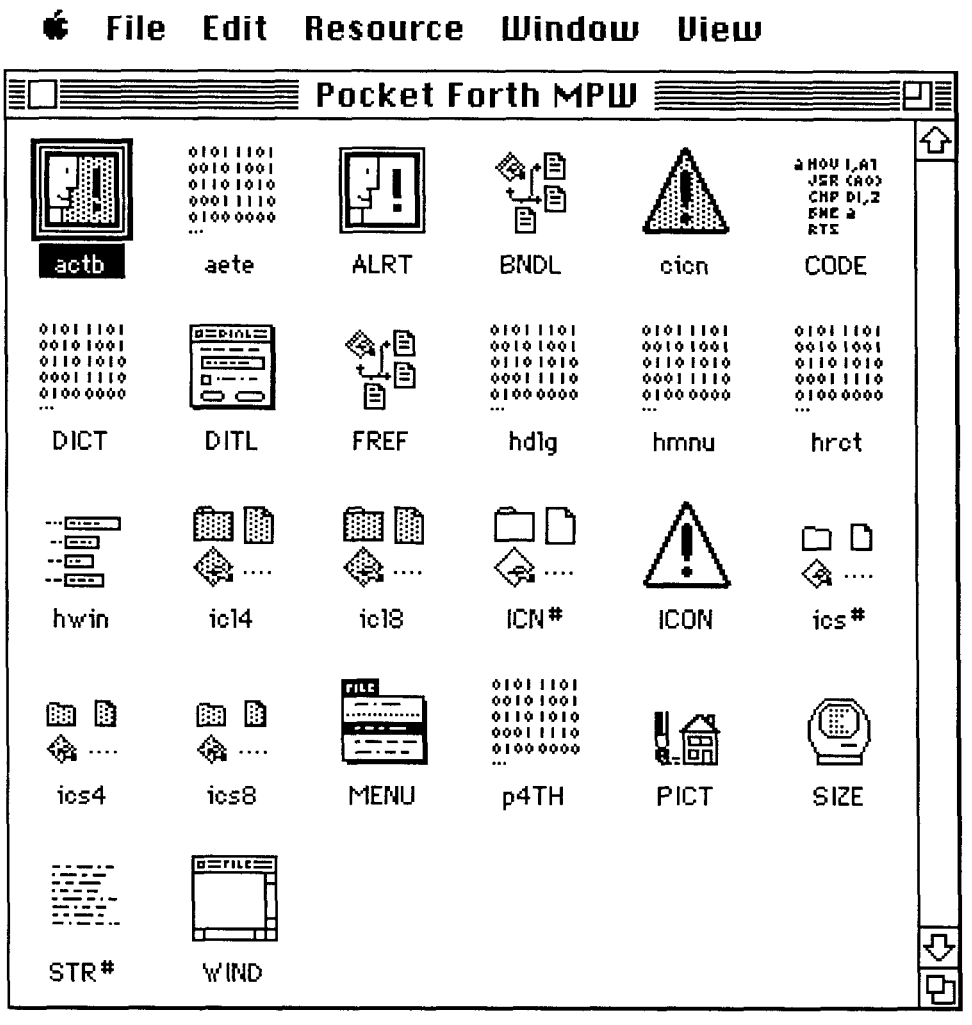

Figure Three. Pocket Forth as it looks to ResEdit

```
: mouse ( -- )   ( mouse button handler )
  -- get where we are now (where the button was first held down)
     and save it (lines 83-85)
  -- while the button is still down (lines 86-98)
     -- get the mouse position
-- if in the drawing rect, draw the segment from the last
     position
-- if in the clear rect, clear the screen
-- ?terminal drop  - quickly check for an event
;
```

`?Terminal drop` in the inner loop processes events while the mouse button is down. Otherwise, PF would execute the loop until the mouse button was released, without allowing other events to be processed. MouseDraw is now 75% complete. The code is written and runs as we want it to run. All that is left is to transform a copy of PF into the application the MouseDraw code expects.

The resources

We need to change several things in order to complete MouseDraw as a standalone application. First, we will change its creator and Finder icon. Second, we will change the About... dialog to our own. Third, we will change the application menus to reflect what MouseDraw expects. Last, we will change the application window name. To accomplish this we will use ResEdit 2.1, Apple's resource editor.

If we open a copy of the PF application with ResEdit, we see a list of resources (Figure Three). Double-clicking on one of these icons will open an editor for that resource. What the editor looks like depends on the resource type: For any of the icons, which can be edited as a group by opening the ICN# resource, a drawing editor is opened. For dialogs (type DITL), an editor suitable to placing text and buttons is opened. If the resource is a user-defined resource, or pure binary data, a simple hex editor is used.

To change the Finder creator (a four-character identifier used to associate files with applications) choose Get File/Folder Info... from ResEdit's File menu, then select the copy of PF we are working with. Simply enter the new creator as any four characters. This is done so that, once we change the icon, we do not change the icon for the PF application itself. Also,

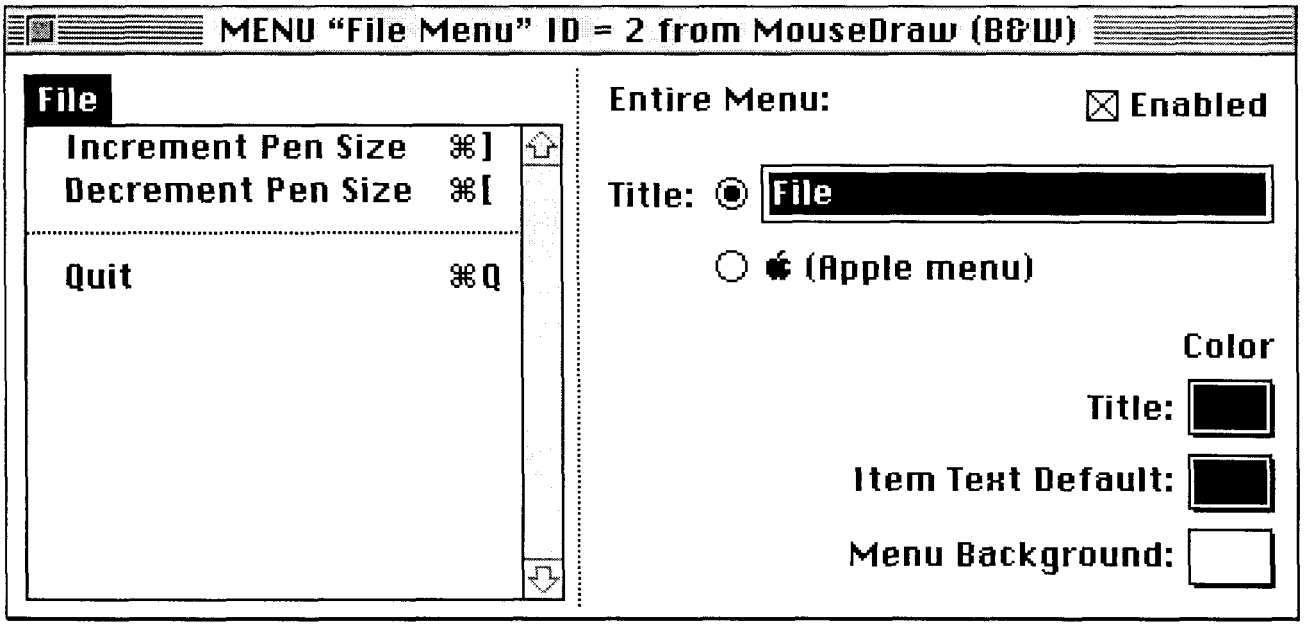

Figure Four. MouseDraw's File Menu.

check that the BNDL flag is selected. Then open the copy of PF and double-click the ICN# icon. We want to edit ICN# 128. Double-clicking this displays a drawing window where we can create our new icon. ICL8 and ICL4 are 8-bit and 4-bit color icons. Make them the same as the ICN# icon. Lastly, open the BNDL resource number 128 and make the signature the same as the creator above.

We need to change the About... dialog. This dialog is presented when the user selects the About... item on the Apple menu. Open DITL, then open number 257. Use the editor to create a new dialog that says something intelligent about the application. At least one element of this must be *enabled* so the user can dismiss the dialog; I usually add an Okay button, along with some static text. Next, open ALRT resource 257 and resize it to match the new dialog.

Lastly, open the MENU resource. Change resource 1 to say "About MouseDraw..." Select resource 3 and click the "enabled" flag on each Edit menu item to disable it. Finally, select resource 2 and make it look like the one in Figure Four. There is now only one thing left to change.

Open WIND resource 128. This is the application's default window. Select Set 'WIND' Characteristics... from the WIND menu, and change the window name. If we so desire, we could also change the default size here, instead of setting it with `wsize` after the application has launched. Save all the changes and quit ResEdit.

The last step is to load our source into the modified application. Launch the modified PF application, which by now should be called MouseDraw, and load the source via the word `open`. Once loaded, enter `save bye` to save the dictionary and exit. The application is now finished. To see the new Finder icon, rebuild the desktop file or use a utility like Michael S. Engber's "Save A BNDL."

Postscript

There are only a few Forths for the Macintosh. Of these, Pocket Forth is the smallest that can be used to build applications. It can access all of the Macintosh Toolbox. It doesn't need much memory—often less than 128K—and it doesn't take mountains of disk space. It is fast enough for most applications, and has floating-point numbers, should you wish to use it for scientific pursuits. The process outlined above may be repeated for any program desired, thereby making Pocket Forth the ideal tool for small applications.

Yet Another Forth Structures Package

Many ways to add a feature like C's `struct` or Pascal's `RECORD` have been presented and discussed in the Forth community. One of them was posted on the USENET newsgroup comp.lang.forth in 1989 (unfortunately I don't remember, by whom; possibly John Hayes), and convinced me with its simplicity, elegance, and power (in a word, with its Forth-ness).

I have used this basic approach ever since, e.g., in the parser generator Gray [ertl97]. It also inspired my approach for an object-oriented Forth extension. The package I present here adds automatic handling of alignments, a bit of syntactic sugar, and optimization of fields with offset 0.

Why explicit structure support?

If we want to use a structure containing several fields, we could simply reserve memory for it, and access the fields using address arithmetic. As an example, consider a structure with the following fields:

a is a float
b is a cell
c is a float

Given the (float-aligned) base address of the structure, we get the address of the field:

a without doing anything further
b with `float+`
c with `float+ cell+ faligned`

It is easy to see that this can become quite tiring.

Moreover, it is not very readable, because seeing a `cell+` tells us neither which kind of structure is accessed nor what field is accessed; we have to somehow infer the kind of structure, and then look in the documentation to learn which field of that structure corresponds to that offset.

Finally, this kind of address arithmetic also causes maintenance troubles: If you add or delete a field somewhere in the middle of the structure, you have to find and change all computations for the fields afterwards.[1]

So, instead of using `cell+` and friends directly, how about storing the offsets in constants:

```
0 constant a-offset
0 float+ constant b-offset
0 float+ cell+ faligned c-offset
```

Now we can get the address of field `x` with `x-offset +`.

1 You may ask why you would want to add a field in the middle. One reason is if you have derived extended structures from a base structure, and want to add a field to the base structure—the new field would appear in the middle of the extended structures.

This is much better in all respects. Of course, you still have to change all later offset definitions if you add a field. You can fix this by declaring the offsets in the following way:

```
0 constant a-offset
a-offset float+ constant b-offset
b-offset cell+ faligned constant c-offset
```

Since we always use the offsets with `+`, using a defining word `cfield` that includes the `+` in the action of the defined word offers itself:

```
: cfield ( n "name" -- )
    create ,
does> ( name execution: addr1 -- addr2 )
    @ + ;

0 cfield a
0 a float+ cfield b
0 b cell+ faligned cfield c
```

Instead of `x-offset +`, we now simply write `x`.

The structure field words now can be used quite nicely. However, their definition is still a bit cumbersome: We have to repeat the name, the information about size and alignment is distributed before and after the field definitions, etc. The structure package presented here addresses these problems.

Usage

You can define a structure for a (data-less) linked list with:

```
struct
    cell% field list-next
end-struct list%
```

With the address of the list node on the stack, you can compute the address of the field that contains the address of the next node with `list-next`. E.g., you can determine the length of a list with:

```
: list-length ( list -- n )
\ "list" is a pointer to the
\      first element of a linked list
\ "n" is the length of the list
    0 begin ( list1 n1 )
        over
    while ( list1 n1 )
        1+ swap list-next @ swap
    repeat
    nip ;
```

Anton Ertl • Vienna, Austria
anton@mips.complang.tuwien.ac.at

Anton Ertl uses Forth as a subject of research focusing on native code compilation and as a tool for non-Forth research. He is a co-author of Gforth, which aims to be both fast and portable.

You can reserve memory for a list node in the dictionary with `list% %allot`, which leaves the address of the list node on the stack. For the equivalent allocation on the heap, you can use `list% %alloc` (or, for an `allocate`-like stack effect—i.e., with ior—use `list% %allocate`).

Note that, in ANS Forth, the body of a `created` word is `aligned` but not necessarily `faligned`; therefore, if you do a

```
create name foo% %allot
```

the memory allotted for `foo%` is guaranteed to start at the body of *name* only if `foo%` contains only character, cell, and double fields.

You can also include a structure `foo%` as a field of another structure, with:

```
struct
...
    foo% field ...
...
end-struct ...
```

Instead of starting with an empty structure, you can also extend an existing structure.[2] E.g., a plain linked list without data, as defined above, is hardly useful. You can extend it to a linked list of integers, like this:

```
list%
    cell% field intlist-int
end-struct intlist%
```

`intlist%` is a structure with two fields: `list-next` and `intlist-int`.

You can specify an array type containing *n* elements of type `foo%` like this:

```
foo% n *
```

You can use this array type in any place where you can use a normal type, e.g., when defining a `field`, or with `%allot`.

The first field is at the base address of a structure, and the word for this field (e.g., `list-next`) actually does not change the address on the stack. You may be tempted to leave it away in the interest of run-time and space efficiency. This is not necessary, because the structure package optimizes this case, and compiling such words does not generate any code. So, in the interest of readability and maintainability, you should include the word for the field when accessing the field.

Naming convention

The field names that come to (my) mind are often quite generic and, if used, would cause frequent name clashes. E.g., many structures probably contain a `counter` field. The structure names that come to (my) mind are often also the logical choice for the names of words that create such a structure.

Therefore, I have adopted the following naming conventions:

- The names of fields are of the form *struct-field*, where *struct* is the basic name of the structure, and *field* is the basic name of the field. You can think of field words as converting the (address of the) structure into the (address of the) field.
- The names of structures are of the form *struct*`%`, where *struct* is the basic name of the structure.

This naming convention does not work that well for fields of extended structures; e.g., the integer list structure has a field `intlist-int`, but has `list-next`, not `intlist-next`.

Implementation

The central idea in the implementation is to pass on the stack, not in some global variable, the data about the structure being built. Everything else falls into place naturally once this design decision is made.

The type description on the stack is of the form *align size*. Keeping the size on the top-of-stack makes dealing with arrays very simple.

`field` is a defining word that uses `create` and `does>`. The body of the field contains the offset of the field, and the normal `does>` action is `@ +` (i.e., add the offset to the address), giving the stack effect `addr1 -- addr2` for a field.

This simple structure is slightly complicated by the optimization for fields with offset 0, which requires a different `does>` part (because we cannot rely on there being something on the stack if such a field is invoked during compilation). Therefore, we put the different `does>` parts in separate words, and decide which one to invoke, based on the offset. For a zero offset, the field is basically a noop; it is immediate, and therefore no code is generated when it is compiled.

Acknowledgments

Marcel Hendrix provided helpful comments on the paper.

Glossary

`%align ( align size -- )`
Align the data space pointer to the alignment `align`.

`%alloc ( size align -- addr )`
Allocate `size` address units with alignment `align`, giving a data block at `addr`; `throws` an ior code if not successful.

`%allocate ( align size -- addr ior )`
Allocate `size` address units with alignment `align`, similar to `allocate`.

`%allot ( align size -- addr )`
Allot `size` address units of data space with alignment `align`; the resulting block of data is found at `addr`.

```
cell% ( -- align size )
char% ( -- align size )
create-field ( align1 offset1 align size
               "name" --  align2 offset2 )
```
`name` execution: `-- addr`
Like `field`, but without the `does>` part.

```
dfloat% ( -- align size )
```

2 This feature is also known as *extended records* [wirth88]. It is the main innovation in the Oberon language; in other words, adding this feature to Modula-2 led Wirth to create a new language, write a new compiler, etc. Adding this feature to Forth just requires a few lines of code.

```
double% ( -- align size )
end-struct ( align size "name" -- )
```
name execution: `-- align size2`
`size2` is `size` aligned with `align`; this ensures that all elements of an array of `name` elements have alignment `align`.

```
field ( align1 offset1 align size
                "name" --  align2 offset2 )
```
name execution: `addr1 -- addr1+offset1`
Create a field `name` with offset `offset1`, and the type given by `size align`. `offset2` is the offset of the next field, and `align2` is the alignment of all fields.

```
float% ( -- align size )
nalign ( addr1 n -- addr2 )
```
`addr2` is the aligned version of `addr1` with respect to the alignment `n`.

```
sfloat% ( -- align size )
struct ( -- align size )
```
An empty structure, used to start a structure definition.

References

[ertl89] M. Anton Ertl. GRAY — ein Generator für rekursiv absteigende Ybersetzer. Praktikum, Institut für Computersprachen, Technische Universität Wien, 1989. In German.

[ertl97] M. Anton Ertl. GRAY — ein Generator für rekursiv absteigende Ybersetzer. In *Forth-Tagung*, Ludwigshafen, 1997. In German.

[wirth88] Niklaus Wirth. "From Modula to Oberon." *Software—Practice and Experience*, vol. 18 no. 7 pp. 661–670, July 1988.

Listing One

Complete text and source code for this article and the objects package from the preceding issue are available at http://www.complang.tuwien.ac.at/forth/objects.zip

```
\ data structures (like C structs)

\ This file is in the public domain. NO WARRANTY.

\ This program uses the following words
\ from CORE :
\ : 1- + swap invert and ; DOES> @ immediate drop Create rot dup , >r
\ r> IF ELSE THEN over chars aligned cells 2* here - allot
\ from CORE-EXT :
\ tuck pick nip
\ from BLOCK-EXT :
\ \
\ from DOUBLE :
\ 2Constant
\ from EXCEPTION :
\ throw
\ from FILE :
\ (
\ from FLOAT :
\ faligned floats
\ from FLOAT-EXT :
\ dfaligned dfloats sfaligned sfloats
\ from MEMORY :
\ allocate

: nalign ( addr1 n -- addr2 )
\ addr2 is the aligned version of addr1 wrt the alignment size n
 1- tuck +  swap invert and ;

: dofield ( -- )
does> ( name execution: addr1 -- addr2 )
    @ + ;
```

```
: dozerofield ( -- )
    immediate
does> ( name execution: -- )
    drop ;

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

: field ( align1 offset1 align size "name" --  align2 offset2 )
    \ name execution: addr1 -- addr2
    2 pick >r \ this ugliness is just for optimizing with dozerofield
    create-field
    r> if \ offset<>0
      dofield
    else
      dozerofield
    then ;

: end-struct ( align size "name" -- )
    over nalign \ pad size to full alignment
    2constant ;

\ an empty struct
1 chars 0 end-struct struct

\ type descriptors, all ( -- align size )
1 aligned   1 cells   2constant cell%
1 chars     1 chars   2constant char%
1 faligned  1 floats  2constant float%
1 dfaligned 1 dfloats 2constant dfloat%
1 sfaligned 1 sfloats 2constant sfloat%
cell% 2*              2constant double%

\ memory allocation words
: %align ( align size -- )
    drop here swap nalign here - allot ;

: %allot ( align size -- addr )
    tuck %align
    here swap allot ;

: %allocate ( align size -- addr ior )
    nip allocate ;

: %alloc ( size align -- addr )
    %allocate throw ;
```

A gentle introduction to defining words...

Approaching CREATE DOES>

Forth has become an indispensable part of my embedded-development toolbox. It has taken a long time to reach this point, because I was never sanctioned by employers to even consider using Forth. Therefore, I had to learn it piecemeal as the years went by, through books and magazine articles and by toying around with public-domain Forth systems. Even so, I always remained on the periphery of applying it to problems, because I was stuck in the learning process and wasn't yet understanding the more advanced concepts.

What attracted me to Forth, and kept my interest, were the various hints I perceived about its usefulness in embedded systems, such as the ability to selectively compile parts of the Forth system into an application, and the technique of remote interactive target compiling. Still other, more nebulous ideas held my curiosity: that a Forth system emerges from Forth source; and that, to understand Forth, one needs to read the source code for the language itself.

Most programmers who are curious about Forth probably never get past the basic features of the language, such as the stack and word definitions, which, while unusual, don't seem to offer much more than traditional languages. To make the leap in productivity that is suggested about Forth, one has to move beyond ordinary programming usage and into the powerful features that set it apart as a fourth generation language. These features provide the means to easily create macro tools for application problem-solving that would be tedious in third-generation languages such as C. As with any powerful tool, one has to scale the learning curve to put it to use. But that price buys time-saving skills that can be applied quickly to practical problems.

I think I am still close enough to the learning process to explain it to others, so in this and in following articles, I will try to explain some of the advanced techniques of Forth that are useful for embedded systems development, and I will try to do it in a way novices can understand. The source code will be short and digestible, and will demonstrate practical applications of Forth in embedded systems. My explanations may gloss over some of the finer details but will, hopefully, get you up and running quickly.

Figure One

```
\ definition time...
0 VARIABLE BYTECOUNT     \ define a variable

\ execution time...
BYTECOUNT @              \ fetch the variable value

20 BYTECOUNT !           \ store a new value in the variable
```

A gentle introduction to CREATE DOES>

The first technique we need to cover is the CREATE DOES> construct for creating defining words, which puzzled me from afar well into my Forth learning phase. I had read a few textbook explanations, but was unable to grasp what it would be used for. Discussion of "parent" and "child" words quickly lost me. At that point, I had not encountered a need for it and, thus, could not appreciate what a powerful tool it is.

I finally stumbled into a clearing and understood CREATE DOES> when I was writing a target compiler for an embedded micro. As I scratched out the requirements for the compiler, I unwound Forth a little more each time I looked for a way to achieve something. Forth always seemed to have the exact answers and, eventually, I came to CREATE DOES>. I probably did it the hard way, but the process unveiled Forth to me in much the same way one has to learn a math concept: by working from the most rudimentary assumptions to an overall method for solving a problem. I will try to help you jump past the beginner's murk and start using this tool right away.

Despite its power and mystery, CREATE DOES> is quite simple to understand and to apply. If you are a C programmer, consider it as something like a C function that could define new C functions that share similar behavior—except it is much easier to do in Forth than in C.

When CREATE DOES> is used in a Forth definition, it confers upon that definition the status *defining word*, which means the word is used to define other Forth words that behave in the manner laid down by CREATE DOES>. VARIABLE is a good example of a defining word; when a VARIABLE word is defined, it is given a value. When the new word is executed, it places its address on the stack, which can be used to fetch or store a value in the variable (see Figure One).

Here is what you do with CREATE and DOES> when you are trying to develop a defining word:

CREATE — Manipulate data to be associated with the new word. In this section of your defining word, you can take data from any source and compile it into the word being defined. You can also execute Forth definitions while you are defining the new words.

DOES> — Define the execution behavior the new word will have. When the defined word (such as BYTECOUNT) is executed, the *parameter field address* (PFA) of the word is placed on the stack. This is, generally, the beginning address of the data laid down in the CREATE section. You can now write Forth code to operate on this data in any way you like.

A practical example

In a recent project, I needed to interface an IBM PC to a serial bus to aid in developing embedded code for a communications device that is similar to a modem. Regrettably, the embedded micro was coded in assembler; but Forth provides

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

Figure Two

```
: MSG           \ MSG is the name of the defining word. When
                \ it is used to define a new Forth word, it...

        CREATE  \ ...creates a new word that contains the set of bytes that was
                \ on the stack when the word was defined...

        DOES>   \ ...and when the new word is executed,
                \ it transmits the byte set out the serial port.
;
```

Figure Three-a

```
80 A8 E8 0 MSG MY_MESSAGE       msg size = 5 bytes      ok
```

Figure Three-b

```
MY_MESSAGE 80 A8 E8 0 F0 ok     (F0 is the checksum for the four bytes)
```

Figure Four

```
80 54 0 5C 0 BE 0 0 55 60 79 0 B7 0 0 B8 0 0 MSG MESSAGE2
80 C0 11 C2 10 16 5B 23 6F 21 6F 22 11 34 5B 24 64 24 6E 24 MSG MESSAGE3
```

a great tool for testing the interface in ways the Windows bus-analysis software we had purchased could not. What I needed was a way to easily create variable byte messages to transmit over the serial line, to be filtered by the embedded micro. I envisioned a defining word that would take a string of bytes for input and associate them with a message name. During execution, I wanted the message word to transmit the bytes onto the bus. (See Figure Two.)

To create a message, one simply types a string of bytes followed by MSG and the desired name of the new word (this can also be done in a source code file or block).

```
80 A8 E8 0 MSG MY_MESSAGE
```

The four bytes before MSG end up on the stack; when MSG executes, it compiles those bytes into the new word. This is what is happening during the CREATE phase of the above MSG definition. When MY_MESSAGE itself is executed, it transmits the four bytes through the serial port. This behavior is set up by the DOES> phase of the MSG definition.

Some other features needed in MSG were a checksum byte to be appended to the byte string, and an indication of the number of bytes entered in a new definition. We are free to use Forth in the CREATE section of MSG however we like, to set up the data for the new word when it executes. Thus, I added code to sum each byte for a checksum, and to print the byte count to the terminal when a new word is defined. In the DOES> section, I provided a simple loop to pull each byte from the data section of the new word and transmit it serially, as well as printing it on the screen.

So, when MY_MESSAGE is defined, the number of bytes entered plus the checksum are displayed (Figure Three-a). MY_MESSAGE in action is demonstrated in Figure Three-b.

Messages can be any length. More examples are given in Figure Four.

Listing One [page 20] is a heavily commented listing for the MSG definition. I've tried to explain every part of the definition in detail, so the novice will not be mystified by the housekeeping actions needed to manipulate the stack data. When one is new to Forth, constructs such as my DO loops seem hopelessly confused, but experience reveals similar, efficient usage in many Forth definitions. The example DO loops use an address on the stack as a pointer to work through data, in the same way that a language like C would use pointers:

```
for (i = 0; i = bytecount; i++)
        *byteptr++ = *data++;   \\ etc.
etc.
```

One difference is that there are no pointer variables in my Forth loops—the pointer is the address on the stack, which is DUPed on each pass and discarded when finished.

A defining word such as MSG represents object-oriented programming at a rudimentary level: It has data (the stack bytes) and functional behavior (defined by DOES>). By striving to use stack items instead of variables, we "hide" the data and functions from the outside world. And the creation of new MSG words with similar behavior but different data attributes is akin to inheritance.

These examples expose another reason why Forth is better than C for embedded systems testing: once the message-defining word was finished, there was no need to create a *main* loop or other C function to figure out what to do with the message functions, because an elegant and extensible user interface already exists in the form of the Forth interpreter. Using the interpreter with the message words allowed me to communicate over the serial bus with devices, in real-time and in any conceivable variation. Contrast this with having to edit, compile, link, and run (oops! error—repeat *n* times) C source code every time one wants to try a new idea with a device. In fact, many Forth practitioners use a host Forth to talk to an embedded micro or remote device—I will try to explore this useful technique in a later article.

Figure Six shows MSG with the comments removed.

Figure Five

```
0 VARIABLE CHKSUM

: MSG CREATE 0 CHKSUM !
        DEPTH

        DUP 1 + BASE @ >R DECIMAL ." msg size = " . ." bytes" SPACE
        R> BASE ! CR

        DUP ,
        0 DO DUP CHKSUM C@ + CHKSUM C! C, LOOP
        CHKSUM C@ FF XOR 1 + C,

        DOES> DUP @ DUP ROT + 2 + DUP >R SWAP
                0 DO 1 - DUP C@ TX. LOOP DROP R> C@ TX. ;
```

Figure Six

```
: MSET CREATE DEPTH DUP , 0 DO , LOOP

      DOES> DUP @ DUP 2 * ROT + SWAP
                0 DO DUP @ EXECUTE DLY CR 2 - LOOP DROP ;
```

An improvement to the serial message example

Once MSG had been defined, a need to transmit groups of messages became apparent. I devised a new word, similar to MSG but called MSET, which could take a set of previously defined MSG words and collect them under a single word name that could then transmit the messages in a burst. A timing delay inserted between each message would provide a convenient means to vary the serial bus loading. Thus, bus traffic could be simulated easily by using MSET words within DO loops, or in other definitions that respond to data on the bus by transmitting message bursts.

Given the MSG words MY_MESSAGE, MESSAGE1, and MESSAGE2, the problem is to group them using another defining word, such as MSET. Forth provides the word ' (pronounced "tick"), which takes the address of the next word in the input stream and places it on the stack. Using DEPTH the same way we did in MSG, we can take the addresses of a group of messages and compile them into an MSET word, and then, during the execution phase (DOES>), take each compiled address and pass it to another convenient Forth word, EXECUTE.

Now, to create a set of messages using MSET, one enters ' before each MSG word in an MSET definition:

```
' MY_MESSAGE ' MESSAGE1 ' MESSAGE1
MSET SET1
```

Now we have a word called SET1 that, when executed, will execute each word in sequence, with a variable delay between each message.

Figure Six shows the uncommented source for MSET. Listing Two [page 21] is the unabridged version.

The wrap-up

The code for this article was developed in hForth, an ANS Forth for the 8086 written by Dr. Wonyon Koh of South Korea. A nice thing about hForth is that the assembly code for the system has high-level Forth definitions inserted as comments in the source, which provide good examples of Forth programming technique. This system is also a good learning tool for understanding how Forth is constructed, because it is derived from eForth, a model for porting Forth to any processor.

In my testing setup using the message words, I have my editor running under Windows and hForth running as a DOS application. To modify and test my source text, I task switch (using Alt-Tab in Windows) between hForth and the editor. hForth also has a logging facility, so one can capture interactive sessions to disk for later cleanup with an editor. To access the serial port, I used the serial I/O routines provided with hForth.

In the next article, I will use CREATE DOES> with some other Forth tools to demonstrate how easy it is to create a custom macro language using Forth. Using this as a basis, we can construct a target compiler system for an embedded microprocessor or remote computer.

References

W. Koh, "hForth: a Small, Portable, ANS Forth," *Forth Dimensions*, Volume XVIII, Number 2.

L. Morgenstern, "Working With CREATE ... DOES>," *Forth Dimensions*, Volume XIV, Number 1.

Dave Taliaferro originally trained as a biologist but, to make a living, he works as a software engineer in real-time embedded systems development. He first encountered Forth while working with industrial robotics, and afterwards was never able to shake his attraction to this strange and beautiful language. When not learning Forth he is pondering history and genealogy.
http://www.geocities.com/Heartland/Plains/6080/talindex.html

Listing One

```
0 VARIABLE CHKSUM       \ define a checksum variable - one could probably
                        \ avoid this, if clever enough

: MSG   CREATE          \ create a new Forth word name

        0 CHKSUM !      \ clear the checksum variable
        DEPTH DUP       \ get the number of bytes on the stack and make a copy

        1 +                                 \ add 1 for the additional checksum byte
        BASE @ >R                           \ save the current base
        DECIMAL                             \ I prefer to read the count in decimal
        ." msg size = " . ." bytes" SPACE \ print the count to the screen
        R> BASE ! CR                        \ restore the base


        \ compile the set of bytes into the new word
        DUP                                 \ make another copy of the byte count
        ,                                   \ compile it into the word

        \ Loop through the stack and compile each byte into the word while
        \ adding it to the checksum. Notice that we have still one more copy
        \ of the count to provide to the DO loop. Each pass DUPs the byte so
        \ it can be added to the checksum.

        0 DO DUP CHKSUM C@ + CHKSUM C! C, LOOP   \ C, compiles bytes


        \ after the loop is done the checksum is 2's complemented and compiled into the word
        CHKSUM C@ FF XOR 1 + C,


        DOES>   \ Instill the following behavior in the new word...

                \ Yikes! The bytes were compiled in reverse order since the
                \ last byte entered becomes the first byte on the stack.
                \ We can take care of this by transmitting them in reverse and
                \ then appending the checksum.

                \ The first thing the new word does during execution is to
                \ place the address of the byte string on the stack.

                DUP         \ Make a copy of the parameter field address.
                @           \ Fetch the count that was compiled into the new
                            \ word during CREATE and is pointed to by the pfa.

                DUP         \ We need a second copy of the count.

                \ The pfa address is added to the count to point to the last
                \ byte, which is the checksum.
                                    \ copy and save it to the return stack
                ROT + 2 + DUP >R SWAP   \ so it can be used to transmit the checksum

                \ On entering DO we have the pfa (which now points to the last
                \ byte) and a count copy for use by the DO loop.

                0 DO 1 - DUP C@ TX. LOOP  \ Subtract 1 from the pfa on each
                                          \ pass to loop backwards though the bytes.

                DROP                \ We have to discard the last DUP'd address.

                R> C@ TX.           \ finally, pull the checksum address from
                                    \ the return stack and transmit
                ;
```

Listing Two

```
( a an -- )
: MSET CREATE              \ Create a new word name.

       DEPTH DUP           \ Get count of stack objects and make a copy.
      ,                    \ The first copy is compiled into the new word to be
                           \ used during the DOES> phase

      0 DO , LOOP          \ The second copy is used for a loop counter
                           \ to compile each address into the new word.

      DOES>

      DUP @                \ DUP pfa provided by DOES> and fetch the DEPTH count
                           \ compiled during CREATE.

      DUP                  \ Make another copy of the DEPTH count.

      2 * ROT + SWAP       \ The addresses are in reverse order so we have to walk backwards
                           \ through the compiled addresses. Multiply the count by 2 and add to
                           \ the pfa address to get the first MSG address.


     \ using the depth count loop backward through the addresses and EXECUTE
     \ them. DLY is a generic timing routine.

       0 DO DUP @ EXECUTE DLY CR 2 - LOOP

      DROP ;               \ using DUP in a loop for address incrementing leaves
                           \ an extra address on the stack
```

Lookup Tables

> "Personally, I consider the case statement an elegant solution to a misguided problem: attempting an algorithmic expression of what is more aptly described in a decision table."
>
> —Leo Brodie, *Thinking Forth*

Introduction

If you think this article is going to start a new discussion about an old controversy, you're dead wrong. Instead, we will present a new way to solve your problems by using a simple concept, called *lookup tables*.

When you examine a few random Forth programs, you will find that this technique is hardly used, apart from the odd weekdays table. That is a shame, because programs using lookup tables are easier to design, easier to debug, and easier to maintain. Furthermore, they usually are smaller and run faster. We will provide a few examples to make our point.

Lookup tables allow you to build fast, small, and easy-to-maintain applications.

One possible reason for their relative scarcity is that lookup tables can be hard to implement in Forth—especially when strings are concerned. In this article, we will also offer some possible solutions to that problem.

Some might argue that OOF or other non-standard extensions offer even better ways to handle these kind of problems. However, ANS Forth is neither object oriented nor is there a Structure Extension (yet), and we consider that discussion to be outside the scope of this article.

What are lookup tables?

Lookup tables are just groups of objects with common characteristics. Using them means one has to consider the objects that are used inside a program and which characteristics they share (and do not share).

An adventure game is a good example. Every room has its own description and exits. Some exits may be hidden until a certain condition is met. In a room, there are objects. All objects have a description. Some may be moved and some may not. Some may perform an action when certain conditions are not met.

This is a perfect example of an application that should use lookup tables. Any other implementation will certainly be clumsy, hard to debug, and impossible to maintain. An example of a adventure game using lookup tables can be found at http://www.IAEhv.nl/users/mhx/adventur.frt. This adventure game was originally designed for 4tH[1] and was skillfully converted to ANS Forth by Marcel Hendrix.

Why lookup tables?

The use of lookup tables is not limited to a single language or a single set of problems. We have even successfully implemented lookup tables in Excel spreadsheets. This significantly reduced the number of manual manipulations and checks of large sets of data.

The data in one sheet is confronted with a lookup table, and automatically produces a "not available" error when an unknown value is encountered. By searching the appropriate category in another lookup table, and subsequently sorting the entire sheet, subtotals can be produced easily. Before that, all checking and sorting was done manually and was prone to error.

But lookup tables can be of use in other situations, too. The Z80 processor is not a high-speed processor, not even by mid-1980s standards. So when Steve Townsend of PSION designed the engine for the Checkered Flag game on the Sinclair Spectrum, he used lookup tables to link gears, speed, and engine revolutions, rather than set formulae. Each alteration of the controls would have made recalculation necessary, which is a very costly operation on a 3.5 MHz processor without any floating-point hardware.[2]

But if you really want to realize how powerful lookup tables are, remember that the file system you're working with is nothing more than a sophisticated set of lookup tables.

Lookup tables are very flexible and can be implemented in various ways. You can even apply any design method for relational databases, if you need to. By building your own search routine, you can define the way you want to access the lookup tables, e.g., by comparing strings or by finding the closest approximation to a certain value. The only limit is your own imagination.

Case study I: The error handler

One of the co-authors of this article, Hans Bezemer, was approached by a friend with an unusual problem. He had been hired by a company to find the source of some error messages coming from one of their most valued applications.

The company that originally built the application had long gone bankrupt. The system administrator had examined the main source of the application, and had been unable to find these messages. After a short while, his friend had been able to trace the messages back to a library, and that seemed to be the end of it.

Then they had offered him to do a follow-up and design some scheme that would prevent this kind of problem. But with whatever kind of documentation scheme he had come up with, it still failed the specification.

1 *Forth Dimensions*, XVIII.3

2 *Sinclair User*, June 1984

Hans Bezemer • hbezemer@solair1.inter.NL.net
Den Haag, The Netherlands

Benjamin Hoyt • benhoyt@clear.net.nz
Hastings, New Zealand

Together, they worked out a scheme that was finally accepted. Instead of giving each programmer the liberty to handle an error in his own way, a set of centrally maintained tables was designed. Every error message had to be channelled through an error-handler, which had the following definition:

```
error-handler      ( c-addr u n1 n2 n3 -- )
\ c-addr u  additional information
\ n1        routine number
\ n2        error number
\ n3        severity
```

The routine number was an index to a centrally maintained table, with this format:

Routine number	(CELL)
Routine name	(STRING)

The error number was an index to another centrally maintained table, with this format:

Error number	(CELL)
Message	(STRING)

The severity indicated how serious an error was. It had one of five different values:

Fatal	(Abort program)
Error	(Continue, but output is questionable)
Warning	(Attention, will try to recover)
Info	(Just issuing some user information)
Debug	(Debugging information)

Of course, numbers have little mnemonic value, so CONSTANTs were added to minimize the chance of human error, e.g.:

```
0      CONSTANT    S_DEBUG
1      CONSTANT    S_INFO
2      CONSTANT    S_WARN
3      CONSTANT    S_ERROR
4      CONSTANT    S_FATAL

0      CONSTANT    E_SOUTOFRANGE
1      CONSTANT    E_EOUTOFRANGE
2      CONSTANT    E_ROUTOFRANGE
3      CONSTANT    E_NODATA
4      CONSTANT    E_ENDOFILE
( etc.)

0      CONSTANT    R_DATAENTRY
1      CONSTANT    R_PROCESS
( etc.)
```

The use of the error-handler was mandatory, although a string with additional information was allowed. Figure One shows a typical use of the error-handler.

The error-handler did several things. First, it checked the validity of the error, routine, and severity values. Second, it would match the severity level against the message level. If the severity level was equal to or greater than the message level, a message would be issued. Third, it would match the severity level against the abort level. If the severity level was equal to or greater than the abort level, the program would be terminated.

Every software developer who wanted to do business with this company had to comply with this scheme from that day on. It proved to be so simple that most quality assurance could be performed by their own system administrator (we do have to confess that Forth was not the language of choice in that particular environment).

Note that the program returns a dummy value. The reason for that is twofold. First, the original C compiler issued a warning when it was omitted. We don't like warnings, since you never know whether it indicates a real error. Second, an ambiguous condition would exist if some smart programmer found a way to correct the error

Figure One. Typical use of centralized error handler

```
:     process                  ( c-addr u -- n)
      over over                \ duplicate filename
      file-status 0=           \ check file status
      if                       \ if ok; process the data
            drop drop          \ discard filename
            S" None" R_PROCESS E_DATAOK S_INFO error-handler
            ( other code)
      else                     \ if not ok; issue error
            R_PROCESS E_NODATA S_FATAL error-handler
            -1                 \ return dummy value
      then
;
```

Figure Two. Lookup-based decompiler

```
-1 constant EOT                        \ end of table delimiter

( search table for x, if found return corresp. value and true flag)
: search-table                  ( x table -- value true | x false )
begin   dup @ EOT =                    \ is it end of table?
      if      drop false  exit         \ no match found
      then    2dup @ <>                \ compare x with value in table
while   [ 2 cells ] literal + \ move to next table entry
repeat  nip  cell+ @  true ;  \ fetch corresponding value
```

and changed the severity from "fatal" to "error." In Forth, it could cause a stack underflow or, worse, introduce a hard-to-find bug.

Case study II: The For32 decompiler

The other co-author, Benjamin Hoyt, recently implemented a Forth decompiler for his For32 system. He first thought of implementing the main engine with one large CASE statement. Then it clicked that a lookup table implementation could have its advantages, and he decided to give it a whack. He came up with a simple lookup table and, to his surprise, it worked the first time!

The table he used is basically a two-dimensional array with the "special case" execution tokens in the first field—e.g., (LIT), (S"), (TO), and many, many more—and the decompiling words in the second. To clue you in a bit, Figure Two provides the definition.

Very simple indeed, as you can see. Of course, every lookup table needs its own search routine. And if you don't want to make one yourself, we will give a general definition later on. Remember that on many (ANS compliant) systems, CASE is not even available. If you have to code CASE yourself, take our advice and make your own search routine. That is a whole lot simpler than developing an entire CASE suite.

Apart from that, every single OF ... ENDOF pair amounts to at least a literal, a compare, and two jumps. On some systems, this can add up to 40 bytes per OF ... ENDOF pair. A similar entry in a lookup table needs only eight bytes. For instance, the difference between the For32 decompiler using CASE and the one using the lookup table amounts to 1.5 Kb.

Another good reason not to use CASE is that it often won't do what you want. CASE only compares integers. If you are not convinced yet, note that lookup tables are usually faster.

Some C compilers (like XL C on the RS/6000) implement the *select( )* statement with a whole slew of jump and compare instructions. This is rather time-consuming, especially with a reasonably sized list of items. With a lookup table, the search is done in a confined area of execution, and a definite speed increase will be noticed.

The catch

The subtle elegance of lookup tables may be clear to you now, but what's the catch? Why are so few Forth programmers using it? A good question, because there are, in fact, one or two hitches you may run across.

For instance, take the string-comparing example mentioned before. Let's say you are coding a macro-command processor. You decide to implement it with a lookup table. You've coded your string-compare lookup routine, then you build your table using ,". This word isn't part of the ANS Forth standard, but is available in many Forth systems.

```
create command-table   ( -- table )
      ," display"        ' do-display ,
      ," end"            ' do-end ,
      ," save"           ' do-save ,
      ," load"           ' do-load ,
      EOT ,
```

But it dawns on you that you were too hasty. The strings aren't of equal length, you have alignment problems, and the whole thing suffers from a complicated and slow search routine. There are a few solutions to this problem, the simplest of which involves using fixed-length strings like this:

```
create command-table   ( -- table )
      ," display"  ' do-display ,
      ," end    "  ' do-end ,
      ," save   "  ' do-save ,
      ," load   "  ' do-load ,
      EOT ,
```

This simplifies and speeds things up, and may work in some situations. But what about the space wasted by long/short string combinations? You can get around this if you define the strings first, retrieve their addresses, and compile them at the appropriate field in the table. But that can get pretty ugly:

```
: push-address
      c" load"
      c" save"
      c" end"
      c" display"
;

push-address

create command-table   ( -- table )
      ,      ' do-display ,
      ,      ' do-end ,
      ,      ' do-save ,
      ,      ' do-load ,
      EOT ,
```

Another solution is to write a definition called M". It parses a string and compiles it into the dictionary, while leaving its address on the stack. Then you'd create your lookup table by "comma-ing" all these addresses into it. (See Figure Three.)

This may work if you have only a few entries to compile, but it gets hard to maintain when you have tens of entries. The problem is that ," compiles strings on the spot, S" only temporarily stores a string when in interpretation mode, and C" lacks interpretation semantics all together. You might give it another try and get something like this:

```
: display-s        c" display" ;
: end-s            c" end" ;
: save-s           c" save" ;
: load-s           c" load" ;

create command-table   ( -- table )
      display-s    ,      ' do-display ,
      end-s        ,      ' do-end ,
      save-s       ,      ' do-save ,
      load-s       ,      ' do-load ,
      EOT ,
```

Okay, this works, too, and it can be maintained with a little trouble, but it certainly doesn't feel good with all those

Figure Three. The M" approach

```
\ string compiling suite )

( c-addr u dest -- )
: place 2dup 2>r  char+ swap chars move  2r> c! ;

( c-addr u -- )
: name, here  over 1+ chars allot  place ;

( "ccc<quote>" -- c-addr )
: m" align here  [ char] " parse  name, ;

( table of macro commands )
m" display"
m" end"
m" save"
m" load"

( the addresses are all on the stack now, in reverse order)
create command-table  ( -- table )
      ,     ' do-load ,
      ,     ' do-save ,
      ,     ' do-end ,
      ,     ' do-display ,
      EOT ,

( search table for string c-addr u)
( give xt true if found else c-addr u false)

: string-search  ( c-addr u table -- xt true | c-addr u false )
begin   dup @ EOT =                      \ is it end of table?
if      drop false  exit                 \ no match found
then dup 2over rot @ count compare  \ compare with c-addr u
while   [ 2 cells ] literal +       \ move to next table entry
repeat  nip nip  cell+ @  true ;    \ fetch xt from column 2
```

wasted headers. Let's see if we can change that.

Solutions and implementations

The 4tH compiler is a very different Forth compiler. Some argue that it is not a Forth compiler at all. We consider this an academic discussion, in this context.

What does matter is that 4tH provides an easy way to define lookup tables, having different segments for strings and integers, and no distinction between compilation and interpretation semantics. There are a number of ways to implement some of this functionality in ANS Forth.

You can try to implement the LMI Forth solution. This compiler features a word called ", which roughly behaves like C" with interpretation semantics. When interpreting, it uses a circular buffer. The trouble is that you never know when the system is about to wrap around.

Another work-around is to ALLOT your own string space and compile your strings there. The catch is that your environment is limited by the amount of string space you have allocated. Once you run out of string space, you have to restart the system. This is one of the solutions Wil Baden proposed (Figure Four).

There are several ways to get around the limited string space. Redefining /STRING-SPACE is a possibility. An obvious way is to allocate the string space in dynamic memory, and reallocate it when needed. But reallocation could invalidate all previously compiled addresses, which is definitely not what we want.

Another ingenious way to use dynamic memory comes from Marcel Hendrix. He allocates each individual string in dynamic memory, as demonstrated in the previously mentioned adventure game.

There are many solutions. Use the one that serves you best.

Still, there is the problem of the search routine. For your convenience, we present a generic solution that can be implemented on virtually every Forth system. For the strings, you either have to create your own solution or use one of ours (Figure Five).

We can even go one step further and define a word called TABLE, which CREATEs a lookup table that, when executed, searches itself and returns the required value:

Figure Four. A solution from Wil Baden

```
( Reserve  STRING-SPACE  in data-space. )
2000 CONSTANT /STRING-SPACE
CREATE STRING-SPACE           /STRING-SPACE CHARS ALLOT
VARIABLE NEXT-STRING          0 NEXT-STRING !

( caddr n addr -- )
: PLACE 2DUP 2>R CHAR+ SWAP CHARS MOVE 2R> C! ;

( "ccc<quote>" -- caddr )
: STRING" [CHAR] " PARSE
DUP 1+ NEXT-STRING @ + /STRING-SPACE >
      ABORT" String Space Exhausted. "
      STRING-SPACE NEXT-STRING @ CHARS + >R
            DUP 1+ NEXT-STRING +!
            R@ PLACE
      R>
;

CREATE months

      STRING" January" ,    31 ,
      STRING" February" ,   28 ,
      STRING" March" ,      31 ,
      STRING" April" ,      30 ,
      STRING" May" ,        31 ,
      STRING" June" ,       30 ,
      STRING" July" ,       31 ,
      STRING" August" ,     31 ,
      STRING" September" ,  30 ,
      STRING" October" ,    31 ,
      STRING" November" ,   30 ,
      STRING" December" ,   31 ,

: .Month 1- 2* CELLS months + @ COUNT TYPE SPACE ;
```

```
( create search table called "name")
( when executed, searches its table for x)

( returning the table value and true if)
( found, else x and false )

: table  ( "name" -- )  create
      does>  ( x -- value true | x false )
search-table ;
```

You can implement different versions with different search methods for different kinds of lookup tables. That allows you to create very powerful applications very quickly. Remember, this is one of the privileges you have when you are using Forth!

Epilogue

Have you ever thought about implementing a Forth dictionary, or even a whole Forth system, using lookup tables? We bet it can be done! In fact, the entire 4tH compiler is centered around four different lookup tables.

Lookup tables allow you to build fast, small, and easy-to-maintain applications. In our opinion, this method has not been used extensively in Forth, partly because there were few facilities to support it. We hope we have provided enough material to give you some fresh ideas and to get you started right away.

Benjamin Hoyt is a sixth-form student who loves programming as a hobby. Before he discovered Forth, he experimented with graphics and AdLib programming in 80x86 assembler. Over a year ago, he built his first Forth compiler and has stayed loyal to the language ever since. He currently uses and is working on his own ANS Forth called For32, running under MS-DOS. Another of his major projects at the moment is For16, a small, ANS-compliant Forth compiler running under MS-DOS.

Hans "the Beez" Bezemer has been using Forth and C since the mid-1980s. He is the author of several shareware programs and the freeware 4tH compiler. 4tH is available at ftp.taygeta.com.

Figure Five. A generic solution

```
\ : th cells + ;

0 Constant NULL

create MonthTable
   1 , "  January " , 31 ,
   2 , " February " , 28 ,
   3 , "   March  " , 31 ,
   4 , "   April  " , 30 ,
   5 , "    May   " , 31 ,
   6 , "   June   " , 30 ,
   7 , "   July   " , 31 ,
   8 , "  August  " , 31 ,
   9 , " September" , 30 ,
   10 , "  October " , 31 ,
   11 , " November " , 30 ,
   12 , " December " , 31 ,
   NULL ,

\ Generic table-search routine

\ Parameters:  n1 =  cell value to search
\         a1 =  address of table
\         n2 =  number of fields in table
\         n3 =  number of field to return

\ Returns:  n4 =  value of field
\         f  =  true flag if found

: Search-Table        ( n1 a1 n2 n3 -- n4 f)
   swap >r          ( n1 a1 n3)
   rot rot          ( n3 n1 a1)
   over over           ( n3 n1 a1 n1 a1)
   0             ( n3 n1 a1 n1 a1 n2)
   begin         ( n3 n1 a1 n1 a1 n2)
      swap over     ( n3 n1 a1 n1 n2 a1 n2)
      th         ( n3 n1 a1 n1 n2 a2)
      @ dup      ( n3 n1 a1 n1 n2 n3 n3)
      0> >r      ( n3 n1 a1 n1 n2 n3)
      rot <>        ( n3 n1 a1 n2 f)
      r@ and        ( n3 n1 a1 n2 f)
   while         ( n3 n1 a1 n2)
      r> drop       ( n3 n1 a1 n2)
      r@ +          ( n3 n1 a1 n2+2)
      >r over over     ( n3 n1 a1 n1 a1)
      r>         ( n3 n1 a1 n1 a1 n2+2)
   repeat           ( n3 n1 a1 n2)

   r@ if
      >r rot r>        ( n1 a1 n3 n2)
      + th @        ( n1 n4)
      swap drop        ( n3)
   else
      drop drop drop ( n1)
   then

   r>            ( n f)
   r> drop          ( n f)
;

: Search-Month          ( n --)
   MonthTable 3 2 Search-Table

   if
      .
   else
      drop ." Not Found"
   then cr
;
```

Continued from page 39

BLOAD can be used to send a full block to the slave to be interpreted (loaded)[3].

Normally, when entering characters into the slave input buffer, the characters would be echoed to the host and displayed immediately. With the above-described downloads, the echoed characters are not displayed until a CR is received (function of SBUF.) from the slave, indicating it is finished. This feature was included to ensure that the slave is finished before any additional operations are attempted.

Load initialization (screens 4–7, 230–23)

Different machines can be configured with combinations of load and initialization screens. The load screens (examples 4–7) not only select the various functions to be compiled, but also define three common display modes and set the path to the COMMAND.COM file.

An initialization screen (examples 230–233) defines the INIT word, which configures memory, sets several variables to default values, resets the mouse driver (if available), and enables the serial port.

Memory management includes, first, releasing most of the memory assigned to the xxx.COM program by DOS with SETBLOCK, then allocating a disk buffer (used by the CHANx scheme) and a serial receive buffer with BLKALOC.

Summary

These tools have been used for a number of projects during the past few years. I find it enjoyable to develop a project and never leave the Pygmy environment. Again, I would like to thank Frank Sergeant for Pygmy Forth, which prompted this adventure for me.

References

1. ftp://ftp.taygeta.com
   file "pyg_embl.exe" in the /pub directory
2. http://www.theramp.net/sferics
   File "pyg_embl.exe" is listed in "Misc. Downloads"
3. Richard W. Fergus, "Development Aids for New Micros," *Forth Dimensions* XVIII.3 Sept-Oct 1996

Arcipher — Alleged RC4

```
 1 (Coded from Bruce Schneier, Applied Cryptography, 2nd edition. )

 3 CREATE S-Box  256 CHARS ALLOT

 5 VARIABLE #I   VARIABLE #J

 7     : C+! (n addr -- ) DUP >R  C@ +  255 AND  R> C! ;
 8     (With 8-bit bytes `255 AND` is unnecessary here. )

10     : cexchange                          (addr1 addr2 -- )
11         DUP C@ >R  OVER C@ SWAP C! (addr1) R> SWAP C! ()
12     ;

14     MACRO S[ ] " CHARS S-Box + "

16 : RC4                                (char -- code )
17     1 #I C+!
18     #I C@ S[ ] C@  #J C+!
19     #I C@ S[ ]     #J C@ S[ ]  cexchange
20     #I C@ S[ ] C@  #J C@ S[ ] C@  +
21         255 AND S[ ] C@  XOR           (code)
22 ;

24 : rc4-init                           (key len -- )
25     256 0 DO  I DUP S[ ] C!  LOOP
26     0 #J C!
27     256 0 DO                              (key len)
28         2DUP I SWAP MOD CHARS + C@  I S[ ] C@ +  #J C+!
29         I S[ ]  #J C@ S[ ]  cexchange
30     LOOP                                  2DROP
31     0 #I C!  0 #J C!
32 ;
```

We have an array of 256 bytes, all different. Every time the array is used, it changes—by swapping two bytes.

The changes are controlled by counters *i* and *j*, each initially zero. To get a new *i*, add 1. To get a new *j*, add the array byte at the new *i*.

Swap the array bytes at *i* and *j*. The code is the array byte at the sum of the array bytes at *i* and *j*.

The array is initialized by first setting it to 0 through 255. Then step through it using *i* and *j*, getting the new *j* by adding to it a key and the array byte at *i*, and swapping the array bytes at *i* and *j*.

All additions are modulo 256.

The name "RC4" is trademarked by RSA Data Security, Inc. So anyone who writes his own code has to call it something else. Here it's called "Arcipher."

From Schneier.

So, what's the deal with RC4? It's no longer a trade secret, so presumably anyone can use it...

RC4 is in dozens of commercial cryptography products, including Lotus Notes, Apple Computer's AOCE, and Oracle Secure SQL. It is part of the Cellular Digital Packet Data Specification.

Here is the announcement of the discovery of "Alleged RC4":

Newsgroups:
sci.crypt,alt.security,comp.security.misc,alt.privacy
From: sterndark@netcom.com (David Sterndark)
Subject: RC4 Algorithm revealed.
Date: Wed, 14 Sep 1994 06:35:31 GMT

I am shocked, shocked, I tell you, shocked, to discover

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

Wil Baden is a professional programmer with an interest in Forth. For the source for this article, send e-mail to WilBaden@netcom.com and request Stretching Forth #16: Arcipher—Alleged RC4.

that the cypherpunks have illegaly and criminally revealed a crucial RSA trade secret and harmed the security of America by reverse engineering the RC4 algorithm and publishing it to the world.

Arnold G. Reinhold in his "CipherSaber Manifesto" at <http://ciphersaber.gurus.com> urges use of "CipherSaber-1":

CipherSaber-1 is a simple use of existing technology:

1. The encryption algorithm is RC4 as published in the beginning of Chapter 17 of *Applied Cryptography*, Second Edition, by Bruce Schneier, John Wiley & Sons, New York, 1996. RC4 is on page 397 in the English edition, ISBN 0-471-11709-9.

2. Each encrypted file consists of a ten byte initialization vector followed by the cipher text. A new random ten byte initialization vector should be created each time encryption is performed.

3. The cipher key, the array K(i) in Schneier's notation, consists of the user key, in the form of an Ascii text string, followed by the initialization vector.

The above is all a programmer needs to know in order to write a program that can encipher and decipher CipherSaber-1 files.

The ten-byte initialization vector doesn't have to be fancy. Use any convenient RNG that is primed somehow with the current time and date.

```
35 VARIABLE Randseed
36 : srand  Randseed ! ;
37 : Rand   Randseed @  3141592621 *  1+  DUP Randseed ! ;
38 : choose  Rand UM* NIP ;

40 : Randomize TIME&DATE 12 * + 31 * + 24 * + 60 * + 60 * + srand ;

42 Randomize

44 (So you can duplicate I'm using a constant instead of Randomize. )

46 1 srand

48 ::  10 0 DO
49     S" Forth Dimensions" PAD place
50     PAD COUNT
51     10 0 DO                          (pad len)
52         2DUP CHARS +  256 choose
53         DUP 0 HEX <# # # #> DECIMAL TYPE
54         SWAP C!
55         1+
56     LOOP
57     rc4-init                          ()
58     S" Hello World" 0 DO              (addr)
59         COUNT RC4 0 HEX <# # # #> DECIMAL TYPE
60     LOOP                             DROP
61     CR
62 LOOP ;;
```

```
BB5E4C58E99B7137B9F3368353D040471F4436E696
1BB203F1C9C91B1B378823E9EEC3597BC40D1DA596
9B032A403A0395FB9023BDD5280500BB4D3FF0BCFF
FEF5796F8DD48DAA38DBA289440B6C49B7A0B3F5C7
BC1DF14A210FC2C0930737D38E63AA36BC3B8CDAC8
4E377DEA14FE14A668C31D418971848478141DFEEA
E6E60B986317D87AD6A5D62C57E2B42BDD80B4CC74
DDCBEE3FE8D0CF944F0D3D6D7D00D84277DAF39841
53CC9421DDAD0D7DFD3916EF15A98C590BFD66FC4D
C310C22F37C26BB8A962B3BDE58A86E5693C5861AE
```

This example shows what the random initialization vector can do for encrypting the same text with the same user key.

I've not implemented CipherSaber-1 fully here, so you can do it yourself and become a CipherKnight.

MPE's Forth Coding Style Standard

Continued from the preceding issue. Portions of this document, including parts of some of the examples, were modified slightly to meet the requirements of magazine layout.

Formal data structures

Where a data structure is to be defined, the organisation should decide how it is to be defined. There is an abundance of data structure definition schemes in Forth, so choose one and stick to it. MPE has its own scheme, supplied in a file called STRUCTUR.FTH.

The use of formal data structures will lead to reliability, and makes the code much more maintainable. For example, see Figure One, which is much more readable and obvious than:

```
DECIMAL
0 OFFSET APPHANDLE
4 OFFSET DEVICE-VECTORS
8 OFFSET PATHNAME
136 CONSTANT PATHDATA
```

This last is full of magic numbers, and contains little of use to the maintenance programmer. In systems without data structure defining words (e.g., embedded standalone systems), the layout in Figure Two is more reasonable because the size of each field is much easier to see, and the structure can be reordered or extended by moving single lines.

Figure One

```
STRUCT PATHDATA          \ - struct-size ; I/O structure
                         \ for paths
 INT APP-HANDLE                  \ handle returned by OPEN
 PTR DEVICE-VECTORS              \ pointer to function table
 128 FIELD PATHNAME              \ zero-terminated file name
END-STRUCT
```

Figure Two

```
\             field name size     function
0
 DUP OFFSET APPHANDLE       4 +       \ handle from OPEN
 DUP OFFSET DEVICE-VECTORS  4 + \ I/O pointer
 DUP OFFSET PATHNAME        128 + \ ASCIIZ name
CONSTANT PATHDATA    \ - size ; I/O structure
                     \ for paths
```

Figure Three

```
: WORD1                  \ n1 n2 - ; word does ...
 ...                             \ lower-case code
 ...
;

: word2                  \ n1 - n2 ; word to ...
 ...                             \ lower-case code
 IF                              \ upper-case structure
 ...
 ...
 ENDIF
;

23 CONSTANT BILL
: word3                  \ n1 - ; function to ...
 ...                               \ lower-case code
 BILL +                            \ upper-case constant
 ...                               \ lower-case code
;
```

Case questions

The case of the words used in a Forth application is a very delicate issue, as different programmers have different preferences. All MPE Forth systems are case-insensitive, and so the case used is only a recommendation, not a requirement.

It is recommended that lower case be used throughout. Firstly, this is easier to type, and secondly it is easier to read. Using upper case throughout is *not* recommended, although traditional in some companies. In documentation however, Forth source code should be written in capitals to distinguish it from the body text. A fixed font such as Courier is also recommended.

However, it may be found that certain classes of words are better capitalized. These might be control structures or constants, or the name of the word as it is defined. Each organisation will find any preference and use it. An important point to remember with any decision made is to be consistent throughout all source code. For example when pro-

Microprocessor Engineering, Ltd.
Southampton, England

gramming for Windows, the word
`GETWINDOWHANDLE`

is much less readable than the word
`GetWindowHandle`

The examples in Figure Three are deliberately in mixed case, and do not follow the convention detailed at the start of the document.

MPE house rule

Lower case is preferred, especially for the standard Forth keywords. Mixed case makes compound word names much easier to read.

A hyphen or underscore can also be used in compound names. MPE is moving away from hyphens to underscores, particularly in mixed language systems where a hyphen may be treated as a minus sign by many parsers.

Converting from screen files

We do not wish to reopen the screen/text file debate, except to say that MPE uses text files unless there is no choice. We do this for a variety of reasons including choice of editors, compatibility with version control systems, and compatibility with third party software.

The move from screens to text files for source code encourages better layout and commenting, as definitions do not have to be crammed into the 14–15 useful lines of a screen.

There are several layout issues to be considered when converting source code in screen files into text files.

The first, and major, issue is whether to maintain the code structure as one page per screen, or to merge several screens into better structured pages. The decision here will depend on how scatter-loaded the existing code is. If the code is simply contiguous, it does not matter if the pages reflect the old screen numbers. However, if the code is loaded one screen here, then another over there, and another over here, etc., the page structure will have to reflect the screen layout in order to avoid breaking the compile sequence. This decision must be made on a per-project basis.

Another concern is that of actual layout. In a screen file, code is often crammed into a screen simply to avoid it falling off onto the next one, and the layout suffers accordingly. However, when converting to a text file, there is no concern over the length of the page or the file, so the layout may be much better. When code is converted from one format to the other, it is easy and quick to simply leave the layout as it is. However, if the code is still to be maintained and upgraded, it is worth the time and effort to relay the code to the layout standard. The benefit will only be apparent later on.

The layout and use of comments can also improve when converting code from screen files to text files. There is a tendency to ignore the comments in order to get more code onto the screen. This can be reversed when converting to a text file, as there is plenty of room. The old technique of using shadow screen for comments may be finished with in favour of comment blocks, much like the header blocks for comments in C. These chunks of comment may be either before or after the code itself but, being in the same file, will be printed and viewed with the code, using only standard tools. It is also possible to extract the comment blocks from the source for printed documentation.

Conclusion:
A note on consistency

This standard presents an approach to thorough coding and layout. One of its main threads is consistency. Even if the layout standard chosen does not entirely reflect this document, the layout adopted should be consistent in what it does. The key features for a standard are:

- Consistency from one programmer
- Consistency between many programmers
- Easy to follow
- Easy to understand
- Code which is easy to read and understand
- Code which is difficult to get wrong because of layout
- A layout which is also pleasant
- Unambiguous meaning

This concludes the series of articles about the MPE Forth coding style. We would like to thank Stephen Pelc and Microprocessor Engineering, Ltd. for graciously sharing this document. We hope it will inspire others to consider reasons for adopting a coding-style standard and to implement one that is appropriate to their own environment. —Ed.

Least Squares Estimation

This month's column is not what I initially planned. I had originally decided to set aside our current thread temporarily and to look at something completely different—the process of software development itself. This is a vitally important topic, so I've decided to treat it as the next stage of these essays, instead of as an aside. Consequently, we will be charging ahead with learning how to make an adaptive PID controller.

You may recall that all the adaptive filters we described last time involved the phrase *least squares*. For adaptive PID controllers, the concept will also show up. This month, we will take a look at just what a least squares estimation is, and how it can be used.

Optimal estimation

Least squares estimation is one of the techniques for doing an *optimal estimation*. In these sorts of problems, we have an adjustable model of the way our system is supposed to behave and some actual measurements of the system. Because our measurements are real world measurements, they will contain some errors (due to calibration, or resolution limitations, or because of noise). It's also possible that our model of the way the system works is not the way it actually works, so there could be some error in the description of it. Nevertheless, in spite of the presence of both types of errors, we want to make the best possible estimation.

There are many ways to quantify what we mean by "the best possible estimate." This measure of the quality of our estimate is called a *cost function* and, frequently, the type of application we are working with will dictate what the cost function will be. A least squares cost function works in the following way. Given a provisional set of model parameters:

- for each measured data point, calculate what the model would give;
- take the difference between the two, this is the error;
- square the error, and sum them for all the data points.

Squaring the errors eliminates any effect associated with the difference between positive errors and negative errors (for some problems, this is *not* a good idea) and turns out to be particularly mathematically convenient (unlike, say, the sum of the absolute values of the errors).

Fitting data to a straight line

The simplest example of the application of least squares estimation is the fitting of data to a straight line. In this problem, our model of the system is the equation for a straight line,

$$y = a\,x + b \tag{1}$$

where a and b are the unknown adjustable parameters. If we let z represent the measurements of what should be y, i.e.,

$$z = ax + b + \text{error}$$

then we can define our cost function as:

$$J(a,b) = \sum_{k=1}^{n} (\text{error})^2 = \sum_{k=1}^{n} (z_k - ax_k - b)^2 \tag{2}$$

To apply the least squares formalism to this, we need to figure out how to minimize J with respect to a and b. We do this by applying a little bit of calculus. The extreme values of a function (maximum and minimum) occur where the derivative is zero.

The fact that it is straightforward to handle the derivative of a squared quantity is what makes using the square more attractive than the use of the absolute value. We have two parameters, so we need to calculate the derivatives with respect to both of them,

$$\frac{\partial J}{\partial a} = -2\sum_{k=1}^{n} (z_k - ax_k + b)x_k = 0 \tag{3}$$

$$\frac{\partial J}{\partial b} = -2\sum_{k=1}^{n} (z_k - ax_k + b) = 0 \tag{4}$$

We need to expand these out and solve for a and b when both of these equations are simultaneously set to zero.

Now we have derived from (3),

$$0 = -2\left[\sum_{k=1}^{n} x_k z_k - (a+b)\sum_{k=1}^{n} x_k^2\right] \tag{5}$$

and from (4),

$$0 = -2\left[\sum_{k=1}^{n} z_k - a\sum_{k=1}^{n} x_k - bn\right] \tag{6}$$

These can now be used to give equations for a and b,

$$a = \frac{1}{D}\left(n\sum_{k=1}^{n} x_k z_k - \sum_{k=1}^{n} x_k \sum_{k=1}^{n} z_k\right) \tag{7}$$

$$b = \frac{1}{D}\left(\sum_{k=1}^{n} x_k^2 \sum_{k=1}^{n} z_k - \sum_{k=1}^{n} x_k \sum_{k=1}^{n} x_k z_k\right) \tag{8}$$

for,

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.

$$D = n\sum_{k=1}^{n} x_k^2 - \left(\sum_{k=1}^{n} x_k\right)^2$$

This is our optimal, least-squares estimate of *a* and *b*. Note we should verify that this solution is the *minimum* solution and not the *maximum* (remember that the first derivative is zero at both places). This verification requires taking the *second* derivatives and establishing that they are positive. This is pretty easy to show, if one looks at (5) and (6). The second derivative of *J* with respect to *a* is the derivative with respect to *a* of the right hand side of (5), which is

$$2\sum x^2$$

Since this is the sum of squared quantities, it is positive, so the second derivative is positive. Doing the same for *b*, we take the derivative with respect to *b* of the right-hand side of (6) and we get *2n*, which again is positive.

Listing One shows an example of a general-purpose, least squares fit routine. It takes data pairs *(x,z)* and returns the optimal estimates of *a* and *b*. For the sample data file in Listing Two, you should get a slope of 0.1781 and an intercept of 0.3687. With a sufficient amount of patience (or by putting Mathematica to work), we can work out equations like (7) and (8) for any polynomial form, not just for a straight line.

A nonlinear example

In the above example, we could solve for the optimal set of parameters because we could separate out the equations to give one just for *a* and one just for *b*. In order to do this, the cost function must be *linear with respect to the parameters:* that is, the parameters do not show up as nonlinear functions of each other (e.g., a^b, or within transcendental functions, such a *cosine*).

Sometimes this linearity can be achieved by applying a transformation to the original problem, to arrive at a new problem which is linear. For example, consider the problem of estimating the amplitude and phase of a signal at a known frequency,

$$\theta_k = A\sin(\omega t_k + \alpha) \tag{9}$$

where ω is known, θ_k and t_k are observed, and where A and α are to be estimated.

The simplest cost function would be

$$J(a,b) = \sum_{k=1}^{n} (\text{error})^2 = \sum_{k=1}^{n} (\theta_k - A\sin(\omega t_k + \alpha))^2 \tag{10}$$

Our equations now give us,

$$\frac{\partial J}{\partial a} = -2\sum_{k=1}^{n} \left[\sin(\omega t_k + \alpha)(\theta_k - \sin(\omega t_k + \alpha))\right] = 0$$

$$\frac{\partial J}{\partial b} = -2\sum_{k=1}^{n} A\left[\cos(\omega t_k + \alpha)(\theta_k - A\sin(\omega t_k + \alpha))\right] = 0$$

These equations are hopelessly intertwined—the α terms cannot be pulled out into an equation for α that is separate from the A in the same way that we achieved (7) and (8). The direct solution of these equations requires an approximation involving successive iterations or some other method.

This problem, fortunately, has the nice property that it can be transformed into another estimation problem which is linear. If we let,

$$x_k = A\sin(\omega t_k + \alpha) \tag{11}$$

$$y_k = A\omega\cos(\omega t_k + \alpha) \tag{12}$$

then, with a little trigonometry, we can write our unknowns A and α as,

$$A = \sqrt{x^2 + (y/\omega)^2} \tag{13}$$

$$\alpha = \tan^{-1}\omega x / y \tag{14}$$

(as long as the amplitude and phase are constant, it does not matter where in our data set we apply (13) and (14), so we drop the indexing subscripts).

Now can we find an optimal estimate of *x* and *y*? If so, we are all set. It turns out that we can. The complicating factor here is that our cost function must now account for the two components *x* and *y*, while we still are relying on the observations θ.

With lots of messy, tedious, but straightforward algebra, we can solve this problem to arrive at:

$$x = \frac{1}{D}\sum_{k=1}^{n} \sin^2 \omega t_k \sum_{k=1}^{n} \theta_k \cos\omega t_k \tag{15}$$

$$y = -\frac{\omega}{D}\sum_{k=1}^{n} \sin\omega t_k \cos\omega t_k \sum_{k=1}^{n} \theta_k \cos\omega t_k \tag{16}$$

for,

$$D = \omega\sum_{k=1}^{n} \cos^2 \omega t_k \sum_{k=1}^{n} \sin^2 \omega t_k - \omega\left[\sum_{k=1}^{n} \cos\omega t_k \sin\omega t_k\right]^2$$

How does one come up with this kind of transformation for a new problem? There are a few fairly standard transformations you can use, for instance,

$$y = ax^b$$

can be converted to a straight line-fit problem by taking the logarithm of both sides. But mostly, finding a suitable transformation is a matter of experience, persistence, and luck.

Conclusion

Now we understand what least squares estimation is, and we are comfortable with how to use it to determine an optimal estimate of the parameters system. You probably won't be surprised to learn that there are other complicating factors that could be considered, but that I have left them out for this introduction. You will notice we have assumed that, when we make a measurement, our knowledge of *x* was exact and that all the uncertainty/error was in *z*. We can reformulate the equations to handle the situation where the error is

in *x* and not in *z*, or even when there are uncertainties in *both x and z*. We can also readily extend the equations to handle the possibility that some *x,z* data measurements are more accurate than others. I have also not shown how to write any of this in matrix form. While the linear algebra formulation is extremely powerful, I have discovered, after years of being a professor, that the mere mention of the term "linear algebra" causes students to quake in fear. It's really not that difficult a subject, but I now know better than to spring it on anyone without some prior preparation.

Least squares is the analytic tool we need to create an adaptive PID controller. Our model will be the PID equations, and the parameters are the gains. Next time, we will work out the optimal estimators for a PID controller and discuss how the estimation equations can be written in a form suitable for a real-time system.

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

Listing One

```
\ lsq.fth   Calculates the Least-Squares optimal fit to a straight line, y = ax + b,
\           from  the sampled x,y data. Presumes that all the measurement uncertainty
\           is in the y
\
\           The input file consists of a single line giving the number of data points,
\           followed by that many lines of x y sample points.
\
\ This is an ANS Forth program requiring:
\         1. The Floating point word set
\         2. The File wordset
\         3. The conditional compilation words in the PROGRAMMING-TOOLS wordset
\         4. The Forth Scientific Library ASCII file I/O words
\         5. The standalone version requires access to the command-
\            line arguments, the PFE version is implemented here
\   There is an environmental dependency in that it is assumed
\   that the float stack is separate from the parameter stack

\ This code is released to the public domain.  August 1997 Taygeta Scientific Inc.

\ $Author:   skip  $
\ $Workfile:   lsq.fth  $
\ $Revision:   1.0  $
\ $Date:   27 Aug 1997 12:17:16  $
\ ==============================================================

S" /usr/local/lib/forth/fsl-util.fth" INCLUDED
S" /usr/local/lib/forth/fileio.fth"   INCLUDED

FALSE CONSTANT STANDALONE    \ set true to run as a standalone script

-1 VALUE fin  \ input file handle


VARIABLE n      \ count of data points
FVARIABLE sumx
FVARIABLE sumxz
FVARIABLE sumz
FVARIABLE sumx2

STANDALONE [ IF]

VARIABLE f_index   1 f_index !

: next_file ( -- c-addr u )  \ get filename from the command line
                             \ when running as a standalone script
```

```
                                \ this version is for PFE
    f_index @ argc >= if
        0 0
    else
        f_index @ argv
        1 f_index +!
    then
;

[ELSE]

: next_file ( -- c-addr u )
    bl word count
;

[THEN]

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

: F+! ( addr -- , F: x -- )     \ +! for floats, presumes separate float stack
     DUP F@ F+ F!
;

\ ============================================================

: lsq-init ( -- )
    0 n !
    0.0E0 sumx  F!
    0.0E0 sumxz F!
    0.0E0 sumx2 F!
    0.0E0 sumz  F!
;

: calc-det ( -- , F: -- d )

    n @ S>F  sumx2 F@ F*
    sumx F@ FDUP F* F-
;

: estimate ( -- , F: -- b a )
    calc-det

    \ calculate b
    sumx2 F@ sumz F@ F* sumx F@ sumxz F@ F* F- FOVER F/

    FSWAP
    \ now calculate a
    n @ S>F sumxz F@ F*  sumx F@ sumz F@ F* F- FSWAP F/
;

: lsq ( --<infile>-- )
    lsq-init

    next_file

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

    CR
```

```
    fin get-int  DUP n !        \ read count of points

    0 DO
        I .
      fin get-float  \ get X point
        FDUP F.
      FDUP sumx F+!
      FDUP FDUP F* sumx2 F+!

      fin get-float  \ get Z point
        FDUP F.
      FDUP sumz  F+!
        F*   sumxz F+!
        CR
    LOOP

    fin CLOSE-FILE DROP

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

Listing Two

```
12
0.0e0 0.0e0
1.0e0 0.56e0
2.0e0 1.11e0
3.0e0 0.72e0
4.0e0 1.28e0
5.0e0 1.44e0
6.0e0 1.41e0
7.0e0 1.81e0
8.0e0 1.60e0
9.0e0 1.78e0
10.0e0 1.92e0
11.0e0 2.55e0
```

Pygmy Embellishments

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

A Forth user for 13 years, Mr. Fergus is heavily involved in a personal, severe weather warning project (http://www.theramp.net/sferics). He appreciates Forth's "interactive control and limited restrictions."

From many Forth applications over a number of years, a collection of favorite tools has been assembled. These tools were added to different Forth packages as the need arose. After difficulties and work-arounds with the various Forth packages, Frank Sergeant's Pygmy has been the final step in the development of my "ideal" Forth development platform. Not only is the Pygmy source code readily available for embellishment, but the editor, metacompiler, and multiple open files make revision and application development a breeze.

Introduction

My applications are generally instrumentation and real-time data acquistion/analysis with a combination of a PC and embedded, Forth-based, single-board systems. The goal of my trek through these developments was a single package which could provide not only development tools, but also the running software. Forth, specifically Pygmy, has made that goal possible.

As these developments progressed, a number of tools were collected which gave Forth full control of the PC. The code is divided between original and excerpts from many publications. At this time, I would like to express my appreciation for the many authors who have published materials which have been sources for ideas or for actual code. I view many publications as a contribution to the "state of the art" and, thereby, available to the development community. I hope some of my efforts, which are available without restrictions, will fit in the same category, or serve as "food for thought," for future Forth developers.

Caveats

The code was tested for the application at hand, but is not guaranteed to be bulletproof.

The code is not included in this article, but is available by FTP from Taygeta[1], my homepage[2], or directly from me (disk and SASE please). These include a modified Pygmy kernel, Pygmy program, and the complete screen file. For legitimate use of the embellished program, the Pygmy distribution and Bonus Disk should be obtained through normal channels, so Frank Sergeant's efforts will be appropriately acknowledged.

Kernel

The original Pygmy kernel is essentially intact, although the word screen locations have been rearranged. Two kernel load screens (screens 1 and 2) are available, to compile a 64K or 128K version. I have found the 64K version sufficient for most applications. If large data arrays are needed, I prefer to allocate memory outside the program segment, thus saving the program segment for code.

Several extension load screens have been sandwiched between the initial kernel load screens (1–3) and the remaining kernel screens (10–96). These screens (4–6), in conjunction with initialization screens at the end of the listing (230–233), provide for different machine configurations.

Throughout the kernel, a number of words were added to provide compatibility with code written for other Forth dialects. Some of these words merely rename existing words, while others provide slightly different functions. The descriptions with the words should be sufficient and, therefore, will not be provided at this time.

The number-display functions (screen 52) were modified to accommodate double-significance numbers. Likewise, the `NUMBER` function (screen 75) converts text strings to single-/double-significance results, along with the `DPL` (decimal position) variable, in the traditional Forth manner while maintaining Pygmy's on-the-fly number base selection.

The `EMIT`-related functions (screens 45 and 49) have been expanded to provide printer or disk logging from the terminal data. In addition, the current output devices are flagged with various bits of the `IOBYTE` variable. More detail will be presented with the description of the terminal/host functions.

The somewhat misnamed `SHIFT?` word (screen 48) returns the condition of the keyboard control keys (Shift, Alt, Ctrl, etc.)

Two convenience words were added to the file-control functions. `REMOVE` (screen 61) deletes a file from the open file list. I use the file list as part of the documentation as a project progresses, and `REMOVE` allows me to clean up the file list easily.

`NEW-FILE` (screen 64) opens a screen file from the command-line name, with two blank screens. Example:
`" C:\FORTH\NEWFILE.PYG" 3 NEW-FILE <Enter>`
opens a new file as unit #3 with pathname in the quotes.

Editor (screens 99–111)

In general, the original editor functions have not been changed. A `FLOAT` (screen 111) word was added to complement the `SETTLE` function by moving non-blank screens to higher locations. In addition, the `ALTERNATE` function (screen 108) was modified to allow additional, unrestricted "flipping" between screens from any open file.

One minor addition, which has saved me some headaches when I have been a little too wild on the keyboard, was a keypress (Ctrl-X) to exit a screen without saving the changes.

The keypress menu/help (screens 99 and 100) was moved to the right side of the screen display. A not-so-obvious advantage was to provide more usable display. Since the screen contents are still visible after exiting the editor, the lines below the screen can be used as work space to test words, with reference to the screen still in view.

DO ... LOOP (screens 136–137)

Although Pygmy includes a `FOR-NEXT` function, at times I prefer the `DO ... LOOP` function (primarily because the index can be used as a pointer, thereby simplifying the stack). These `DO ... LOOP` constructs include both signed (`+LOOP`) and unsigned (`/LOOP`) increments. Unlike some Forth dialects, these

definitions are consistent and do not include the limit value in the either the positive or negative incrementing loop.

String variables (screen 138)

String variables are defined with VAR$. It is used in the following manner: VAR$ <name> <string>"

When defining, the first space after the name delimits the name, while any and all following spaces will be included in the string. In addition, a null character is added at the end of the string but is not included in the character count. Calling *name* will return the conventional TYPE format of address and count.

$+ and $++ are used to combine strings. In general, the $+ word will build a single string in the PAD area from the two strings (address count) on the stack. The $++ is a little faster when adding more than two strings, although the $+ will also work.

Double-significance math (screens 139–145)

The double-significance words in these screens were extracted from Frank Sergeant's Bonus Disk.

MAKE-DOER (screen 145)

One of my favorite constructs is the MAKE-DOER construct described by Brodie in *Thinking Forth*. This is similar to the DEFER (and IS) word used to make a dictionary entry for a word which cannot be defined at the time.

MAKE-DOER allows the same non-defined dictionary entry (DOER) to be created, and redefined later with MAKE. Several words to be described later will demonstrate the use of this construct. The major difference from DEFER is that the MAKE code is not a word definition by itself, but code contained within a definition.

Memory access (screens 146–149)

Several words have been assembled to access memory. The words follow the Forth tradition with regard to C@, @, 2@, etc., in which a prefix character is added to indicate the address format: E for segment-offset, or L for long (20-bit) addressing. In addition, words are available to switch between segment-offset and long addressing.

Extended memory (screens 150–151)

If space has been reserved with the INT=xxxx switch of HIMEM.SYS in the CONFIG.SYS file, extended memory can be accessed for transfers via XMOVE. One word of caution: when DOS is loaded high, it is loaded in the first 64K of extended memory. The INT=xxxx reserves memory beginning at the same address; therefore, the first 64K of extended memory cannot be used.

Memory allocation (screens 152–153)

The memory allocation words allocate and free memory as required. The allocate words return the segment address of the beginning byte. If the allocation is improper, a flag and error code is returned, which can be used with ABORT" to describe the failure and abort the program. Additional description will be included when the initialization screens (230–233) are discussed.

Time-Date-Clock (screens 154–156)

The system time and date can be accessed as individual parameters or as a string variable, or can be displayed directly. In addition, each parameter can be reset, which also resets the CMOS clock.

The MS word provides a delay function, which is independent of the machine clock. It uses the CMOS clock interrupt to provide delays in nearly 1 millisecond (977 microseconds) increments.

CLK@ returns a double number representing the clock ticks (about 18.2 per second) since midnight. I find this a fast way to calculate differential time throughout the day.

TAG$ and C.TAG$ return string variables which include the current year, month, day, and an incrementing number. C.TAG$ differs from TAG$ by substituting an input character for the tens year digit. My purpose for these words is to generate unique filenames, generally for data, which also include the date information.

Disk-file access (screens 157–159)

The disk-file access words were originally included to provide compatibility with Uniforth, and are used primarily for data transfers. Although there is some redundancy with the Pygmy functions, the CHANx file accesses are independent functions, with a separate disk buffer. This buffer can be located anywhere in low memory and, therefore, can be used to move data directly to or from the disk without an intermediate transfer step (via the normal disk buffer).

Files can be opened from within a word definition. Example:

```
VAR$ DAT  .DAT"
```

defines a file-extension string.

```
: TEST CHANB TAG$ DAT $+ $CREATE ;
```

Executing TEST will open or create a file named with the current time and date plus .DAT extension as unit 14. At a later time, executing CHANB will enable REOPEN, CLOSE, BUF-READ, BUF-WRITE, WRBYTE, RDBYTE, EOF@, and BOF@ to access the file via unit 14 (CHANB). Concurrently, files could be opened for CHANA and CHANC for similar accesses.

Display control (screens 160–167)

Display control includes video mode (text, graphics, etc.), character and background color, character attribute, scrolling, cursor, and display pages (up to eight).

MODE! and MODE@ set and recall the video mode. Generally, a machine will support many video modes (hardware-dependent) and MODE (screen 229) can be used to test a mode value. I use three "generic" modes for various effects: CLR80 (text), MEDGPH (320 × 200 graphics), and HIGPH (640 × 480 graphics). These words are defined during the program load process to use the optimal mode for each machine.

When in text mode, the entire screen can be scrolled in either direction with SCROLL or, with a little manipulation, a portion of the screen can be scrolled with UL!, LR!, and (SCRL).

Attributes can be built by selecting a color and following with the attribute word or words. Example:

```
RED BLINK BOLD
```

will set the character color with a bold (high intensity) blinking attribute.

```
BLUE MAT
```

will add the background color attribute to the character attribute.

Combining the above:

```
RED BLINK BOLD BLUE MAT DISPLAY
```

will set the display for blinking bold red characters on a blue background.

Up to eight separate text display pages may be selected with DSP.PAGE. Each page may be manipulated separately, including the page attributes.

The ATTR$ is useful to highlight a character block for attention or prompting. It changes the attributes, but not the characters.

DOS shell (screens 171–172)

DOS commands can be executed at the ok prompt or from within a word. To execute a DOS command from a keyboard entry, type DOS [command] <Enter>. Example:

```
DOS DIR C:\ <Enter>
```

will show the C drive directory and return to the Pygmy ok prompt. Typing:

```
DOS <Enter>
```

will call the command interpreter and provide the standard DOS command-line functions. To return to Pygmy, type EXIT <Enter>.

To execute a DOS command from within a definition, build the appropriate string variable for the DOS$ word to execute. Example:

```
VAR$ DIRECTORY DIR C:\" : C:DIR ( --- )
DIRECTORY DOS$ ;
```

The C drive directory will be displayed when C:DIR is called.

To allow for different machine configurations, the COMMAND.COM path is defined on the load screen as a string variable (COMD).

Mouse (screens 173–183)

A number of BIOS calls related to mouse control have been coded, but only a few are used routinely. The mouse driver reset, cursor on/off, and position control are contained on screen 178. Another useful call is M!GCB (screen 180), which sets the graphic cursor shape. Cursor shapes are defined with a 32-byte array, as detailed on screens 174–177. By switching the number base to 2, the two masks can be drawn directly with 1s and 0s, which represent each mask pixel.

Text windows (screens 184–203)

The window function overlays a portion of the display screen with a window screen. The overlayed text is saved, to be restored when the window is closed. Allocated memory (outside the program segment) is used for storage; therefore, a relatively unlimited number of windows may be open at one time.

The window is selected by defining the upper-left corner position, the horizontal and vertical width, and the border and text attributes (OPEN.WIN, screen 196). OPEN.NORM is similar, except default white attributes are used. A border (double-lines graphic characters with separate attributes) outlines the opened window.

Various "emit" words have been defined for functions to stay within the selected window. Generally, these words are the conventional Forth word with a W prefix; e.g., WEMIT for EMIT, WCR for CR, and WTYPE for TYPE, etc.

Several special windows are available to provide complete functions. WIN.WORDS draws a window at position *x,y* with *n* horizontal spaces (only one line), and includes a prompt defined with the PROMPT Doer. The prompt may be a default word(s). Before executing the window, the window contents may be changed via the keyboard. Pressing Enter will execute the words in the window and close the window. Example:

```
: F.DIR ( --- ) MAKE PROMPT ." DIR
C:\FORTH " ;AND 10 10 20 WIN.WORDS ;
```

If not typed over, the C:\Forth directory will be displayed when F.DIR is called.

A file can be opened via the CHANx scheme with WIN.FILE, similar to the above. Example:

```
: DATA ( --- ) MAKE PROMPT ." TEST.DAT"
;AND CHANA 10 10 20 WIN.FILE ;
```

DATA will draw a window showing TEST.DAT and open a file named TEST.DAT, unless changed, as CHANA (unit 13).

A keyboard number entry window can be drawn with WIN.KB# at a selected position and width. Either single- or double-significance entry format may be used. With window closure, a single or double number is left on the stack. Although somewhat awkward, the significance may be determined with the variable DPL.

Serial port (screens 204–219)

Full serial port capability is provided by direct access to the machine hardware. The receive function is interrupt-driven, with a large buffer outside the program memory segment, thereby reducing timing problems during high-speed transmissions.

COM.ON resets the buffer pointers, installs the interrupt vector, and initializes the serial port hardware. COM.OFF disables the serial port and restores the prior interrupt vector.

Any serial port address, interrupt number, baud rate, and character setup can be used. Words are included for Com1 and Com2, 9600–1200 baud, and "8N1" characters. Example:

```
COM1 9600B COM.ON
```

enables serial port 1 for 9600 baud, and it can be disabled with COM.OFF. To re-enable, only COM.ON is necessary.

Several definitions are provided for terminal emulation and host-slave functions. These words were written to coordinate the development of small Forth-based microprocessor systems. In general, these systems are connected via a serial cable, and the PC becomes both terminal and disk resources for the system[3].

Terminal emulation is provided with TT, which includes a disk logging and printer function. The emulation is entered with TT <Enter>, and is exited with the F10 key.

While in emulation mode, a disk logging function is toggled on/off with the F1 key. When toggled on, a prompt for a filename will be displayed. After entering the filename, all subsequent terminal display will be written to the file until F1 is pressed. The disk logging is especially useful to transfer a slave program to a disk file for burning EPROMs. Similarly, the F2 key toggles the printer on/off.

A function (HH) is included to pass file blocks to and from the slave via the serial cable, in addition to terminal emulation. Obviously, the slave must have the appropriate code to send and receive the blocks.

Simulated input strings can be sent to the slave with $LOAD, which passes a string variable to the slave, followed by a carriage return. For slaves with a 1024-character input buffer,

Continued on page 27

Call for Papers: FORML Conference

New FORML dates...
...the week* before *Thanksgiving!

The original technical conference for professional Forth programmers and users.

19th annual FORML Forth Modification Conference
November 21 – 23, 1997

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

THEME:

"Forth at the Millennium"

What are the challenges for Forth as we reach the Millennium? Will the year 2000 present problems for existing programs? Many organizations are asking for certification that software will work perfectly as we move to 2000 and beyond.

How will certification be accomplished? Encryption is required for more applications to keep transactions private. Proposals for incorporating encryption techniques are needed for current and future applications. Your ideas, expectations, and solutions for the coming Millennium are sought for this conference.

FORML is the perfect forum to present and discuss your Forth proposals and experiences with Forth professionals. As always, papers on any Forth-related topic are welcome.

Abstracts are due October 1, 1997 • Completed papers are due November 1, 1997

Mail abstract(s) of approximately 100 words to:
FORML, Forth Interest Group • 100 Dolores Street, Suite 183 • Carmel, California 93923
or send them via e-mail to FORML@forth.org

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

Guy Kelly, Conference Chairman • **Robert Reiling, Conference Director**

Advance Registration Required — Call Today!

Conference attendee in double room $460
Non-conference guest in same room $330
Conference attendee in single room $600
Under 18 years old in same room $190
Infants under 2 years in same room – free
FIG members and guests eligible for 10% discount (through November 1)

phone: 408-373-6784
fax: 408-373-2845
e-mail: FORML@forth.org

FORML, Forth Interest Group
100 Dolores Street, Suite 183
Carmel, California 93923

The FORML Conference is sponsored by the Forth Interest Group