

F O R T H

D I M E N S I O N S

Extensible User Interface

Build a Target Compiler

Mushroom Identification

Runstk – Stack Utility

Stack Gymnastics Made Easy

OFFICE NEWS

Forth Interest Group Membership Benefits, SIGs, and FORML

In 1997, many changes took place in the Forth Interest Group. (I wonder when it will be that I don't write to you about changes?) The development of new levels of membership support was one important change. Another important change was the establishment of the Special Interest Group mail boxes in the members-only section of the FIG web site. A third change was the availability of sponsorships for FORML. There were many more, as you know, but these are the ones I would like to give you more information about.

Each of these activities was conceived and implemented both to help the Forth Interest Group to expand its available member services and, in turn, to help us more easily fund operations.

The following is a listing of our current membership levels and their benefits. Regardless of what type of membership you choose, please know that each and every member is a valued member.

Forth Interest Group Membership Levels

*Standard Membership, \$45.00 per year
(non-U.S. addresses add \$15.00)*

Benefits:

- One-year subscription to *Forth Dimensions*, FIG's bi-monthly magazine
- 10% discount on all publications and products
- Full access to FIG site on World Wide Web, including access to Special Interest Groups on the members-only page
- e-mail forwarding service

Benefactor Membership, \$125.00 per year

Benefits:

- All the benefits of Standard Membership, plus
- Special listing in *Forth Dimensions* as a Benefactor Member
- First Class postal delivery of *Forth Dimensions*

Company/Corporate Membership, \$125.00 per year

Benefits:

- All the benefits of Standard Membership, plus
- Five copies of each issue of *Forth Dimensions* sent First Class postal rate, for employees to share.
- Fifty-word "Corporate Member" listing in *Forth Dimensions* to describe your products and services.
- Corporate Member Advertising Rate sheet for *Forth Dimensions*.
- Corporate Members listing on the Forth Interest Group home page and a link to your Web site.

Library Membership, \$125.00 per year

Benefits:

- All Standard Membership benefits, plus:
- An extra set of the year's *Forth Dimensions* (six issues) at the end of the publishing year.
- A copy of the FORML Proceedings, the written record of each year's FORML Conference.
- Link to your Web site from the Forth Interest Group's home page.

We are also in the process of changing the way our membership information database is used. More and more, we are trying to obtain current e-mail addresses. It would be helpful—if you haven't received e-mail from the FIG office before or if you've recently changed your e-mail or forgot to tell us—to send us an update. Doing this will help us to increase the range of services we can provide to you.

For example, in the near future (I'm not sure exactly how near that will be) we hope to automate our membership renewal date reminders, for those of you who have Internet access. Getting renewals in a timely fashion helps us to keep *Forth Dimensions* coming to you uninterrupted. It will be crucial to have current e-mails (if you have e-mail) to be able to do this. If you don't have e-mail, not to worry: we will still send out hard copy reminders.

Special Interest Groups

Here is a list of the current SIGs available to you:

fig-camel	fig-pygmy
fig-games	fig-robotics
fig-linux	fig-safety
fig-mops	fig-win32for
fig-palmtops	forml20
fig-pilot	

As a member in good standing, you can have your e-mail linked to any of these mailbox discussion groups. They are established for your use, and are an excellent way to meet and share information with those who have the same "special interests."

Here's how to join a Special Interest Group:

1. Go to the FIG home page at <http://www.forth.org>
2. Click on the link for the Special Interest Group mail servers
3. Click the box next to the group(s) you would like to join
4. Fill out the form with your name, FIG member number, and e-mail address
5. Click the button that says, "Join SIGs"

After your request is sent, you will be added within the next day or so. We feel this is an exciting new service. Is there a Special Interest Group you would like to see added? Let us know, we're here to serve you.

FORML

This year's FORML Conference will be our 20th! Already we have participants giving us titles for the talks they are planning on presenting. As we get closer to November, more information will be available (or you can sign up for the SIG FORML20@forth.org and have up-to-date information). Our editor, Marlin Ouverson, has agreed to be the Program Chair this year, and Bob Reiling is FORML's Director. With your participation, it's guaranteed to be the best FORML ever!

A change that was implemented last year, and that we would like to continue this year, is the availability of increased financial sponsorships for FORML. Last year, we had three sponsors: FORTH, Inc. was a Bronze Sponsor; John D. and Jae

Continued on page 5

6

Stack Gymnastics Made Easy*by Ronald T. Kneusel*

Forth's stack nature is one of its strong points, but manipulating a large number of stack items can be somewhat difficult with traditional stack words. What is described here is based on a similar construct found in a Forth-ish programming language the author saw a number of years ago, and makes complex stack manipulations very easy.

9

Building a Remote Target Compiler*by Dave Taliaferro*

This represents the culmination of this three-part series about *remote target compilation*. A remote Forth target compiler runs on an embedded development host machine and allows interactive production of executable Forth and assembler routines for a target microprocessor. As keyboard definitions or source files are interpreted by the host Forth, the resulting code and data is transparently uploaded to the target for immediate testing and further programming.

18

Mushroom Identification*by Charles Samuels*

The plan was to allow users to point and click to describe an unknown mushroom to the program, and to get an identification. But the author had not done any serious programming for the past 12 years, and Windows had passed him by. So he had to create a serious data set *and* catch up on Windows programming. With LMI's WinForth and a third-party app to help with the interface, he developed what he hopes will become a commercial success.

19

An Extensible User Interface*by John J. Wavrik*

Interface design, whether graphical or not, deserves far more attention than it usually receives. It is, after all, how most others will experience—and, hopefully, use—our code. The author shows his research work to non-programmer mathematicians and students, and an obscure interface would result in even greater obscurity for his subject material. Here he provides his command-line interface tool.

28

Runstk – Stack Utility*by Warren Heath*

This text editor allows the display of the data stack symbolically anywhere within a word shown on the display. It allows the safe virtual running of source code, and can also automatically check cumulative stack effects against the given stack picture for the word. It can put the cumulative stack picture into the source for documentation, and can also show the stack picture of any word in the system.

DEPARTMENTS

2	OFFICE NEWS	32	NEW PRODUCT ANNOUNCEMENT
4	EDITORIAL Around the world	32	STANDARD FORTH TOOL BELT Character Literals
5	LETTERS Why is Forth not more popular?	33	STRETCHING STANDARD FORTH Double-Number Arithmetic
8	FORTH ON THE WEB	35	SPONSORS & BENEFACTORS

Around the World

That Forth is an international phenomenon is taken for granted by everyone. Major conferences have been held in several European countries and in Australia, and a formidable technical troupe visited Forth sites in China some years ago. There is a Forth web site in French, and an interesting Remote Experiment Lab in Brazil that lets web browsers execute their code remotely on an 8051 microcontroller; it uses CamelForth by Brad Rodriguez (a Canadian), and it isn't a net-based simulation. There is a considerable amount of German Forth information available, including a content-rich magazine, thanks to the long-standing efforts of Forth experts in that country. The U.K. is home to several Forth institutions and many Forth users, and to another admirable journal, *Forthwrite*, some of whose contributors have written for our own publication (and we hope will do so again!).

Of course, this cursory look cannot include all our respected readers and authors, from The Netherlands, Austria, Switzerland, Mexico, Japan, and other countries where Forth has taken root. We salute them all and welcome their continued participation in the international Forth community!

We cannot ignore Forth and the Forth users who happen to live on the other side of some geographical divide or political boundary. Some of them are doing exciting Forth research (albeit sometimes documented in a language we do not read well), while others of them are reminding us how to communicate Forth to the newly initiated; and still others are re-discovering Forth's strengths in resource-constrained environments, something on which we who earn a living from this language (or hope to) might capitalize.

On a more purely human note, of course, we are always happy to find common interests with those from differing circumstances and cultures. We are strengthened by diversity, even while we appreciate our differences.

Forth Dimensions invites written contributions from all Forth users, but particularly reminds our international readers that we would like to hear from them. And the FORML Conference this year (see the back cover) would welcome more international participants. It is this Conference's 20th anniversary, so we hope you will come both to share in the technical program and to celebrate with us!

—Marlin Ouverson, Editor

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

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

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

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

Fred Behringer
Planegger Strasse 24
81241 Muenchen
Germany

E-mail: behringe@mathematik.tu-muenchen.de

Forth Dimensions

Volume XIX, Number 6
March 1998 April

Published by the
Forth Interest Group

Editor
Marlin Ouverson

Circulation/Order Desk
Trace Carter

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

Subscription to *Forth Dimensions* is included with membership in the Forth Interest Group at \$45 per year (U.S.) \$60 (international). For membership, change of address, and to submit items for publication, the address is:

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

Copyright © 1998 by Forth Interest Group, Inc. The material contained in this periodical (but not the code) is copyrighted by the individual authors of the articles and by Forth Interest Group, Inc., respectively. Any reproduction or use of this periodical as it is compiled or the articles, except reproductions for non-commercial purposes, without the written permission of Forth Interest Group, Inc. is a violation of the Copyright Laws. Any code bearing a copyright notice, however, can be used only with permission of the copyright holder.

The Forth Interest Group

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

FORTH DIMENSIONS (ISSN 0884-0822) is published bimonthly for \$45/60 per year by Forth Interest Group at 1340 Munras Avenue, Suite 314, Monterey CA 93940. Periodicals postage rates paid at Monterey CA and at additional mailing offices.

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

LETTER TO THE EDITOR

Why is Forth not more popular?

The editorial in the January/February issue posed the question "...why is Forth not more popular?" I would like to address this issue, at least in the Windows arena. First we have to realize that there are at least two types of people who use programming languages: the professionals and laymen. For every professional programmer, there are probably 1,000 people who program to solve specific tasks related to their profession. I belong to the latter group, who are only interested in solving the problem in the fastest way possible.

Computer hardware has made enormous advances in the last 15 years, but software development is still in the dark ages. Changes in computer languages and operating systems have primarily made them more complex and difficult, especially for the non-professional or occasional programmer.

What is needed is a *programming application* (PA) where GUIs can be used to create complex programs. This idea is used to some extent in some of the web-page creation programs and in Borland's Resource Workshop.

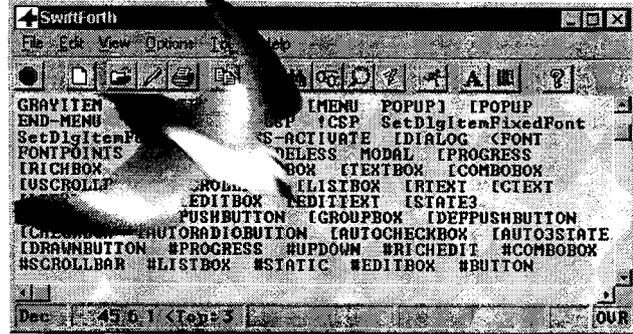
Drag-and-drop of mouse-selected objects can create complex dialogs, menus, and display items. Selection of such objects in the run-time program could execute Forth words or display other objects or windows. The programming application should hide the complexity of the operating system and handle all the mundane garbage. The end product of the PA would be an executable file of minimal size.

A well-designed PA could make creating Windows applications simple. I am a firm believer in the idea that a better mousetrap sells. Forth could leapfrog ahead of other programming languages by going after those 1,000 programmers who largely have been abandoned.

Charles Samuels
fungus@gte.net
<http://www.alaska.net/~arktika/discover.html>

All-new development environment from FORTH, Inc.

SwiftForth™ for Windows 95/98 and Windows NT



- Super-efficient implementation for speed (32-bit subroutine-threaded, direct code expansion)
- Full GUI advantages (like drag-and-drop editing; hypertext source browsing; visual stack, watchpoints, and memory windows) but retains traditional command-line control and tools
- Complies with ANS Forth, including most wordsets
- Easy to add DLLs and to call DLL functions
- DDE client services for inter-application communication
- Files and blocks supported
- Simple creation of windows, menus, dialogs, etc. — no third-party tools needed
- Flexible, extensible access to system callbacks and messages, and exception handler

FORTH, Inc.

111 N. Sepulveda Blvd., #300
Manhattan Beach, CA 90266-6847
800.55.FORTH ■ 310.372.8493 ■ FAX 310.318.7130
forthsales@forth.com ■ www.forth.com

Call today for data sheet
or visit our web site!



Office News, continued from page 2

H. were Silver Sponsors; and Taygeta Scientific was a Silver Sponsor. Lockheed-Martin was an equipment contributor.

We would like to see more sponsors for our 20th year! If you are interested in being a Bronze, Silver, or Gold Sponsor, or have equipment and/or services you would like to offer, please contact the office and we will give you the details.

As always, together we make a difference.

Cheers,

Trace Carter
Forth Interest Group
100 Dolores Street, Suite 183
Carmel, CA 93923
office@forth.org

This classic is no longer out of print!

Poor Man's Explanation of Kalman Filtering or, How I Stopped Worrying and Learned to Love Matrix Inversion

by Roger M. du Plessis

\$19.95 plus shipping and
handling (2.75 for surface U.S.,
4.50 for surface international)

You can order in several ways:

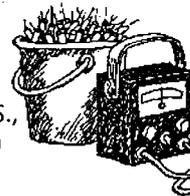
e-mail: kalman@taygeta.com

fax: 408-641-0647

voice: 408-641-0645

mail: send your check or money order in U.S. dollars to:

Taygeta Scientific Inc. • 1340 Munras Avenue, Ste. 314 • Monterey, CA 93940



For information about
other publications offered
by Taygeta Scientific Inc., you
can call our 24-hour message
line at 408-641-0647. For your
convenience, we accept Master-
Card and VISA.

Stack Gymnastics Made Easy

Forth's stack nature is one of its strong points, but manipulating a large number of stack items can be somewhat difficult with traditional stack words. What is described here is based on a similar construct found in a Forth-ish programming language I saw a number of years ago, and makes complex stack manipulations very easy.

The complete Stacker program is found in Listing One. This version is for Pocket Forth on the Macintosh. It is meant for use within a colon definition to generate a specific stack arrangement based on a given resequence string. For example, the following definition takes the top three stack values and removes the one second from the top:

```
: example1 ( a b c -- b c )
  abc ==> bc ;
```

Of course, this could be done just as easily with a `rot` `drop` sequence but what if, instead, we needed something like this:

```
: example2 ( a b c d -- c a d b )
  abcd ==> cadb ;
```

Here the traditional stack-manipulation words would be cumbersome. The Stacker notation, which mirrors the stack comment, makes this a much simpler task. Of course, there is a price to pay, which will become clear below, but it is generally a small one.

Stacker works as an immediate word, executing as soon as it is encountered. When `==>` is executed, the next token is read as the resequence string, which must be in upper case. Here, Pocket Forth "uppercases" for us. It then compiles first a call to `>store` to place the stack items into the storage array, and then compiles as many calls to `@store` as are needed to reproduce the resequence string. Note that Pocket Forth uses `r` where most other Forths would use `i`. For example,

```
: example3 ( a b c -- b c a ) abc ==> bca ;
```

is compiled as,

```
: example3 ( a b c -- c b a )
  abc >store \ top three stack items to
            \ the array
  1 @store \ fetch the second item, "b"
  2 @store \ fetch the third item, "c"
  0 @store ; \ fetch the first item, "a"
```

This is where the price has to be paid. Using Stacker results in code that is slower than the traditional stack words.

When to use Stacker and when to use the faster stack words is a judgment call. Naturally, a complex stack word used only once will result in a negligible increase in execution time.

Looking at Listing One shows lines three through ten defining simple constants. Stacker expects the top stack item to be the number of stack items to be resequenced. The constants makes the code easier to read. Lines 12 and 13 define storage for the stack values, here set for eight 16-bit value, and for the resequence string, which may be up to 80 characters long. Note that this version includes no error control; nothing stops the user from referencing array cells that aren't valid, as in `abc ==> abf`. Lines 15-18 define access words for the storage array. Stack values are placed in the array so that the highest stack item is the highest array element. This makes it easier to convert a character of the resequence string into an array index.

The main word, `==>`, is defined in lines 22 through 28. It is an immediate word, because we want it to execute while in compile mode. Line 23 gets the resequence string text and moves it to the `text$` buffer. `Token` is predefined in Pocket Forth; it gets the next token from the input stream and places it at here as a counted string. This text is then copied to the `text$` buffer to prevent it from being overwritten as words are compiled into the dictionary. Line 24 contains the cryptic phrase,

```
[ ' >store literal ] compile
```

which takes the execution address of `>store` and compiles it as a literal into the definition of `==>`, which is only being defined at this point. When `==>` is executed as an immediate word while defining a new word, the execution address of `>store` will be compiled as a subroutine call into the new word being defined. Pocket Forth's use of `compile` is non-standard. This phrase appears again in line 26 to set up the call to `@store` necessary to re-load the storage array elements. Specifically, lines 25-27 set up the loop which places repeated calls to `@store` to retrieve the appropriate array elements according to the resequence string. It loops over the resequence string elements, turning them into indices into the storage array, and compiles them as literals into the word being defined. Finally, line 28 ends the definition and marks `==>` as an immediate word. Lines 32 through 37 serve as examples of how one could redefine the traditional stack words using Stacker.

Stacker is small and easily added to any program. It makes the sometimes annoying task of stack manipulation just a little easier. Additionally, Stacker can be added to virtually any Forth system. This, of course, will surprise no one who is familiar with the power of Forth.

Listing One

```
1 - \ Stacker - Stack manipulator
2 -
3 - 1 constant a          \ constants to make things pretty
4 - 2 constant ab        \ really number of stack items to fiddle with
5 - 3 constant abc
6 - 4 constant abcd
7 - 5 constant abcde
8 - 6 constant abcdef
9 - 7 constant abcdefg
10 - 8 constant abcdefgh
11 -
12 - CREATE store 16 allot \ storage for stack values (8 max)
13 - CREATE text$ 80 allot \ storage for token text
14 -
15 - : () ( n -- addr ) 2* store + ;
16 - : @store ( n -- v ) () @ ; \ get nth
17 - : >store ( a b ... n -- ) \ put stack values in storage, in order
18 -   1- 0 swap DO r () ! -1 +LOOP ;
19 -
20 - \ Use this only in a colon definition
21 -
22 - : ==> ( -- ) \ get a resequence string and compile appropriate words
23 - token here text$ here c@ 1+ cmove \ save token text
24 - [ ' >store literal ] compile \ get values
25 - text$ text$ c@ 1+ + text$ 1+ DO
26 -   r c@ 65 - literal [ ' @store literal ] compile \ put a value
27 - LOOP
28 - ; IMMEDIATE
29 -
30 - \ Standard stack words re-defined with ==>
31 -
32 - : swap. ( a b -- b a ) ab ==> ba ;
33 - : dup. ( a -- a a ) a ==> aa ;
34 - : over. ( a b -- a b a ) ab ==> aba ;
35 - : nip. ( a b -- b ) ab ==> b ;
36 - : rot. ( a b c -- b c a ) abc ==> bca ;
37 - : 2dup. ( a b -- a b a b ) ab ==> abab ;
```

URLs — a selection of Web-based Forth resources

The MOPS Page

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

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

Frank Sergeant's Forth Page

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

Pygmy Forth and related files.

EE Toolbox: Software Development: FORTH Internet Resources

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

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

The Pocket Forth Repository

<http://jldh449-1.intmed.mcw.edu/pf.html>

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

AM Research, Inc., The Embedded Control Experts

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

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

Forth on the Web

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

A collection of links to on-line Forth resources.

Laboratory Microsystems, Inc.

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

The commercial site of LMI, with product information.

The Forth Source

<http://theforthsource.com/>

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

The Journal of Forth Application and Research

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

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

COMSOL

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

Computer Solutions supplies tools for microprocessor designers and programmers in the U.K.

MicroProcessor Engineering, Ltd.

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

MPE specialises in real-time and embedded systems.

Forth Interest Group in the United Kingdom

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

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

The Home of the 4tH Compiler

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

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

Space-Related Applications of Forth

<http://groucho.gsfc.nasa.gov/forth/index.html>

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

FORTH, Inc.

<http://www.forth.com>

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

Forth Interest Group Home Page

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

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

Forth Information on Taygeta

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

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

Jeff Fox and Ultra Technology Inc.

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

Information about Forth processors.

Offete Enterprises, Inc.

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

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

Forth Online Resources Quick-Ref Card

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

Extensive list of links to Forth enterprises and personalities.

The Forth Research Page

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

Peter Knaggs' list of Forth resources.

Yahoo Page on Forth

http://www.yahoo.com/Computers_and_Internet/Programming_Languages/Forth/

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

The Open Firmware Home Page

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

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

American National Standard Forth Information

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

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

Building a Remote Target Compiler

At long last we reach the apex in this three-part expose that rips the lid off the obscure yet powerful embedded development technique known as *remote target compilation*.

In the first article (FD XIX.3), we observed the CREATE DOES> mechanism for implementing defining words which can generate classes of child words with identical run-time behavior and variable data attributes. In the second article (FD XIX.5), it was seen that defining words can easily spawn data-compiling child words, and when these words are fed into the Forth interpretation stream a custom assembler or compiler emerges in a few lines of code.

A *remote* Forth target compiler runs on an embedded development host machine and allows interactive production of executable Forth and assembler routines to a target microprocessor. As keyboard definitions or source files are interpreted by the host Forth, the resulting code and data is transparently uploaded to the target for immediate testing and further programming.

In this article, I will demonstrate how a remote target compiler can be built using defining words, the Forth interpreter, and a few other nifty tricks. Since we have already covered the necessary groundwork, this piece will first explore the basic elements of remote Forth target compilation, then will describe a 56002 DSP target compiler I wrote using a handful of Forth concepts that collectively do something amazing—interactively compile and execute code from one architecture to another. These ideas are merely an extension to the last article, “Easy Target Compilation.”

Because I used a standard macro assembler to write the 56K Forth, my target compiler had some interesting twists to true Forth target compilation, which uses a Forth assembler to generate the target nucleus. In addition to producing a target Forth kernel, the macro assembler provides a means for the host Forth to “clone” the target Forth. Using a listing of the target Forth code addresses generated during the nucleus assembly process, a new dictionary is created on the host that enables the interactive development of Forth and assembler routines on the target.

This technique makes remote target compiler construction accessible to the pedestrian Forth programmer. Public-domain Forths exist in assembly form for most processors; coupled with any host Forth on their favored operating system, the development environment described here can be easily duplicated.

Some basic concepts of remote Forth target compilation

A *resident* Forth contains the Forth interpreter and compiler entirely on the native processor the Forth is running on. In the original eForth for the 56002, the target contained

an interpreter, compiler, and name dictionary, in addition to the target-executable code dictionary. This consumed about 9K processor words in the 56002. The goal of the remote target compiler concept is to move the Forth interpretation and execution tasks from the target microprocessor to the host computer. This leaves the minimal executable target Forth system needed to create and execute new applications interactively, and to act as an operating system for applications when the host computer is removed.

In other words, the embedded target includes a run-time interpreter to execute the application Forth source routines but does not include the interactive Forth interpreter/compiler. The interactive development occurs on the host machine; a run-time interpreter and the Forth kernel words on the target communicate with the host during development and allow for immediate testing of routines. When development is complete, a binary image is produced for ROMing or loading on the target.

This opens up interesting possibilities for embedded development, for now the target system’s behavior can be controlled by ASCII text scripts interpreted on the host Forth. Both host and target code execution and compilation can occur within the same source code scripts.

Forth exhibits metacompilation, the ability to reproduce itself on the machine on which it is running or on another computing system. For a Forth to regenerate itself to a different architecture, a target assembler implementing the target’s instruction set is written (see the preceding article, “Easy Target Compilation”), which is used to construct a Forth virtual machine for the target, which is itself used to build a high-level Forth model. This compilation process generally takes place in the host Forth’s memory, producing an image of the target Forth that can be loaded into the target memory for execution.

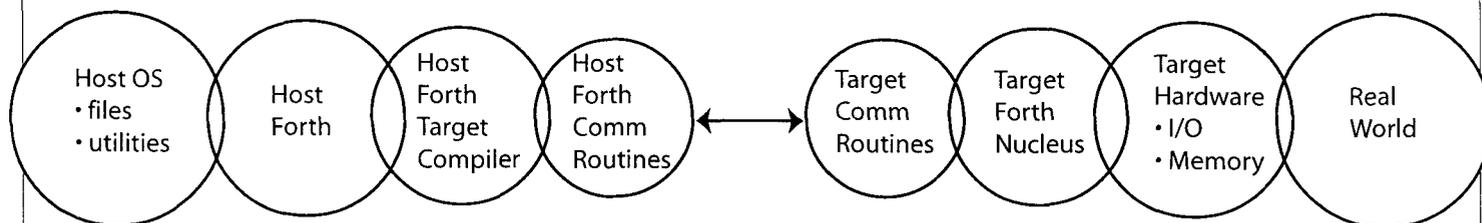
In the 56002 target compiler, metacompilation is used to “clone” the target nucleus, which already exists in the form of the eForth macro assembly source code. The tricky task of cross-metacompiling the host Forth to a target architecture Forth is eliminated. The assembler source provides the addresses for all the words in the target Forth’s dictionary, which are used to create a target clone dictionary on the host. (This new target dictionary may be identical to the host’s own Forth dictionary; depending on the state of target compilation, either dictionary may have the correct homograph.)

Metacompilation becomes simple, because now we merely use CREATE DOES> to construct a defining word that creates host clones for each target word, and then use [to interpret those words. With the addition of some serial communication routines and the embedded Forth virtual machine, *remote* target compilation is easily achieved.

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

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

Figure One



A minimal target Forth system would contain the core primitives (about 32 in eForth), and whatever additional words are needed to support the application. In a well-tuned application, this could amount to a few hundred bytes of code. Each assembly primitive consists of a few lines of machine code that implement some operation in the Forth virtual machine, such as pushing a data item to the stack or transmitting a byte out the serial interface.

One important primitive provides a threading mechanism, called the *inner interpreter*, that executes the list of addresses that comprises a high-level Forth definition. This primitive can be called through a communication routine to remotely execute target Forth definitions.

Thus, the component parts of a generic remote target compiler include:

- A target Forth nucleus consisting of the inner interpreter and core primitives.
- A host Forth for target program development and file management.
- The host Forth target compiler—a program running on the host that interactively compiles executable code into the target.
- Host and target communication routines; how one Forth talks to another.

Figure One gives a celestial representation of the remote target compiler development system universe.

Simple example:

A remote Forth target compiler for the 56002 DSP

The target Forth nucleus—56002 eForth

The Motorola 56002 is a 24-bit digital signal processor (DSP) with a Harvard architecture that can execute memory operations in parallel. The multiple address registers and three memory spaces allow it to operate on time sequences of input samples, the basis of digital filtering and signal processing.

I ported eForth, by C.H. Ting, to the 56002 by converting the MASM 8086 source to Motorola's freeware 56K macro assembler. Since eForth and its descendant, hForth, are designed for porting, the task was pretty straightforward. Implementations for a number of microprocessors exist in the public domain.

At reset, a microprocessor fetches its starting code location from a specific hardware address, called the *reset vector*. It then jumps to that location and begins executing whatever instructions are there. From this raw beginning, the system's hardware configuration is set up, after which the application code is started. On even the simplest architecture, we can breathe life into silicon by having a Forth virtual machine awaiting the execution path of the microprocessor after hardware initialization. For 56002 eForth, this consists

of the inner interpreter, called `doLIST`, and the core primitives that implement the stack-based virtual computer.

In the eForth assembly source code, macros are used to create a linked list of the Forth dictionary. The macros package each definition with a sequence of interpreter instructions that keep the chain of threaded execution resolved. These same macros also define the address for each word in the Forth dictionary, and links to the next and previous word.

The address of an executable Forth word (routine) is called the word's *code address*.

A high-level Forth word in memory is, basically, a list of data and/or the code addresses of other Forth words that make up the definition.

The code address, the first address in this list, contains a jump to the Forth inner interpreter, `doLIST`, the mechanism for executing the list of data and code addresses that make up the word being executed.

To boot eForth on an embedded micro, the first instructions after hardware initialization set up the Forth VM by initializing the Forth stack pointer, return pointer, and the inner interpreter pointer. To initiate the Forth machine, the code address of a high-level Forth word is copied into the interpreter pointer register. The code then jumps to the location pointed to by the register, which for every high-level (or *colon*) Forth word, is a jump to `doLIST`.

For an embedded application, the first word may be a control program of some sort. In a resident eForth, the first Forth instruction is `COLD`, which initializes the stack, the TIB, and the terminal I/O, then enters `QUIT`, the familiar interactive Forth interface, also known as the *outer interpreter*.

To use the 56K nucleus in a target compiler system, the name dictionary was removed by modifying the assembly macros. In the target compiler, we don't need it, since name interpretation takes place on the host. Only the code addresses are necessary. Most of the words that make up the high-level Forth model were also removed, leaving the primitives and math words. With the addition of simple communication routines, the eForth nucleus becomes fully usable for remote target development.

The hForth host

The host Forth system used for this project is hForth 86 by Wonyong Koh of South Korea. This is a public-domain, ANS-compliant Forth that runs under MS-DOS. The kernel can be adapted to non-DOS x86 systems and other micros as well. hForth is a derivative of eForth 86, which was the model used for eForth 56—the target kernel.

Any decent Forth can be used, since the target and host do not have to be the same model. For the 68HC11, Pygmy would be a good candidate to try this out on, since the registered version (\$15) comes with a metacompiler, a 68HC11

Figure Two

```
[ 56ke4th.asm line 139] :    doLIST    code address:  A0Fh
[ 56ke4th.asm line 165] :    next       code address:  A16h
[ 56ke4th.asm line 187] :    ?branch   code address:  A28h
```

...etc.

Forth assembler, and serial I/O routines.

The host Forth remote target compiler—HFTCOM56

When the 56K target Forth nucleus is assembled through the Motorola 56000 DOS assembler, the assembly macros write the ASCII names for the Forth words, along with their code addresses, to a text file. (See Figure Two.)

Thus, we have a listing of the 200+ words that make up the operating system/language known as Forth, as well as the target system code addresses for each word.

Send in the clones...

Using the listing from the target56.asm assembly, a Forth compiler can be produced by parsing each line for a name and code address. A host Forth "clone" definition is created with the same name as the target word, and contains the code address in the host definition. Listing One (see next page) shows the essential target compiler.

The target compiler reads the listing file and creates an executable host "clone" of each embedded target nucleus word. Observe the defining word TClone in Listing One. When this word is executed, it creates a host Forth word with a parameter field value that is either a target word code address, or the address for the target HERE—the next available dictionary location in the target. The run-time behavior of the new child word is defined by TCOMPILE? :

- if the host is in *target* compile mode: the clone word compiles its code address into the host memory target image (where a new word is being compiled)
- if the host is in *remote* mode: the clone word transmits its code address to the target for remote execution
- if the host is in *host* mode: the clone word pushes its code address onto the host stack

The host Forth system has also been modified so that stack items entered in the interpreter are either compiled into the host target image or pushed onto the target stack, depending on the state of the remote flag.

To clone the target Forth dictionary from the assembly listing file, each line of the file is parsed into a buffer, to which the string TClone is appended. The buffer is passed to EVALUATE, which performs the normal Forth input stream interpretation.

We now have one leg of the target compiler—a means through TClone to create executable clones of the target nucleus. The next step is to devise a means to create new target routines using the nucleus clones and data in Forth definitions. We again use the defining word TClone and the Forth interpreter to create a target compiler for the target architecture.

Host Forth interpretive "umbilical compiling" to the target

To create new target Forth definitions interactively, which is a foundation of the Forth paradigm, we can co-opt the host Forth interpreter and compiler to generate executable code for the target, even if it has a different architecture.

When a new target definition is created in the host Forth it is *umbilical compiled* to the target transparently. This means that the list of target Forth code addresses or assembly object code that comprises the new definition is linked into the target memory as soon as the definition has been correctly entered. The code is now a part of the target Forth operating system.

T: and T; from Listing One define the rest of the remote target compiler. To create a new target word, the user enters T: (pronounced "t colon") and the desired name for the new word, its definition, and T; ("t semi"), which terminates the compilation sequence.

```
T: MYWORD TARGETWORD1 ... TARGETWORDn ... T;
```

We could have named these routines : and ; because T: and T; reside in a separate dictionary than the Forth dictionary.

A target definition buffer is created in host memory with a Motorola .lod file header and the jmp doLIST opcode. As the programmer types existing target word names into the host terminal interface, they are executed by the host. On execution, they compile their code addresses into the definition buffer.

When the programmer ends his definition (by typing T;), another .lod directive is appended to the host target definition image. The image is then transmitted to the embedded target and becomes part of the target Forth.

A "clone" of the newly defined word is also created at this time, and if the host Forth is not in the target compiling state, the word can be executed on the embedded target by typing its name into the host Forth terminal interface. As more definitions are added, the compiled code is appended to the image buffer. This buffer can be transmitted to the target or saved to a DOS .lod file for later use.

On reset, the target retains the new definitions—this is useful in case the programmer locks up the target. With some additional programming, new definitions could be written into target flash RAM (as on the 56002 EVM)—allowing the target to power-on autostart the new Forth application.

How the target compiler works

Here again is the essential target compiler :

```
: TClone
  CREATE , DOES> TCOMPILE? ;

: T;
  'TEXTIT @ EXECUTE !TCOMPILE DEF>T ;

: T:
  LOCATE! THERE? TCOMPILE TClone
  JMPDOLIST, [ ] ;
```

When T: initiates a new target definition, LOCATE! sets

Listing One

The essential target compiler source code

```
( compile a numeric string into target code space - used to compile 24-bit numbers )
( WHERE $numeric count -- )
: T_,
  DUP 6 SWAP - >R ROT R> +
  ROT ROT
  0 DO DUP C@ ROT 2DUP C! 1+ DUP WHERE! \ copy chars into buffer
  SWAP DROP SWAP 1+ LOOP DROP        \ and inc WHERE for ea char
  DUP BL SWAP C! 1+ WHERE!           \ insert blank after number
  1 THERE +! ;

( compile a 16-bit number into target space )
( target code addr -- )
: T, WHERE? SWAP S>D (d.) T_, ;

\ ( target code addr -- ) execute target address and get response
: T@EXEC BASE @ >R
  HEX
  @ SPUSH          \ fetch target word code pointer and send to target
  TEXEC IN??      \ get ascii response, if any
  R> BASE ! ;    \ restore base

\ if we are in target compile mode, the word code addr is compiled into the defbuff
\ else
\ if we are in REMOTE mode, the word code addr is sent to the target
\ if we are in HOST mode, the word code addr is pushed on the host stack

: TCOMPILE?  TSTATE IF
  @          \ fetch target word code pointer
  T,        \ compile into host target space
ELSE
  REMOTE IF T@EXEC \ send code address to target and execute
  ELSE @         \ else push on host stack
  THEN
THEN ;

\ compile the opcode for jmp doLIST into target space
: JMPDOLIST,  WHERE? 'JMP 2@ (d.) T_, ;

\ the 56k target compiler

: TCLONE CREATE , DOES> TCOMPILE? ;

: T; 'TEXTIT @ EXECUTE !TCOMPILE DEF>T ;

: T: LOCATE! THERE? TCOMPILE TCLONE JMPDOLIST, [ ] ;
```

the host's target buffer pointers. `THERE?` fetches the next available target code address and pushes it on the host stack. `TCOMPILE` sets the host to target compile mode. When the code address and name string (`MYWORD` in the example above) are passed to `TCLONE`, a new host Forth definition is created whose run-time behavior (`TCOMPILE?`) is to transmit its code address to the target, compile it into the host target definition buffer, or push it to the host Forth stack.

At this point, we begin compiling the new target word's definition—the assembly object code that executes on the target. This high-level Forth object code is composed of addresses of other target words or literals (embedded numeric values), arranged as a list to be executed by `doLIST`, the Forth inner interpreter. `JMPDOLIST`, compiles a jump to `doLIST` into the code address for each new word.

[in the `T`: definition puts Forth in interpretation state; that is, it parses the blank-delimited words from the input stream and executes them.

When each clone word entered in a new target definition executes, it compiles its code address into the target definition buffer. That's all there is to it.

The compilation is actually finished before the user types in `T`; — notice that, when the user hits `CR` after entering a sequence of literals or previously defined Forth words, the system is returned to compile mode by the word `] ("r brack")`. `T`; thus appends a code address for `EXIT`, the target Forth primitive that ends a colon definition—the complement to `jmp doLIST`. `!TCOMPILE` turns off target compilation so that `DEF>T` can transmit the new definition to the target.

Now you may be wondering how literals, numeric data entered in definitions, get compiled into target code. I wondered myself, as the project deadline loomed. The host Forth outer interpreter would have to be changed so that it would understand when to target compile numbers. For instance, a definition containing hex numbers :

```
/ strobe the pins on port B
T: STROBE-PORTB
  0 PBD C! FF PBD C ! 0 PBD C! T;
```

The problem is that, normally, Forth would see the numbers, `0h` and `FFh`, and push them on the host stack, when we need them compiled into the target. This level of Forth programming seemed beyond my reach, but luckily I had chosen `hForth`, which offered a simple solution (fortuitously described in the distribution readme file).

Listing Two

```
: targetAlso
  (doubleAlso)          \ the normal numeric interpretation routine
  TSTATE IF             \ if we are in target compile mode...
    'doLIT @ EXECUTE    \ compile target doLIT
    \ doubleAlso leaves 1 on stack if number is a single
    1 = IF T,           \ if single, target compile it
      ELSE              \ if double, target compile it
        WHERE? ROT ROT (d.) T_,
      THEN EXIT
  THEN
  REMOTE IF             \ if we are in remote mode...
    1 = IF SPUSH EOT TX \ if single, push on target stack
      ELSE
        (d.) DPUSH EOT TX \ if double, push on target stack
      THEN EXIT
  THEN DROP ;
```

During interpretation, when Forth encounters a token it first searches the dictionary to find a word match; on failure, it tries to convert it to a number. In `hForth`, the interpretation number-conversion routines are vectored in a table. To add new number types or to change Forth's behavior with numeric input, one merely has to write the handling routine and store its address in the interpretation vector table, called `'doWord`.

Listing Two gives a definition that causes `hForth` to target compile literals.

In a target compiled definition, a literal must end up on the target stack when the routine is being executed by the target kernel. `doLIT` is a target word that extracts a literal from the memory location following `doLIT` and pushes it on the Forth stack.

To store the address of `targetAlso` in the interpretation table, we simply enter:

```
' targetAlso 'doWord 3 CELLS + !
```

How the newly compiled word is remotely executed on the target

At the beginning of the colon word is a jump to `doLIST`, which processes the word by threading through the addresses (other Forth words) in the definition. If the host is not in target compiling mode when a target clone is executed, the clone's code address is transmitted to the target instead of getting compiled into a new definition. The target system has a small monitor routine, affectionately called the *mini-interpreter*, which picks up this code address and passes it to `doLIST` for execution. Since the mini-interpreter is the first address in the threading chain, it is the place execution returns to when the Forth program (starting from the code address, a high-level word) is finished.

Communication between host and target

Target-to-host communication interface

The embedded target, in addition to containing the executable Forth nucleus words and the inner interpreter mechanism, has a small assembly-level routine (the mini-interpreter) for communicating with the host computer. This routine recognizes two commands:

1. push a 24-bit value (data or address) to the target Forth stack
2. execute the target Forth word code address which is on the target Forth stack

This is all that is required for complete host/target communication and control, since the routines for program loading, character I/O, etc. are all part of the target Forth nucleus, and are executed by placing their code addresses on the stack and remotely executing them through the mini-interpreter. (See page 16.)

Listing Four is the mini-interpreter assembly source code.

The mini-interpreter employs a jump table to retrieve the command vector to execute, and recognizes single-byte ASCII commands in the range 10h – 1Fh. This allows for dumb terminal testing of the interface. Currently, three commands are in the table, with room for 13 more. Character I/O is vectored to remove dependence on the serial port for host/target control. The three commands are:

`^T` push data to stack (must be followed by 04h – EOT)

`^U` execute address on stack

`^V` jump to resident Forth

To insert a new command, the address of the desired sub-routine is written into the table. Pressing the corresponding terminal key will then execute the function. Usually, the kernel code would be re-assembled, although it is possible to modify the interpreter while it is RAM.

Host-to-target communication interface

On the host side are the Forth routines to communicate with the target. Given the two commands required by the target, the host Forth can be modified to transmit code addresses and stack elements to the target. The host can emulate target routines by having the host “clones” transmit their code addresses to the target stack and executing them.

Here is the Forth source code for pushing data to the target and executing target routines. TX and RX are the PC serial port character transmit and fetch routines. EOT signals an end of transmission to the target mini-interpreter.

```
\send an ascii numeric string to target
\ ( addr count -- )
: TYPE>T
  ?DUP IF 0 DO DUP C@ TX CHAR+ LOOP THEN DROP
;
```

```
\send a double to target (up to 24 bits)
: D>T (d.) SPILL TYPE>T ;
```

```
\send an ascii string to target
: $>T SPILL COUNT TYPE>T ;
```

```
\send a single to target (up to 16 bits)
: S>T S>D D>T ;
```

```
\push a 16-bit single number to the target stack
: SPUSH ^T TX S>T EOT TX ;
```

```
\push a 24-bit double number to the target stack
: DPUSH ^T TX D>T EOT TX ;
```

```
\execute word on target stack
: TEXEC ^U TX ;
```

From these primitives, the rest of the host-target interaction is established.

A real-time linker?

I didn't have time to write a Forth assembler for the 56K, but wanted to demonstrate that the system could interactively target compile assembly object code. To do this, I came up with a host word `:ASM56` that shells from hForth to DOS, where the assembly module is written and assembled, and then reads the resulting object code file into hForth and links it as a Forth word into the target.

To create an assembly language routine that integrates directly into the target kernel, the user defines a new Forth word with the desired name (e.g., `:ASM56 FIRBLK1.ASM`).

On entering the new definition name, the host Forth:

1. Fetches the current target code pointer.
2. Opens an assembler template file containing macros and hardware equates pertinent to the target system memory map.
3. Copies the template file, and the current target code pointer, to a file named in the definition (e.g., `FIRBLK1.ASM`).
4. Shells to DOS.

At this point, any DOS editor can be used to add code into the new file, which has the necessary pointers for the assembler to locate the code correctly. After the code has been edited, assembled, and converted to a COFF format file (.lod file) without errors, the user exits back to the host Forth (type `c:>exit`).

When the host Forth is returned, it “clones” the assembly object code into the host target dictionary, and transmits the .lod file to the target. The new executable assembly object code exists on the target, and can be executed by typing its name into the host Forth interpreter while it is in `REMOTE` mode. It can also be used with other target Forth words in a new `T:` definition.

This entire process is transparent to the user, with the exception of the editing/assembling loop. `:ASM56` thus performs the duty of a real-time linker by allowing new assembly object code to be located into the target Forth operating system.

Listing Three is a screen dump, with additional comments, of the shell process...

Parting thoughts

To execute filter code on the DSP, the full address register must be available. The 56K nucleus contains a mechanism to push the Forth virtual machine context and enter assembly object code, then return to Forth. Thus, filter routines called as high-level Forth words will run at assembly speed.

The different concepts that made the compiler work are fertile ground for additional embedded tool building using Forth, and are worthwhile topics for the student or researcher.

Listing Three

```
:ASM56 FILTER1.ASM
Now edit your new assembly source file : FILTER1.ASM
...and run it thru asm56000.exe...
press a key to shell to dos

\ at this point edit the file filter1.asm, assemble, and convert to a .lod file

c:> code filter1 \ this batch file runs filter1.asm thru the assembler

c:> fix filter1 \ this runs the strip and lod utilities

c:> exit \ exit DOS to hForth...

Target word name : FILTER1
.ASM file : FILTER1.ASM
.LOD file : FILTER1.LOD

\ FILTER1 is now part of the target vocabulary and can be executed by entering
\ its name into hForth

WORDS

FILTER1 CTOP COLD :CODE RESET GETLINK SETLINK 'BOOT WARMhi hi ?CSP !CSP t.S
tDUMP X Y P PMEM YMEM XMEM ~ t. U. U.R .R PACE EMIT KEY ?KEY D2F STR2FRAC
FRACTABL DECPNT tDECIMAL tHEX FILL MOVE CMOVE @EXECUTE HERE
```

In a professional environment, one would want to opt for a complete development system from a Forth vendor. A problem is that these packages are priced out of the reach of the curious, and therefore limit Forth's exposure. (But they still cost less than comparable embedded C development systems. A typical C cross-compiler costs on the average \$1 - \$2K, and remote source-level debugging tools range from \$750 - \$20K).

In my latest job, I have been using VxWorks, a pricey, embedded UNIX-style operating system running under a UNIX or Windows host. It includes cross-compiling, source-level debugging, and dynamic linking, among other features. The host-side development tools are written in the Tcl scripting language, which has parallels in Forth.

The remote target compiler described in this article shares some of these features: cross-compilation, dynamic linking, and a scripting interface. The target nucleus and mini-interpreter could easily be extended to implement a C or assembly remote source-level debugger for embedded micros, which could be used with the numerous C cross-compilers.

Given Forth's power, the scope of such a project is fairly small. If an embedded C remote source-level debugger written in Forth were made freely available to the masses of microcontroller programmers, Forth could gain greater exposure, and (I hesitate to be so bold) help return Forth to its rightful status as the language of choice for embedded programming.



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

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

TCJ
The Computer Journal

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

Listing Four

Mini-interpreter assembly code

```
;; the mini-interpreter

; init forth virtual machine
;   synchronize with host
;   vm_init : clear sp
;             init forth virtual machine
;
; cmdloop : move #cmdtable, r1
;           inchar?
;           no -> jmp cmdloop
;           yes
;             hi nibble == 1?                ; valid commands 10h-1Fh
;             no -> jmp cmdloop
;             yes
;               mask low nibble -> n1
;               p:(r1+n1),r0
;               jmp r0                        ; execute command
;           jmp cmdloop

; set up forth virtual machine

vm_init    jsr    tack                ; ack forth word executed
           move   #0,sp              ; clear the stack
           move   #SPP,r7            ; init vm
           move   #RPP,r6

; To exit a primitive to NEXT or a colon to EXIT need to have loaded r5, the Forth
; instruction pointer, with address of vm_init - the forth virtual machine
; initialization.

           move   #vm_init,r5

cmdloop    move   #cmdtable,r1
           clr    a

           jsr    getchar            ; character in a0

gotchar    move   a,y0                ; save input character

           move   #>$01,x0            ; test if high nibble == 1
           rep   #4
           lsr   a
           cmp   x0,a
           jne   cmdloop            ; no - not valid character

           move   y0,a                ; yes - move low nibble into n0
           move   #>$000f,x1          ; mask out high nibble for table index
           and   x1,a
           move   a,n1
           nop
```

```

jump          move    p:(r1+n1),r0    ; set up mini-interpreter command jump
              jmp     r0              ; execute mini-interpreter command

              jmp     cmdloop

;; mini-interpreter command routines

; valid commands : (only two needed for forth kernel interface...)

; tpush -          push data to target stack
; execute -       execute address of forth word on target stack

tpush         jsr     indata           ; get up to 6 ascii chars and convert to number
              move    b,x:-(r7)      ; push the number on forth stack
              jmp     cmdloop         ; return to mini-interpreter

execute       move    x:(r7)+,r4      ; pop code address off forth stack
              nop
              jmp     (r4)           ; execute forth word
              jmp     cmdloop         ; return to mini-interpreter

eforth        jmp     eFORTH          ; go start resident eForth

;; mini-interpreter command table

; Command table interpreter jumps on control characters. ^P is a problem
; since it locks up hForth (print command). So we are starting with ^T thru
; ^Z as usable control chars in DOS, which puts the entries at location 4 thru
; 10 ; ascii 14h thru 1Ah. Empty positions in table vector to cmdloop.

cmdtable      dc      cmdloop          ; 0    10h - ^P
              dc      cmdloop          ; 1    11h - ^Q

              dc      cmdloop          ; 2    12h - ^R
              dc      cmdloop          ; 3    13h - ^S
              dc      tpush            ; 4    14h - ^T
              dc      execute          ; 5    15h - ^U

              dc      eforth           ; 6    16h - ^V
              dc      cmdloop          ; 7    ...
              dc      cmdloop          ; 8
              dc      cmdloop          ; 9
              dc      cmdloop          ; a
              dc      cmdloop          ; b
              dc      cmdloop          ; c
              dc      cmdloop          ; d
              dc      cmdloop          ; e
              dc      cmdloop          ; f

```

Mushroom Identification

In 1996, I retired (for the second time) and, with time on my hands, decided to create a computer program to identify mushrooms to the species level. Over the years, I had had some experience designing specialized database programs and was interested in the possibilities of using the power of computers to identify wild mushrooms.

My original plan: obtain a digital data set of mushroom species and their characteristics from the U.S. government, or from university researchers; create a way for a user to enter similar data into a computer; and create a search engine to compare the two data sets. Estimated time to completion: two or three months, tops.

Well, that pipe dream lasted about as long as it took to make a few calls and search the internet. There is no such data set, at least not that I could find. Being brave and more than little stupid, I decided I could create this database. I mean, how hard could it be? Right? I won't bore the readers with the effort it took to research and record identification characteristics for 1,000 species of wild mushrooms, the arbitrary number I had selected. However, I will say that it took months.

The next step was to create the data entry program. The plan was for the user to point and click to describe the unknown mushroom to the program. I had not done any serious programming for the past 12 years, and Windows had passed me by. So, not only did I have to create a serious data set, I also had to play catch-up on Windows programming. Since I have used LMI Forth packages from the early days, I decided to use LMI's WinForth.

WinForth had just about everything I needed, including the hooks into the operating system. But some of the commands to create check boxes and other items in dialog boxes were difficult to use and required considerable trial and error. The result was less than satisfactory. I decided to abandon the use of WinForth to create the dialog boxes and, instead, use Borland's Resource Workshop, which has drag-and-drop and other functions specifically intended to make it easy to create dialog windows and menus.

After the dialogs and menus are created and stored in the .exe file as a static resource, they can be called as needed from WinForth. E. g.,

```
: show_about  
  " about" ['] about_dialog loaddlg drop ;
```

In the example, " about" is the name of the static resource in the .exe file, "about_dialog" is the dialog handler which sets up lists and then, when the dialog is exited, recovers user actions. Loaddlg is the WinForth command to load the dialog.

This combination of WinForth and Borland Resource

Workshop made for a fast and powerful development system. A great-looking dialog window with numerous check boxes, bitmapped graphic buttons, and lists can be created in a matter of a couple of minutes. And this was invaluable, as the program eventually consisted of over 200 bitmaps, 30 menus, and 65 dialogs. Some of the dialogs had more than 20 buttons, checkboxes, lists, and other items.

Because I elected to use some specialized Borland items within the dialogs, it was necessary to load the Borland library at the start of the program:

```
: loadlib DS0  
  " bwcc.dll" asciiz loadlibrary equ borlib ;
```

The command loadlibrary is a WinForth command to call the Windows API to load the library. The variable borlib is used to hold the return so that the library can be released when the program exits, i.e., borlib freelibrary.

The remainder of the programming was pretty straightforward, except for the search routines. Because of the ambiguity of many descriptions (e.g., is the cap brownish red, reddish brown, or perhaps rust brown?) a form of fuzzy logic was used in the search routines. For each attribute, an "importance" and a "nearness" value was assigned. This allowed the program to find the correct mushroom even when some of the data entered were not identical to the data stored in the database.

If I were designing a Windows-based Forth, I would omit all commands to create buttons and other dialog items, and concentrate on building in commands to integrate third-party software and to make it easy to use the Windows API functions. The availability of programs like Borland's Resource Workshop and Microsoft's Help Workshop, which can be used by any programming language, make it unnecessary to reinvent the wheel. The combination of these programs makes it a snap to create sophisticated user interactions.

I learned a great deal during the endeavor and it was well worth the effort, even if it turns out a complete commercial failure. The fun was in the doing.

"I became involved with Forth when I wrote a weather observation database program in BASIC for an early Apple. It was so slow that it was unusable, and Forth was the only other language available. I ported the program to Forth and it ran great. The program was ported to LMI Forth shortly after the IBM PC first appeared.

"The mushroom identification program is my first commercial venture and, while it does not have glitzy graphics, it works and it is the first wild mushroom identification program on the market."

An Extensible User Interface

Almost all the Forth applications I write for research work in mathematics have one or two dozen top-level commands. I use these systems in interactive sessions: data is entered, some commands are invoked, the results are examined, then further commands are issued. New commands may be temporarily introduced during a session; as the research project evolves, new features may also be added permanently. Thus, Forth is used to provide a computing environment which is interactive and an underlying system which is flexible. The research system is extended and modified as it is used.

The present article is the outgrowth of work to prepare Forth systems for use by others. I am interested in showing my research work to other mathematicians and in integrating computer use with some of the pure mathematics courses I teach. In both cases, very few members of my intended audience know anything about computer programming, and essentially none know Forth. My earliest attempts to show my work to others involved providing supplementary written material on Forth. This would allow my applications to be used in essentially the same way I used them (see FORML 90). This approach was successful with some instructional material used in a course in which I was teaching Forth. In general, however, it assumes that people are willing to learn the basics of a computer language in order to use an application (or even to find out if the application interests them). Very few people are willing to do this. I, therefore, wanted to find a simple way to add a user interface to existing applications. There were several criteria for such an interface:

1. It should be easy to use.
2. It should be easy to add to an existing application (without requiring the application to be specially written).
3. It should be easy to extend as the underlying application is extended.
4. It should allow the user to invoke all the top-level commands.
5. It should have an integrated help system.

An interface which satisfies these criteria is described in this article. It is a *commands-completion* interface: The user sees a list of commands. He types enough letters to identify a command uniquely, and the rest of the command name is completed for him. At this point, the command immediately starts execution. The user is prompted for any input needed to carry out the command. Typing the command `INFO` and then another command will provide descriptive information about the command (rather than executing it). The enclosed source code also shows an alternative: type the command name preceded by a question mark.

This interface has solved several problems. For instruc-

tional programs, it has provided students an easy way to interact with an application. They can learn to use it very quickly. It allows mathematical applications to be used when there is no time to teach programming. It allows me to modify and extend an application and interface without recompiling the code of the system. (I turn software over to the computer center at the start of a course and do not have access to it thereafter.) I can also produce optional modules which extend the interface as well as the application.

This interface may also be useful to others to allow Forth work to be shown outside the Forth community. Forth applications usually do not run "standalone." To run an application, a Forth system is needed. This fact puts Forth at a disadvantage with respect to compiled languages. Anyone who wishes to show their Forth applications to those outside the Forth community must usually supply a Forth system with the application or force potential users to obtain one on their own. In some cases, this means they must mess with adapting source code to another version of Forth. The user interface provides an alternative: Elizabeth Rather informs me that Forth, Inc. and other vendors of commercial Forth systems allow their systems to be supplied without fee or license with turnkey applications. A Forth application with this user interface can be "turnkeyed" (i.e., headers removed, one top-level word, and the application saved as an executable). A Forth application can, therefore, be supplied in a trouble-free, load-and-run form just like applications written in compiled languages.

Example

Figure One presents an example showing the interface used for an instructional application in Group Theory. The application computes information about groups of order up to 32 (see FORML 90). In these examples, the user's input is underlined.

Implementation

The menu names of commands are stored in a binary tree together with the execution token of the Forth word needed to carry out the command. When the user types a character, the tree is searched. If a unique entry is found, the command is completed. If no match is found, the system beeps and removes the erroneous letter. If several matches are found, the system waits for further letters.

New commands are added by `>CMD <menu_name> <Forth_word>`. The Forth word must prompt the user for information needed to carry out the command. The group table for group 8, for example, is obtained in the underlying Forth system by "`8 Table`". A new word, `%Table`, is created which contains the help information, prompts for input of a

group number, and executes the underlying Table word. A command is added to the menu by >CMD TABLE %Table. (The commands in the tree are not part of the dictionary, so there is no problem if they have the same name as in the underlying Forth application.) See the source listing for information about the "help" and input words.

The source code that accompanies this article may be downloaded via FTP from:

<ftp.forth.org/pub/Forth/FD/1998>

```

: %Table
  Help:
  This prints a table for the group requested
  (and makes that the current group). Elements
  are represented by letters A to Z and the symbols
  [ \ ] ^ _ and `
  Help;
  Input" for group number \Get-Num " CR Table ;
>CMD TABLE %Table

```

Figure One. Example use of interface.

```

CENTER          CENTRALIZER  CHART           CONJ-CLS
COSETS          EVALUATE     EXAMPLES        GENERATE
GROUP           HELP         INFO            ISOMORPHISM
LEFT            NORMALIZER   ORDERS          PERMGRPS
POWERS          QUIT         RESULT          RIGHT
SEARCH          STOP         SUBGROUPS       TABLE
X

G1>> CHART  Order of Groups (1-32 or 0) Number 12
      20  21  22*  23*  24*
      There are 5 Groups of order 12
      2 abelian and 3 non-abelian

G1>> CHART  Order of Groups (1-32 or 0) Number 6
      7   8*
      There are 2 Groups of order 6
      1 abelian and 1 non-abelian

G1>> TABLE  for group number 8

  _A_B_C_D_E_F_
A |A B C D E F
B |B C A F D E
C |C A B E F D
D |D E F A B C
E |E F D C A B
F |F D E B C A

G8>> INFO
      This will provide information about the next
      command you use. INFO and X do the same thing
      but X is quicker to use.

G8>> EVALUATE
      This is used to evaluate an expression in the current
      group. An expression is a collection of group elements
      and inverses which is evaluated left to right. An
      apostrophe following a letter is used to indicate the
      inverse of the letter. Thus BC'D will give the product
      of B followed by the inverse of C followed by D

```

```
G8>> EVALUATE (use ' for inverse) bd= F
G8>> EVALUATE (use ' for inverse) db= E
```

This system has a sub-menu of commands for permutations:

```
G8>> PERMGRPS
CREATE      ELEMENTS    HELP      INFO
INSTALL    MAIN        MULTIPLY  QUIT
X
```

```
PERM>> ?CREATE
```

This will determine the subgroup of S_n generated by a given set of permutations (given as a product of cycles). You must put in n (for S_n) and then the generators using numbers 1.. n for example (1 2)(3 4 5). The program will only compute groups up to order 51. If the resulting group has order 32 or less, you can install the table as one of the groups 1-5.

```
PERM>> CREATE
```

Subgroup of S_n -- what is n ? Number 4
Put in generators as product of cycles.
End with a blank line

Generator (1 2)(3 4)

Generator (1 2 3 4)

Generator

Group is of order 8

A ()	B (2 4)	C (1 2)(3 4)
D (1 2 3 4)	E (1 3)	F (1 3)(2 4)
G (1 4 3 2)	H (1 4)(2 3)	

Source Code Listing

Supplements to ANS-Forth

1. The words Comment: and Comment; can be defined in a similar way to Help: and Help; below.
2. AT (same as AT-XY) and AT? are used to set and find cursor position.
3. UPC (ch -- ch') converts a character to upper case
UPPER (addr cnt --) converts a string in place
4. DEFER and IS are used for vectored execution
5. (.") is the literal string handler put in place by ."
6. NUMBER? (addr len -- d flag)
flag is TRUE if number was properly converted
d is the double number obtained
7. The following are common:
: 3DUP 2 PICK 2 PICK 2 PICK ;
: -ROT ROT ROT ;
: NOT0= ;
: >= < NOT ;
: CELL 1 CELLS ;
: BEEP 7 EMIT ;
: OFF FALSE SWAP ! ;
: ON TRUE SWAP ! ;

Source Code

```
\ ****      Command Completion Interface      ****
\          John J Wavrik   Dept of Math
\          Univ of Calif - San Diego

          30 CONSTANT Max#Cmds
          16 CONSTANT CmdSize   \ make a multiple of bytes/cell
          0 VALUE   #Cmds
CmdSize CELL + CONSTANT EntrySize

comment:
  A user is presented with a list of commands and needs only
  to type enough letters to identify the command uniquely.

  New commands are introduced by >CMD <listname> <executable>
  where <listname> is the name made available to the user and
  <executable> is a Forth word to be executed. (Typically the
  executable is a Group package Forth command which has been
  supplemented by queries for input).

  The listwords are stored alphabetically in a binary tree
  to enable partial words to be easily found. Each node
  has a name (the list word) which is a string (maxsize SZ),
  and three addresses (cells): the CFA of the executable,
  and the address of left and right subtrees.
comment;

\ ****      Binary Search Tree for Strings      ****

\ Counted String Operations

: $! ( $ addr -- ) OVER C@ 1+ MOVE ; \ no test for fit
: $. ( $ -- ) COUNT TYPE SPACE ;
: $Compare ( $1 $2 -- -1 | 0 | 1 )
  \ -1 = $1 is before $2
  \ 0 = $1 equal to $2
  \ 1 = $1 is after $2
  >R COUNT R> COUNT COMPARE ;

: $< $Compare 0< ;
: $= $Compare 0= ;
: NCompare ( $1 $2 n -- -1 | 0 | 1 )
  \ compare first n characters
  \ must pad strings with blanks if n is big
  ROT 1+ ROT 1+ ROT ( addr1 addr2 n )
  TUCK COMPARE ;

Max#Cmds      CONSTANT #Nodes
CmdSize      CONSTANT SZ      \ maximum string size for names
SZ 3 CELLS + CONSTANT NodeSZ  \ size of node in bytes
0 VALUE FreeNode      \ address of free node variable
VARIABLE Len-Name      \ length of longest name

: $!! ( $ addr -- )
  OVER COUNT Len-Name @ MAX Len-Name ! DROP
  $! ;

CREATE 'Tree1 #Nodes NodeSZ * ALLOT
VARIABLE FreeNode1
CREATE 'Tree2 #Nodes NodeSZ * ALLOT
VARIABLE FreeNode2
```

```

comment:
  In this application there is a main menu (using Tree1)
  and a submenu (using Tree2) activated by a command on
  the main menu. The same idea can be used to allow multiple
  submenus.
comment;

'Tree1 VALUE Tree \ can extend to several trees

: Tree.Init Tree #Nodes NodeSZ * ERASE
  Tree FreeNode ! 1 Len-Name ! ;

\ All operations refer to the "current tree".
\ The address of the root of the current tree is
\ given by Tree. The address of the last filled
\ node is given by FreeNode

: Tree1 'Tree1 TO Tree FreeNode1 TO FreeNode ;
: Tree2 'Tree2 TO Tree FreeNode2 TO FreeNode ;
Tree2 Tree.Init
Tree1 Tree.Init

: NewNode ( -- addr ) NodeSZ FreeNode +!
  FreeNode @ DUP NodeSZ ERASE ;
  \ there is no error trap here if the tree is full

: Left ( n_addr -- l_addr ) SZ + @ ;
: Right ( n_addr -- r_addr ) SZ + 1 CELLS + @ ;
: Exec ( n_addr -- ) SZ + 2 CELLS + @ EXECUTE ;
: Left! ( x n_addr -- ) SZ + ! ;
: Right! ( x n_addr -- ) SZ + 1 CELLS + ! ;
: Exec! ( e_addr n_addr -- ) SZ + 2 CELLS + ! ;
: Name! ( $ n_addr -- ) DUP SZ BLANK $!! ;
: Leaf? ( n_addr -- flag )
  DUP Right 0= SWAP Left 0= AND ;

comment:
  Notice that we assume (and use) the fact that the name
  of a node is stored at the address of the node -- while
  pointers are stored at offsets from this name address.

  Notice also that storing a name (by Name!) pads the
  name with blanks -- to allow use of NCompare
comment;

DEFER (>Tree) \ this allows recursive definition for
  \ storing a new name in the tree
: Go-Left ( $ n_addr -- ) DUP Left
  IF Left (>Tree) ELSE
  SWAP NewNode TUCK Name!
  SWAP Left! THEN ;
: Go-Right ( $ n_addr -- ) DUP Right
  IF Right (>Tree) ELSE
  SWAP NewNode TUCK Name!
  SWAP Right! THEN ;
: (>Tree)-AUX ( $ n_addr -- )
  DUP C@ 0= IF Name! ELSE
  2DUP $Compare
  DUP -1 = IF DROP Go-Left ELSE
  1 = IF Go-Right ELSE
  ( 0 = IF) 2DROP THEN THEN THEN ;
' (>Tree)-AUX IS (>Tree)

```

```

\ Put a new name in the tree -- eventually the execution
\ address will be stored also. Note that this does not
\ store duplicate names.
: >Tree ( $ -- ) Tree (>Tree) ;
\ Given a string $, count n, and node address n_addr
\ Find a node in the subtree with root at n_addr so
\ that the name matches the string up to n characters
: (NFind) ( $ n n_addr -- n'_addr t | f ) DUP 0=
  IF DROP 2DROP FALSE ELSE
    3DUP SWAP NCompare
    DUP -1 = IF DROP Left RECURSE ELSE
      1 = IF Right RECURSE ELSE
        >R 2DROP R> TRUE THEN THEN THEN ;
: NFind? ( $ n n_addr -- t | f )
  (NFind) DUP IF SWAP DROP THEN ;
\ See if a string matches the first n characters of
\ some node in the tree. Indicate if multiple match
: NFind ( $ n -- n_addr -1 | n_addr 1 | f )
  \ -1 = more than one match
  2DUP Tree (NFind) ( $ n addr t | $ n f )
  IF >R 2DUP R@ Left NFind? -ROT
    R@ Right NFind? OR
    R> SWAP IF -1 ELSE 1 THEN
  ELSE 2DROP 0 THEN ;

: Node.L ( node -- ) ?DUP IF 2 SPACES COUNT DROP
  Len-Name @ TYPE
  THEN ;

: CR_4 ( cnt -- cnt' ) ?DUP 0= IF CR 4 THEN 1- ;

: (Print-Nodes) ( cnt tree -- cnt' ) ?DUP
  IF DUP Leaf? NOT
    IF TUCK Left RECURSE
      OVER Node.L CR_4
      SWAP Right RECURSE
    ELSE Node.L CR_4 THEN
  THEN ;

: Print-Nodes CR 3 Tree (Print-Nodes) DROP ;

\ **** Keyboard Input Routines ****

VARIABLE Tfound VARIABLE TAddr
8 CONSTANT BS 7 CONSTANT BELL 27 CONSTANT ESC 127 CONSTANT DEL

\ ClrKey
\ If the user types in more characters than needed
\ to complete a command, this clears the extra characters
\ from the keyboard buffer.

: ClrKey BEGIN KEY? WHILE KEY DROP REPEAT 30 MS ;

\ Del-In Do-ESC
\ The following are actions to be taken by BS or DEL
\ and ESC. n is the number of characters so far in the
\ input word. c is an arbitrary character (it is dropped
\ but included for compatibility with other action words)

: Del-In ( n c -- 0 | n-1 )
  DROP DUP IF 1- BS EMIT SPACE BS
  ELSE BELL THEN EMIT ;

```

```

: Do-ESC ( n c -- )
  DROP TFound ON TAddr OFF
  DUP 0 ?DO 0 Del-In LOOP
  ." *** cancelled *** " CR ;

: Check-Tree ( a n char -- a n+1 ) \ sets tfound
  3DUP EMIT + C! 1+ ( a n+1 )
  OVER 1- ( $ ) OVER NFind
  DUP 1 = ( unique ) IF DROP TFound ON TAddr ! ELSE
  0= ( none ) IF BELL EMIT BS Del-In ELSE
  ( several ) TFound OFF DROP THEN THEN ;

\ Notice that characters from keyboard are uppercased

VARIABLE Help? VARIABLE FirstChar

: TExpect SZ PAD 1+ \ get characters until found in tree
  0 ( len adr 0 ) TFound OFF TAddr OFF FirstChar ON
  BEGIN 2 PICK OVER - ( len adr #so-far #left )
  0<> TFound @ 0= AND

  WHILE KEY UPC ( len addr #so-far char )
    DUP [ CHAR ] ? = FirstChar @ AND
      IF EMIT Help? ON ELSE
    DUP BS = IF Del-In ELSE
    DUP DEL = IF Del-In ELSE
    DUP ESC = IF Do-ESC ELSE
    DUP BL > IF Check-Tree ELSE
    DROP THEN THEN THEN THEN THEN
    FirstChar OFF
  REPEAT DUP 0 ?DO BS EMIT LOOP 2DROP DROP
  ClrKey
  TFound @ IF TAddr @ $. 2 SPACES THEN
  TFound @ 0= ABORT" character count exceeded " ;

: CExpect ( -- )
  TExpect TAddr @
  ?DUP IF Exec THEN ;

\ **** Command Completion Module ****

\ Notice that command names are uppercased

: >CMD ( -- ;; follow by <name><action> )
  #Cmds Max#Cmds >=
  IF ." Command list is full " CR BEEP
  ELSE BL WORD DUP COUNT UPPER
    DUP >Tree
    DUP C@ NFind 1 =
      IF ' SWAP Exec! ELSE
      TRUE ABORT" Error in insertion " THEN
  THEN ;

\ ***** a Help System for Command Words *****

: Make, ( delimiter -- )
  \ Defining word for words that compile input
  \ string up to delimiter.
  CREATE ,
  DOES> @ PARSE HERE >R DUP C, DUP ALLOT
  R> 1+ SWAP MOVE 0 C, ALIGN ;

0 Make, ,0 \ compile entire line as counted string
CHAR " Make, ," \ compile up to a quote

```

```

CHAR \ Make, ,\ \ compile up to a backslash

comment:
  The words Help: and Help; are used to bracket text
  which describes what a command does and/or how it is
  used. This text is put at the start of a definition.
  If the user presses X or type INFO before a command,
  this information is displayed instead of having the
  command action carried out. Help: and Help; should
  be at the start of new lines with the descriptive
  text on lines between (just as "comment:" and "comment;"
  are used to bracket the current paragraph).
comment;

: HelpX 0 Help? ! ;

: Help: ( -<text> Help;>- ) \ the word Help; must start
      \ a new line
      POSTPONE Help? POSTPONE @ POSTPONE IF
      BEGIN >IN @ BL WORD DUP COUNT UPPER
      COUNT S" HELP;" COMPARE 0=
      IF DROP TRUE
      ELSE >IN ! POSTPONE (." ) POSTPONE CR
      REFILL 0=
      THEN
      UNTIL
      POSTPONE HelpX
      POSTPONE CR POSTPONE EXIT POSTPONE THEN ; IMMEDIATE

: %INFO CR
  ." This will provide information about the next" CR
  ." command you use. INFO and X do the same thing" CR
  ." but X is quicker to use." CR
  -1 help? ! ;

\ **** Main Loop ****

: %END
  CR
  ." This will end the command interface (but not the" cr
  ." groups program). You can resume use of the commands" cr
  ." interface by typing 'commands'." cr cr
  ." *** Exit the program by typing `bye` *** " cr
  DROP
  TRUE ABORT" ++++++ " ;

: %Help
  Help:
  This prints a list of all current commands
  Help;
  CR Print-Nodes CR ;

\ Commands
\ This is the top level word used to start
\ the interface

: Commands FALSE %Help
  BEGIN
    CR ." >> "
    [ ' ] CExpect CATCH DROP DUP
  UNTIL DROP ;

```

```

Tree1 Tree.Init
>CMD INFO %INFO          >CMD X    %INFO
>CMD STOP %END           >CMD QUIT %END
>CMD HELP %Help

\ ****  Commands for prompted input  ****

\ Get-TIB
\ This is a word which gets (and edits) keyboard input until terminated by
\ pressing ENTER.  The input must be placed at the start of the terminal
\ input buffer.  The buffer pointer is reset.  The input should be displayed
\ right after the prompt.  When Get-TIB is finished the cursor should be
\ right at the end of input.  ANS standards do not specify the display and
\ editing actions for ACCEPT -- so some systems may require a custom version.

: Get-TIB ( -- )
  AT? QUERY AT          \ Put cursor at end of prompt
  >IN @ 0 WORD          \ Put cursor at end of input
  COUNT TYPE >IN ! ;

\ ****  Samples for prompted input  ****
comment:
  The following prompted input words are included as samples.
  An input word should be designed for each type of data.  It
  should provide a prompt; get an input line (using get-TIB);
  process the line and perhaps check for validity; and leave
  on the stack whatever the action word expects to find.
  Invalid input can either throw an exception or discard the
  invalid input to allow the user to try again.
comment;

2VARIABLE Save-Pos
: Get-Num ( -- n )
  AT? Save-Pos 2!
  BEGIN Save-Pos 2@ AT          \ reposition to start
  Get-TIB BL WORD
  COUNT ?DUP 0= THROW          \ empty input aborts the command
  NUMBER? IF DROP TRUE
  ELSE 2DROP BEEP FALSE \ invalid input starts over
  THEN
  UNTIL ;

\ Fancy input routine

: Pos ( char -- pos ) \ pos = 0 if not found
  >IN @ SWAP PARSE 2DROP
  >IN @ #TIB @ > ( past end of buffer )
  IF 0 ELSE >IN @ THEN
  SWAP >IN ! ;

: Input"
  BEGIN
    [ CHAR] \ Pos
    IF POSTPONE (." ) , \
      BL WORD DUP COUNT UPPER
      FIND 0= ABORT" word not found"
      COMPILE, FALSE
    ELSE
      [ CHAR] " Pos
      POSTPONE (." ) ,"
    THEN
  UNTIL ; IMMEDIATE

```

Runstk – Stack Utility

Ever need more than three things on the stack?
Ever wanted to quickly know for sure what is on the stack while writing or maintaining a word?

Can you remember the stack effects of words you wrote last month or last year?

Do all your word branches have the correct stack outputs?
Do you sleep well?

Well cheer up! Your stack nightmares are over.

Abstract

The Runstk development tool is a text editor that allows the display of the data stack symbolically anywhere within a word shown on the display. It allows the safe virtual running of source code, and can also automatically check cumulative stack effects against the given stack picture for the word. It can put the cumulative stack picture into the source for documentation, and can also show the stack picture of any word in the system.

I have been thinking about writing this article for several years, but was finally motivated by the article in the May-June 97 *Forth Dimensions*. Julian Noble wrote about a simple way to check the stack condition after a series of words executes. His goal was to write a small, portable tool, and he succeeded admirably. Runstk is the other side of the coin, a large, not-too-portable, somewhat complicated piece of code.

However, Runstk does have some advantages:

1. Fast lookup of all previous names, so new names can be chosen uniquely.
2. Consistent, annotated, well-written stack pictures using meaningful symbols can also serve as a quick reference manual for your system during edits—the stack effects of that word you almost remember.
3. For ROM-based systems, you can make sure any word changes will have correct stack operations before the ROM is burned.
4. Check the stack picture at any point in a word and optionally put that picture on a line as a comment.
5. Check the whole word for input and output stack picture correspondence by exercising all branches in a word.
6. Warns if anything is left on the return stack.
7. Could check all the branches in your application for stack effects.
8. Part of this stack picture system is the ability to symbolically run a word while in the editor. This may be useful for education, since stacks seem to bother most new users and may possibly improve Forth acceptance if all stack effects can be seen easily while writing a word—no surprises later.

9. Can handle dual-output stack pictures.

Disadvantages

1. Have to put a stack picture for every word in the source (prototyping).
2. Have to run all of the source through Xref before stack pictures are available for fast use in the editor, but will look up new word stack pictures in the current file.
3. Need large program space to run, since this program itself is over 64K.
4. Has to be integrated into an editor, so not fully portable.
5. Only recognizes ANS stack picture conventions now, so your system may not work with it.
6. Cannot handle dual-input stack pictures (I only had one of these and rewrote it as a single input).
7. Need a fairly fast computer and hard disk for large source projects, since there is a large amount of processing (a 33 MHz 486 is okay).

I am now using Forth for an instrument control program on the IBM PC with a pigtail development system for the R65F12 processor. I know the 6502 is old, but it is still very adequate for distributed processing.

I have had the unusual opportunity to write, change, maintain, and add to this program (among other things) for the last ten years or so in LMI's PCForth+. The stack documenter discussed in this article was a necessary tool to develop to make this process possible and still get some sleep.

How It Works

Runstk needs stack pictures written in a consistent stack picture "language" in order to find the stack pictures and understand them.

General stack pictures:

Variable name (comment about variable) variable init here
non-colon defining word stack picture added by Runstk or Xref

```
: wordname ( N1 N2 -- N3 T | F \ comment )  
\ ANS style + comment  
  other lines of word ;
```

Current stack picture language:

1. The picture is a valid parenthetical comment, as shown above following the word name.
2. Symbols must be separated by white space (e.g., N1 and -- are symbols) to ease the parsing chore.
3. The input data stack side is on the left and the output data stack side is on the right after execution of the word.
4. The execute symbol -- separates the input and output

sides (--- and --> are also allowed).

5. The top of any stack is to the right, as usual.
6. Uses the \ character to separate the stack from any comments—not standard or required. [Figure One]
7. Uses | to separate different output stack possibilities—not required. The true case must be listed leftmost for the branch checker.
: FIND (As -- Ntok 1 | As 0 \ 0= not found)
8. Uses a blank (or ~ symbol) to indicate an empty stack on that side of the execute symbol:
: DROP (N1 --)
if no output, -- may be omitted:
: DROP (N1)
if no input, -- is required:
: HERE (-- Adhere)
9. Uses symbol starting with name to ignore next word in source, which helps show the action of a word such as tick:
: ' (nameword -- Adcfa)
10. Uses anytext" that ends with a " to ignore source up to the next "
: " (text" -- addr nlen)
Words using other text delimiters have to be specially written in Runstk.
11. Uses < > to enclose an expression in which stack item symbols are substituted. Blanks are not required within expressions. Useful to document postfix math into infix expressions; see example below.
: + (N1 N2 -- <N1+N2>)
12. May use S: to indicate data stack:
: DUP (S: N1 -- N1 N1)
13. Pictures with other stack indicators (e.g., C:, F:, and R:) are ignored, so be sure the data stack picture comes first after the word name.
14. Stack picture lookup is case sensitive.
15. Has a few different data types according to the first letter of any symbol (sugar, anyone?). These data types are used to make stack pictures easier to write, but still allow the correct number of items on the internal symbol stacks:
 - D — (or d) double number has two items on the stack, split internally to keep track.
 - S — counted string has address of first character and length on stack, split internally to the ANS *c-addr ulen* style. Lower-case s is still considered a single number, for old code compatibility
 - R — real is handled as a single number for now whether on the data or separate float stack. This will work out if words are carefully written independent from the actual floating-point representation.
 - t — Triple now handled by writing stack picture manually with three items—see S5pic.txt T* (i.e., not commonly used).All other initial letters assume the symbol is a single number.

Stack Picture Examples:

```
: DUP ( N1 -- N1 N1 )
```

Figure One

```
: PICK ( N1 N2 N3 2 -- N1 N2 N3 'N1 0-based stack positions
```

```
: TYPE ( S1 -- )  
OR  
: TYPE ( c-addr ulen -- )  
: COUNT ( A0 -- S1 )  
OR  
: COUNT ( A0 -- c-addr ulen )
```

The Xref companion development tool combs through all of your source code, and saves in a file the name, stack picture, file where defined, and line or screen of the word. All lines in the Xref file are sorted by word name for fast lookup by binary search. This must be redone after every session which changes the stack effects of enough words about which you care for the section where you are working.

The S5pic.txt stack picture file is used by Runstk, has stack pictures of most ANS words, and is included instead of Xref. Made from Rick VanNorman's ANS DPMI-based Forth S4.4 source file. I have modified the stack pictures to work with Runstk but have not verified the correctness of them all.

A decompiler program could add the stack picture and a comment about its source file and line or screen number from the Xref file while in the editor for fast reference. Runstk doesn't decompile, but will show the stack picture of a word.

The Runstk program sets up its own symbolic data and return stacks initialized from the input stack picture of the word being checked. The cursor is within the word to check. Then the stack picture of each word in the body acts in turn on these internal stacks until the cursor or the semicolon at the end of the word is encountered. The internal stacks have each stack item saved as separate symbols; i.e., a double puts two symbols on the internal data stack.

When the cursor is encountered, the stack picture at that point is shown, along with anything left on the return stack. You may elect to insert the stack picture at that point into the source file as a comment. If the cursor is at the end of the word, warnings are also given about stack depth differences compared to the output stack picture for the word, or a non-empty return stack. Dual-output words will make these warnings less definitive, however.

How the Internal Stacks Work

[See Figure Two-a.] Since there is an expression (enclosed in < >) on the output stack, first—for each symbol in the expression—we substitute what is on the data stack in the same position as the input stack position for that symbol. [Figure Two-b]

Then we unify the symbols in the stacks. Take the data stack symbol for each input stack position, and substitute it in the output stack for all occurrences of the input stack symbol in the output stack. This is easier to show than describe. [Figure Two-c]

Finally, pop the number of input stack items off the internal data stack and push the entire output stack on the internal data stack. [Figure Two-d]

This system documents postfix to infix notations very well, especially if symbols are chosen carefully. For example, see

Figure Three.

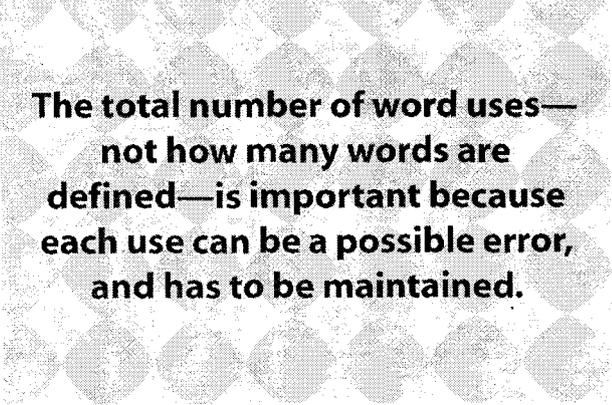
Of course, there are some details not noted above. An asterisk is added to the beginning of all actual symbols on the internal input and output stacks. This hopefully makes the internal symbols unique, to prevent double substitutions.

If the internal symbols were not unique, a word with the stack picture:

```
( N1 N2 -- N2 N1 N2 ) ( TUCK )
```

would come out incorrectly.

Assume an initial data stack as shown in Figure Four-a. After the first unification of N1 from the input stack, the output stack becomes as in Figure Four-b. Then, after the second unification of N2 from the input stack, the output stack becomes that given in Figure Four-c.



**The total number of word uses—
not how many words are
defined—is important because
each use can be a possible error,
and has to be maintained.**

Clearly this is not the correct result after TUCK.

Words that handle the data stack in an enumerated way—such as PICK, ROLL, BURY, NDROP, and NDUP—also have to be specially written, since this stack picture language cannot understand them. Also, Forth-79 and Forth-83 have a different stack-position numbering system.

Words that manipulate the return stack have to be specially written. The return stack is not part of the stack picture language, since it almost always has to be unchanged at the end of a word.

Control structure words also have to be specially written. Words that contain control structures cause Runstk to ask which branch to execute when a word such as IF, UNTIL, WHILE, LOOP, or OF is encountered. Runstk can also automatically check all branches of a word against the expected output. Runstk cannot yet handle the new ANS extended control structures using multiple WHILES, etc.

Other words, such as [CHAR], are also written specially to limit the number of stack picture reserved words.

I have only seen minimal discussion about program size and complexity, for Forth or otherwise. There are programs on the Internet—such as concordance.fth and ftags.fth on Taygeta.com—to analyze Forth projects.

My current system has about 2 megs of source screens, is about 500K compiled, has about 8500 words at last count, 6200 possible branches, 18K lines of code, and 77K word uses.

Compare this to F-PC as delivered, with about 2.5 Mb source and 3000 words, or Win32Forth with 1.7 Mb of source and over 5000 words.

When the world began, only assembler code was available

and a line of code meant one line of assembler containing a single opcode. Today, a line of code is a single, non-empty line of source, no matter how many words are used or defined on that line. This definition has always seemed a little loose to correlate well to program complexity. I count branches and total word uses to get a better feel for complexity.

A branch is a section of code that the program may execute, and relates to the complexity of a program. For example, an IF always has two branches, and a CASE has the number of OFs + 1 branches. Each branch must be checked for stack effects, debugged, and exercised. Obviously, we're not talking about the possible number of branch combinations, which even further increases complexity.

The total number of word uses (TWU), not how many words are defined, is important because each word use can be a possible error, and has to be maintained.

In the past, I was faced with documenting Forth complexity savings for a project in converting a BASIC program to Forth. TWU allowed comparing dissimilar languages somewhat, and also showed that the Forth version was only one-half as complex as the BASIC one, even though more features were added during conversion (of course).

Factoring and reusing sections of code obviously reduces branches and total word uses and, therefore, complexity.

I bought my first Apple][+ in late 1979, with a full 16K RAM, and discovered the joys of BASIC spaghetti code. I discovered Forth through the BYTE Forth issue (Aug. 1980). The language looked interesting, so I bought a copy of 4th from Information Unlimited Software to save typing in the fig-Forth 6502 Model, which I also bought.

I had only programmed a little Fortran in engineering college before this, and didn't feel up to debugging 6502 assembly language yet. After several months of occasional pecking away at this new language, I finally experienced the paradigm shift known as understanding Forth.

This knowledge, along with the knowledge from a book on structured programming by Warnier and Orr, helped land my first professional Forth position in 1983, leading a team that programmed the first Forth versions of Stockpak II and Trendline for Standard & Poors. We used MVP Forth for the Apple II and the new IBM PC, running and developing totally on two floppy disks. (The PC was probably just a fad, its instruction set was so obtuse compared to the RISC-like 6502, but it did have huge 360K floppies.) I have been using Forth ever since for real-time control of instruments.

I am a long time user of LMI's PcForth+ and NMI's RSC-Forth in the R65F12 processor sold by Rockwell. I even used MultiForth on the Amiga for many projects while earning a masters degree with a concentration in artificial intelligence. I have also used other Forths, such as Pygmy, Jforth Amiga, F-PC, LMI WinForth, and MPE Proforth. F-PC was not available when my current project started, and its limited heads space of about 5000 words was too small for the number of words required.

Figure Two-a

Stack Position	4	3	2	1	Top	
Internal Data Stack	E	D	C	B	A	Initial

Next Word stack picture (N1 N2 -- N2 <N1+N2> \ comment)

Next Word stack picture symbols split into

Stack Position	1	Top	
	N1	N2	Input Stack Picture
	N2	<N1+N2>	Output Stack Picture

Figure Two-b

Stack Position	1	Top	
	N1	N2	Input Stack Picture
	B	A	Internal Data Stack
	N2	<N1+N2>	Output Stack Picture Original
	N2	<B+A>	Output Stack Picture with expression Substituted

Figure Two-c

Stack Position	1	Top	
	N1	N2	Input Stack Picture
	B	A	Internal Data Stack
	N2	<B+A>	Output Stack Picture with expression Substituted
	A	<B+A>	Output Stack Picture with symbols unified

Figure Two-d

Stack Position	4	3	2	1	Top	
Internal Data Stack	E	D	C	B	A	Initial
Internal Data Stack	E	D	C	A	<B+A>	After word executes

Figure Three

Example:

```
: N^2 ( N -- <N^2> )
  DUP * ;
```

```
: PI* ( N -- <PI*N> )
  31416 10000 */ ;
```

```
: CYLVOL ( Nlength Nradius -- Nvolume \ all in same units )
  N^2 PI* * ; ( <Nlength*PI*<Nradius^2>>> )
```

Figure Four-a

Stack Position	1	Top	
	N2	N1	notice the non-unique symbols in the data stack and in the input stack picture

Figure Four-b

Stack Position	2	1	Top	
	N2	N2	N2	output stack

Figure Four-c

Stack Position	2	1	Top	
	N1	N1	N1	output stack

Character Literals

A silly thing about the Standard is [CHAR] **x** for the character "x". Eight characters of code to represent one character? When interpreting, we can do it in six characters, **CHAR x**, which is still silly.

Of course, this being Forth, we don't have to put up with this silliness. We can rename and combine [CHAR] and **CHAR**.

But to what? And how?

We don't want to test **STATE** to do it because that would violate some Forthers' religious preferences.

& has been used for this, but it has no intuitive or mnemonic connection. In bed and in the dark, where I do a lot of good stuff, I think **C** would be a good name. It's short for "character" and can be used to replace **CHAR** and [CHAR]. **C#** is almost as good, but some systems already use it for current column number, and besides, it's longer.

Now how do we get **C x** to work compiling and interpreting without our testing **STATE**?

```
: [ LITERAL]
  BASE @ >R  DECIMAL
  0 <# #S #> EVALUATE  R> BASE ! ;
```

This definition of [LITERAL] is for people who think that

testing **STATE** is treason or worse. For those who want to define it the old-fashioned way:

```
: [ LITERAL]
  STATE @ IF  POSTPONE LITERAL THEN ;

: C          CHAR [ LITERAL] ; IMMEDIATE
```

Another useful literal is for control characters. I like **CTRL** for this.

```
: CTRL      CHAR 64 XOR [ LITERAL] ; IMMEDIATE

CTRL G CONSTANT <BELL>
CTRL H CONSTANT <BACKSPACE>
CTRL I CONSTANT <TAB>
CTRL J CONSTANT <LINEFEED>
CTRL M CONSTANT <RETURN>
CTRL ? CONSTANT <DELETE>
```

```
// CTRL | . | @ G H I J M ? \ \
( 0 7 8 9 10 13 127 )
```

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

Wil Baden is a professional programmer with an interest in Forth. For a copy of the source for this article send e-mail requesting Standard Forth Tool Belt #4: Character Literals.

NEW PRODUCT ANNOUNCEMENT

NEW WINDOWS-BASED FORTH DEVELOPMENT SYSTEM SwiftForth Combines Simple Interface to Windows With Top Performance

MANHATTAN BEACH, CA – FORTH, Inc. announces the release of SwiftForth, an extremely fast Forth system that is fully integrated with the Windows 95/NT operating systems and capable of as much real-time performance as is possible in these environments. The system is fully compliant with ANS Forth, but also retains much compatibility with FORTH, Inc.'s successful polyFORTH product line.

A pre-release version of the system has been operational and available on a limited basis to "early adopters" with good prior Forth experience since Fall, 1997. User response has been enthusiastic.

Extensive use of Windows user interface features facilitates all aspects of programming and testing software. The system

provides easy access to all WIN32 functions and all 32-bit DLLs.

SwiftForth's 32-bit subroutine-threaded implementation with direct code expansion yields benchmark times more than three times faster than other popular Windows Forths, and compilation speeds of ~8,000 lines/second on a 200 MHz PC.

The product includes advanced debugging features such as live "watch points" and memory windows. Interactive development features **LOCATE** (source display), **SEE** (code disassembly), hyperlinked source view and cross-reference, and many more convenient features.

FORTH, Inc. also provides custom programming and engineering services, along with a full line of Forth-based development systems for embedded systems and MacOS programming, and the EXPRESS industrial controls package.

FORTH, Inc.

111 N. Sepulveda Blvd., Suite 300

Manhattan Beach, CA 90266

(800) 55-FORTH or (310) 372-8493, FAX (310) 318-7130

forthsales@forth.com • <http://www.forth.com>

Double Number Arithmetic

The Double-Number word set is unusual in that all the words in it can be defined in Core words.

We're not going that far yet. However, we are going to define the fundamental operations of addition and subtraction. We will also define multiplication and division, even though these are not part of the Double-Number word set.

```
1 : +CARRY ( a b -- a+b carry ) DUP >R + DUP R> U< 1 AND ;
2 : -BORROW ( a b -- a-b borrow ) OVER >R - R> OVER U< ( -1 AND ) ;

4 : D+      ( a . b . -- a+b . ) ROT + >R +CARRY R> + ;
5 : D-      ( a . b . -- a-b . ) ROT - >R -BORROW R> - ;
```

These definitions presume 2's complement arithmetic: **1 AND** in the definition of **+CARRY** can be replaced by **NEGATE**. And as shown, **-1 AND** in the definition of **-BORROW** is done for you.

+CARRY and **-BORROW** will be used in division. If **D+** and **D-** are already in your system, then you can define them:

```
: +CARRY ( a b -- a+b carry ) 0 TUCK D+ ;
: -BORROW ( a b -- a-b borrow ) 0 TUCK D- ;
```

Multiplication of two double numbers to get a double number result is a little trickier. It requires three multiplications.

```
7 : D*      ( a . b . -- a*b . )
8   >R SWAP >R      ( a b ) ( R: bhi ahi )
9   2DUP UM* 2SWAP  ( a*b . a b )
10  R> * SWAP R> * + + ( a*b . ) ( R: )
11 ;
```

Those were simple routines, so it was a surprise to discover the complexity of Double Number Division.

DU/MOD (divd . divr . — rem . quot .)

Given an unsigned 2-cell dividend and an unsigned 2-cell divisor, return a 2-cell remainder and a 2-cell quotient.

"Double Unsigned Division with Remainder".

The algorithm is based on Knuth's algorithm in volume 2 of his *Art of Computer Programming*, simplified for two-cell dividend and two-cell divisor.

DU/MOD (divd . divr . — rem . quot .)

Given an unsigned 2-cell dividend and an unsigned 2-cell divisor, return a 2-cell remainder and a 2-cell quotient.

"Double Unsigned Division with Remainder".

The problem in long division can be demonstrated by dividing 99 by 19 in decimal. The first attempt at guessing the quotient is 9 — 1x goes into 9x 9 times. That's too much, so we lower our guess to 8, then 7, then 6, and at last 5.

If we were using 32-bit cells as the base instead of decimal digits, for the equivalent problem FFFFFFFF.FFFFFFFF by 00000001.FFFFFFFF we would reduce our first guess 2 billion times before getting it right.

So we want a way to make our first guess close to the right value.

Knuth shows that if you normalize the divisor and adjust the dividend accordingly, you can make a first guess that's no more than 2 greater than the right guess.

Normalizing the divisor means to make the high-order digit not less than half the base.

For decimal 99/19, we normalize by multiplying top and bottom by 3 or 4. So the problem becomes 297/57 or 396/76 and we guess 5 at once.

With 2's complement arithmetic we normalize by shifting left until the hi-bit is set.

Looking at the code in **DU/MOD** you can see that Forth is not designed for functions like that. Forth is optimized for functions of one or two variables. **DU/MOD** needs two cells for the original dividend, three cells for normalized dividend/remainder, two cells for divisor, one cell for quotient, three cells for quotient-times-divisor, one cell for normalization factor. So it will be messy, even with local variables or in a profane (i.e., not in the temple of Forth) language.

Here's the logic.

Handle case of leading zero in divisor.
 With non-zero leading "digit" in divisor:
 Normalize divisor and dividend.
 Guess leading "digit" of quotient, multiply the normalized divisor by it, and subtract the product from normalized dividend.
 If the result is negative, subtract 1 from quotient and add normalized divisor to dividend; if the result is still negative, do it again.
 Undo normalization of dividend to get remainder.

This routine should work for 16-bit arithmetic or 32-bit arithmetic. It can be modified to work with half-cells to define **UM/MOD** given **/MOD**.

```

13 ( Double Number Division )

15 ( TUM* TUM/ Triple Unsigned Mixed Multiply and Divide )
16 : TUM* ( n . mpr -- t . . ) 2>R R@ UM* 0 2R> UM* D+ ;
17 : TUM/ ( t . . dvr -- n . ) DUP >R UM/MOD R> SWAP >R UM/MOD NIP R> ;

19 ( T+ T- Triple Add and Subtract: t1 . . t2 . . -- t3 . . )
20 : T+ >R ROT >R >R SWAP >R +CARRY 0 R> R> +CARRY D+ R> R> + + ;
21 : T- >R ROT >R >R SWAP >R -BORROW S>D R> R> -BORROW D+ R> R> - + ;

23 : normalize-divisor ( divr . -- divr' . shift )
24   0 >R BEGIN DUP 0< NOT WHILE D2* R> 1+ >R REPEAT R>
25 ;

27 ( DU/MOD Double Unsigned Divide with Remainder )
28 : DU/MOD ( divd . divr . -- rem . quot . )
29   ?DUP 0= IF ( There is a leading zero "digit" in divisor. )
30     >R 0 R@ UM/MOD R> SWAP >R UM/MOD 0 SWAP R>
31   EXIT THEN
32   ( Normalize divisor and dividend. )
33   normalize-divisor DUP >R ROT ROT 2>R ( divd . sh)( R: sh dvr . )
34   1 SWAP LSHIFT TUM* ( divd . . )
35   ( Guess leading "digit" of quotient. )
36   DUP R@ = IF -1 ELSE 2DUP R@ UM/MOD NIP THEN
37   ( Multiply divisor by trial quot and subtract from divd. )
38   2R@ ROT DUP >R ( divd . . dvr . quot)( R: shift dvr . quot)
39   TUM* T- ( divd' . . )
40   ( If negative, decrement quot and add to dividend. )
41   DUP 0< IF R> 1- 2R@ ROT >R 0 T+
42   ( If still negative, do it one more time. )
43   DUP 0< IF R> 1- 2R@ ROT >R 0 T+
44   THEN THEN
45   ( Undo normalization of dividend to get remainder. )
46   R> 2R> 2DROP 1 R> ROT >R LSHIFT TUM/ ( rem . )( R: quot)
47   R> 0 ( rem . quot . )( R: )
48 ;
    
```

Thanks to Martin Laueter for help with **TUM***.

SPONSORS & BENEFACTORS

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

Corporate Sponsors

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

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

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

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

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

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

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

www.kernelforth.com

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

www.theforthsource.com

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

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

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

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

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

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

Individual Benefactors

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

Zvie Liberman
Marty McGowan
Gary S. Nemeth
Marlin Ouverson
Richard C. Wagner



Twentieth Anniversary of the FORML Conference

"Forth Interfaces to the World"

November 20–22, 1998 • Pacific Grove, California

FORML welcomes papers on a variety of Forth-related topics, even those which do not adhere strictly to the published theme. Some theme-related topics of interest, and for which papers are particularly sought, include:

Overcoming the Limits to Growth

Forth in "Foreign" Embedded Environments
(e.g., Windows CE, Inferno, pSOS, Vrtx)

Forth and Rapid Application Development (RAD)

Forth on New 32-bit Embedded Chips

Forth in a Windows World

Co-Existing with C

Forth and the Internet/Java

"20/20: Hindsight and Vision" is planned as a two-part evening panel. Part one will offer a look at Forth's history—what worked well and what might have been done differently—and will feature participants who played key roles in Forth's evolution; part two will evaluate Forth's current status and propose courses of action to lead Forth into a stronger position in coming years.

Among the expected presenters and attendees:

Wil Baden, author of "Stretching Standard Forth" and "Forth Tool Belt" series in *Forth Dimensions*

Everett F. "Skip" Carter, Jr., President of Forth Interest Group, author of *Forth Dimensions* "Forthware" series, President and owner of Taygeta Scientific

John D. Hall, former FIG President, Open Firmware engineer at Apple Computer

Glen Haydon, owner of Mountain View Press, author of *All About Forth*, and Forth philosopher

Charles H. Moore, inventor of Forth

William Ragsdale, founding FIG President and original FIG board member, programmer, entrepreneur, and financial publisher

Elizabeth D. Rather, President of FORTH, Inc. and co-author of *Forth Programmer's Handbook*

C.H. Ting, owner of Offete Enterprises, former FIG board member, custodian of eForth

Conference Chairman: Marlin Ouverson – editor@forth.org

Conference Director: Robert Reiling – ami@best.com

The FORML Conference is held at the Asilomar Conference Center, a National Historic Landmark noted for its wooded grounds just yards from Pacific Ocean dunes and tidepools on California's Monterey Peninsula. Lodging and all meals included with conference registration, and spouses and guests of conference participants can join numerous recreational outings and activities.

Please confirm your attendance early—accommodations may be limited due to this facility's immense popularity.

Call for Papers

Please submit the subject of your paper as soon as possible in order to be included in pre-conference publicity. Final titles with abstracts are due by October 1, 1998. Completed papers should be received by November 1 in order to be included in the conference notebooks that are distributed to all attendees.

E-mail submissions may be sent to editor@forth.org with "FORML paper" in the subject line. Hard copy may be mailed to FORML Conference Chairman, c/o Forth Interest Group, 100 Dolores Street, Suite 183, Carmel, California 93923.

Inquiries about conference registration may be directed to office@forth.org or to the address above.