$10 Volume XVIII, Number 5 January 1997 February

FORTH DIMENSIONS

FORML '96

Zen Floating Point

Garbage Collection

A Stack-Based Dataflow OS

Back to (Object-Oriented) Forth

Towards a Standard Forth Multi-tasker

Contents

Features

Departments

Editorial

Forth Dimensions
Volume XVIII, Number 5
January 1997 February

Published by the
Forth Interest Group

Editor
Marlin Ouverson

Circulation/Order Desk
Trace Carter

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

Subscription to *Forth Dimensions* is included with membership in the Forth Interest Group at $45 per year ($53 Canada/Mexico, $60 overseas air). For membership, change of address, and to submit items for publication, the address is: Forth Interest Group, 100 Dolores Street, suite 183, Carmel, California 93923. Administrative offices: 408-37-FORTH (408-373-6784), Fax: 408-373-2845

The Forth Interest Group

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

"*Forth Dimensions* (ISSN 0884-0822) is published bimonthly for $45/53/60 per year by the Forth Interest Group, 100 Dolores Street, suite 183, Carmel, California 93923. Second-class postage paid at Oakland, CA. POSTMASTER: Send address changes to *Forth Dimensions*, 100 Dolores Street, suite 183, Carmel, California 93923."

It would be remiss of me to let this issue pass without special acknowledgments. It is possible, even probable, that the average reader of *Forth Dimensions* is primarily a subscriber only, one who notes with some casual interest but little personal involvement the affairs of the Forth Interest Group. For example, the growing impact of ANS Forth is becoming more apparent each month but very few will recall that it was conceived at a working group held years ago at the FORML Conference or that it gestated for several years in the hearts and minds of hard-working committee members. And the FIG presence at last year's Embedded Systems Conference was important for both the language and the organization as a whole, but it was made possible by a FIG Chapter in coordination with a Forth vendor.

One thousand-and-one details drive any organization, and the people who manage them are rarely noticed—because they do their job, and more, so well. Not-for-profit businesses, in particular, rely on the good will and generous spirits of volunteers and of paid employees who often end by volunteering more than they are paid. It isn't fair, but it's a fact and a widespread one, at that.

Even as I write this, FIG is completing a peaceful change of guard. John Hall, a long-term member, served as FIG President until recently and then continued in charge of business operations. He dedicated a considerable percentage of his time and his home to our collective, corporate needs. And when FIG needed someone to handle circulation and the mail-order business, he found a solution nearby: Frank Hall was a newly retired postal employee when his brother asked if he would lend a hand. Frank graciously accepted the challenge, adopting FIG and putting in more hours than any part-time, post-retirement position has a right to ask. His help and amiable presence proved invaluable.

We are very grateful to John Hall and Frank Hall for their years of service and friendship to FIG and to its members. They dealt with all the stressful details, political and organizational, that groups like ours tend to generate, enabling the rest of us to enjoy the publications and conferences and meetings with little heed to their work behind the scenes. John and Frank have our collective thanks.

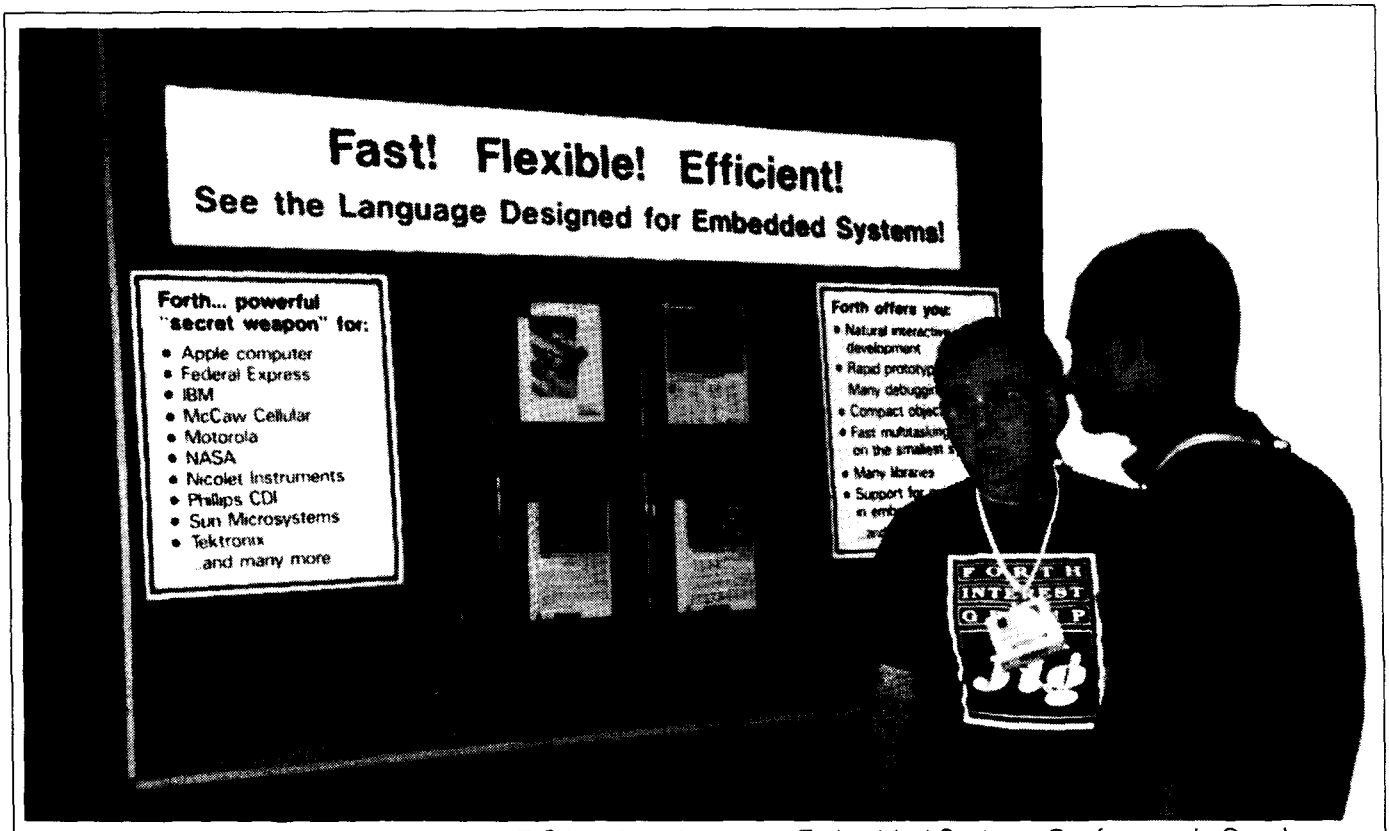
Now the new team is hard at work. Trace Carter and her husband, FIG President Skip Carter, are headquartered near FORML's conference facility, where they have ably shouldered FIG's business operations (in addition to Skip's contributions to this magazine, to the Forth Scientific Library project, and to other endeavors). It is a very real pleasure to welcome them both.

Call for Articles

Like a good conversation, this magazine requires input from every participant. So, once again, we must remind our loyal readers and authors that we need your written contributions. Authors who are more proficient in Forth than in English will receive an editor's friendly assistance, and those who lack confidence in their Forth skills can rest assured that their material will receive technical review before it is published. Anyone intimidated by the thought of writing for *Forth Dimensions* should remember that our readers have varying levels of expertise—for each expert interested in advanced or experimental articles, there is someone else who needs a tutorial on POSTPONE or CATCH and THROW, tips on writing readable code, and explanation of the differences between Forth implementation techniques; plus, everyone enjoys stories about applications, large or small, created in Forth. (See the writer's guide on page six of this issue.)

We always look forward to hearing from our readers, but never more so than when our desks have been cleared and we are planning for upcoming issues. Write soon!

—Marlin Ouverson, editor@forth.org



Doug Hammed entertains a visitor to the FIG booth at the recent Embedded Systems Conference in San Jose, California. The booth, provided by FORTH, Inc., attracted nearly 100 visitors during the three-day conference.

OfficeNews

The physical move of the Business and Sales office of the Forth Interest Group is now complete. We are still in the process of reorganizing and unpacking, but will be operational as you read this.

It's exciting to begin this particular new project. I've been in offices and have worked in administration and management for the last 18 years. My hope is that, with this experience, I can serve the Forth community so that each member will feel that he/she is getting what they need from the main office. The best way to ensure that you get what you need is for you to communicate to me how you feel the office can be of greatest service to you—e.g., member recruitment, member benefits, chapter set-up, job referral, educational literature and, of course, book sales just begin the list.

FIG is a member-supported organization and, as such, its greatest assets are its members. You, as members, make up the heart and soul of FIG and now is the ideal time to make it your FIG. Let me know what we in the main office can do to help. From time to time, I'll write again with "news and updates" from the FIG office to let you know what exciting new things are happening.

So, thank you in advance for your patience as we unpack, for your ideas and action as we grow and change, and for your participation. It's also appropriate to offer a special thank you to John Hall for his long stewardship of the business office, and to Frank Hall for the solid adminstrative foundation that currently exists in the FIG office, on which we can build.

Trace Carter
Administrative Manager
Forth Interest Group Business & Sales Office
100 Dolores Street, Suite 183
Carmel, California 93923 USA
phone: 408-37-FORTH • fax: 408-373-2845
e-mail: office@forth.org

dot-quote

"If you can't crash it, who's driving?"

—Rick Hohensee
from comp.lang.forth

Writing for Forth Dimensions

I once received an author's manual from another technical magazine. It was 26 pages of intimidating restrictions, rules, and regulations. You got the impression that, even if you managed to learn and obey all those constraints, the editors would be doing you a big favor if they deigned to publish your work.

Our guidelines, by contrast, are meant to encourage you. They are minimal, in order to accommodate different kinds of writing (scholarly/academic, anecdotal, philosophical, editorial, how-to, tutorial, etc.) and so as not to discourage anyone from sharing their Forth knowledge and experience.

Being published in the most widely read Forth periodical brings both personal and prfessional benefits. And the Forth community's vitality springs from good ideas, well communicated, circulating through many interesting and informed minds. We encourage you to write, and we hope that, like our other authors, you will be pleased when your article appears and will want to write often.

If you have an idea but aren't sure if it would make an appropriate article, or if you have any other questions, please feel free to contact me. It will be my pleasure to assist in any way I can.

—Marlin Ouverson, Editor
editor@forth.org
c/o Forth Dimensions
100 Dolores Street, Suite 183
Carmel, California 93923 USA

What Makes a Good *FD* Article

We'd like to hear your suggestions. If you *feel a subject is important, interesting, befuddling, or helpful, chances are good that many of your fellow readers of* FD *will agree. Here are just a few ideas from our list of editorial wishes:*

- Tutorials about fundamental Forth techniques like factoring, minimizing stack juggling, and readability.
- Description of interesting details of ANS Forth that differ from previous practice.
- Application stories that show Forth in action. Describe the challenges, tell how you arrived at a solution, and share the final results.
- Classical programming problems demonstrated in Forth (e.g., sorts, filters, date routines) and efficient Forth versions of useful features found in other languages (e.g., strings á la Snobol, *grep* from C).
- Human-interest profiles of successful Forth vendors, corporate users, implementors, and innovators.
- Hardware-related pieces about, for example, Forth chips, embedded systems, robotics, and data acquisition. Such articles are most useful if they include "how-to" info.
- Experimental work that charts new directions for Forth or that addresses perceived deficiencies.
- Work that consolidates previously published material or that builds on or subtantially improves earlier articles.
- Ways to improve Forth's visibility and acceptance.
- Academic papers, including Computer Science perspectives, are welcome although *FD* is not formally refereed.

Tips for New Writers — and getting rid of "blocks"

Every writer in an individual, so what encourages one may inhibit another. But if you can't seem to get started—or to finish—one of these ideas might help.

- Plan your article. Try top-down design and bottom-up writing: Start with your subject, make an outline, and check the logical flow of the information. Write the subsections, then work on smooth transitions between them. Add a motivational introduction and overview, then write a conclusion with observations, suggestions, and a summary.

- Thinking too critically or analytically while writing can dry you up. If this happens, just relax and jot down all your important points without regard for logic, format, spelling, etc. This is the brain-dump phase, and no one but you will see it. Later, simply re-write and organize for clarity, focus, conciseness, organization, and completeness.

- Challenge the reader. When you take a stand or issue a challenge, readers tend to get involved, thinking and acting on their own—a worthy goal of many writers. So provoke readers to improve upon or extend your work, and to test it in their own environments; and encourage them to report their findings to *FD* in an article or letter to the editor.

- Have a beta test. Ask a friend or co-worker to read your article, or present it at a local FIG Chapter meeting to see if it communicates as well as you hope and to elicit useful feedback. Accept advice that makes your code or technique or article better (and tactfully ignore the rest).

- Ask for help. Often there are people at work, at the local FIG Chapter, or at on-line Forth venues who are happy to critique your ideas or even to co-author an article.

- Finally, *learn when to let go.* No one ever feels completely ready to deliver that code or manuscript. Like many an application program, an article is subject to endless revision, refinement, and improvement until it ships. If it communicates well enough, is complete enough, and is accurate, ship it and move on!

What to Include

- ❑ Your name as you wish it to appear in print.
- ❑ Your city, state (or province), and nation of residence.
- ❑ Brief autobiographical information to share with readers—such as education, relevant employment, your introduction to and use of Forth, current projects, and other interests.
- ❑ Your e-mail address for publication, so interested readers can contact you. If you have a personal Web site, you can include that URL as well.
- ❑ If your article depends on a specific Forth dialect or implementation, be sure to specify it (e.g., ANS Forth, Forth-83,
- ❑ Your complete mailing address, so we can mail complimentary contributor's copies of the issue in which your work appears.
- ❑ Permission to post your code, in the form you have sent it to us, on one or more FTP sites so readers can download it; and/or include the URL of an FTP site where you will post the code.
- ❑ Daytime and evening telephone numbers and a fax number (if any). These will *not* be published, and will be used only if we have questions or problems during the editing process.
- ❑ If you send hard-copy manuscript, artwork, or disks that need to be returned to you, include a self-addressed envelope with adequate postage affixed to it.

How to Submit Your Work

If in doubt, just ask—we can work out the details.

E-mail

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

"Snail" mail

Or, mail a hard copy *with* PC or Mac diskettes containing your material to *FD* Editor, c/o the Forth Interest Group.

File formats

If your article includes equations or other items highly dependent on your particular software and/or fonts, include a PostScript file, if submitting electronically, so we can generate a reliable hard copy for proofing. Formatted files that can be read by MS Word are acceptable (e.g., RTF, WordPerfect, and many others); or just send the text as ASCII—you can indicate any critical formatting with informal tags, e.g., <italic>like HTML<end-italic>.

Illustrations

Figures are black-and-white and can be vector (PostScript) or bitmapped. They likely will be re-drawn for clarity, style, and compatibility.

Author Recognition Program

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

Articles

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

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

Letters to the Editor

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

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

Garbage Collection in Forth

Jim Schneider
Campbell, California

One of the most agonizing things about Forth is the lack of decent memory management. The steps taken by the ANS Forth Technical Committee to standardize a memory allocation wordset is good start, but it's only a start. Recently, when I was writing a fairly complex string processing library, I ran into some of the wordset's shortcomings. Since the words could return strings that were in either ALLOTted memory or ALLOCATEd memory, it was very awkward to keep track of which regions of memory needed to be explicitly FREEd.

Although I don't admire LISP's irritating syntactic quirks, I do like the fact that it keeps track of memory items and will discard memory that's no longer needed. I think a Forth garbage collection utility would be invaluable. The code in this article is a first step in that direction.

To use the garbage collection functionality, compile Listing Two into your Forth system. Then, when you want to be able to allocate temporary scratch space and then forget where you put it, invoke START-SCOPE. This will return a single cell scope token which is used to keep track of when to do garbage collection. Allocate and resize away with the functions GC-ALLOC and GC-RESIZE. When you are done, make sure the scope token is on top of the stack, and invoke END-SCOPE. This frees all memory allocated by the GC-ALLOC function since the most recent START-SCOPE.

The program contained in Listing Two is an ANS Forth program requiring the memory allocation and search order wordsets, ALSO and PREVIOUS from the search order extension wordset, and code equivalent to that found in Listing One (which defines the word VOCABULARY).

Note: *Do not* free or resize memory allocated with GC-ALLOC or GC-RESIZE with the standard words FREE or RESIZE! Ignore this warning at your own peril.

Listing One.

```
( vocab.f )
( An ANS FORTH vocabulary mechanism )

: VOCABULARY ( create a named wordlist )
        WORDLIST CREATE , DOES>
        >R GET-ORDER SWAP DROP R> @ SWAP
SET-ORDER ;
```

Glossary

BLOCKLIST (-- a-addr)
A variable that holds the address of a linked list of allocated blocks. The blocks are allocated for use by DRAW-NODE (and its implementation factor EXTEND-FREELIST) in creating and maintaining the free-list.

CUR-SCOPE (-- a-addr)
A variable that holds the address of a linked list of pointers to the memory allocated by GC-ALLOC. The address of the linked list is referred to as a *scope-t* in the listing.

DISCARD-NODE (node-t --)
Returns a node to the free-list. The name comes from an allusion to cards. When you want a new card, you "draw" one. When you want to get rid of a card, you "discard" it. I think this is a much better naming choice than the unfortunate "new" and "delete" that C++ is stuck with.

DRAW-NODE (-- node-t)
No, this won't make a picture of a node on your display device! This takes a node from the free-list, after first making sure that the free-list has some nodes to take.

END-SCOPE (scope-t --)
Recovers all memory allocated via GC-ALLOC since the most recent START-SCOPE. Restores the previous scope designated by *scope-t*.

EXTEND-FREELIST (--)
Adds 3eh (that's 62 decimal) nodes to the free-list, if possible. I chose 62 nodes because several memory allocators are more efficient if the number of bytes allocated is close to, but less than, a power of two. On a 16-bit machine with two-byte cells (e.g., CELL returns 2), this would be 250 bytes allocated at a whack; while on a 32-bit machine with four-byte cells, this would be 500 bytes allocated at once. Notice that there are still three cells left for the allocation routine to use, before the size overflows a power of two.

FREELIST (-- a-addr)
A variable that holds the address of a linked list of two cell nodes. These nodes are used to keep track of allocated memory in all currently active scopes.

Listing Two.

```
( gc.f )
( scope tracking and garbage collection )
( placed in the public domain by Jim Schneider, 17 September 1996 )

VOCABULARY GC-HIDDEN ALSO GC-HIDDEN             ( Avoid namespace clutter )
GET-CURRENT DEFINITIONS VALUE OLD-CURRENT
BASE @ VALUE OLD-BASE HEX

: RESTORE-STATE ( return the system to the original base & search order )
        OLD-BASE ?DUP IF BASE ! 0 TO OLD-BASE THEN
        OLD-CURRENT ?DUP IF SET-CURRENT PREVIOUS 0 TO OLD-CURRENT THEN ;

VARIABLE FREELIST 0 FREELIST !
VARIABLE BLOCKLIST 0 BLOCKLIST !

: XALLOC ( Allocate memory or die )
        ( size -- a-addr )
        ALLOCATE ABORT" Can't allocate memory" ;

: XRESIZE ( Resize an allocation or die )
        ( a-addr \ size -- a-addr' )
        RESIZE
        ABORT" Can't resize memory or attempted to resize a bogus block"
        ;

: EXTEND-FREELIST ( Add 3eh nodes to the free list )
        ( -- )
        7D CELLS XALLOC BLOCKLIST @ OVER ! DUP BLOCKLIST ! CELL+
        FREELIST @ OVER FREELIST ! SWAP 3E 0
        DO
                DUP CELL+ CELL+ OVER ! CELL+ CELL+
        LOOP
        CELL- CELL- ! ;

: DRAW-NODE ( Draw a node from the freelist )
        ( -- node-t )
        FREELIST @ ?DUP 0=
        IF
                EXTEND-FREELIST FREELIST @
        THEN
        DUP @ FREELIST ! 0 OVER ! 0 OVER CELL+ ! ;

: DISCARD-NODE ( Return a node to the freelist )
        ( node-t -- )
        FREELIST @ OVER ! 0 OVER CELL+ ! FREELIST ! ;

VARIABLE CUR-SCOPE 0 CUR-SCOPE !

OLD-CURRENT SET-CURRENT

: START-SCOPE ( Start a scope for garbage collection )
        ( -- scope-t )
        CUR-SCOPE @ 0 CUR-SCOPE ! ;

: GC-ALLOC ( Allocate memory, track it in the current scope )
        ( size -- a-addr )
        DRAW-NODE SWAP CELL+ XALLOC 2DUP ! SWAP 2DUP CELL+ ! CUR-SCOPE @
        OVER ! CUR-SCOPE ! CELL+ ;

: GC-RESIZE ( Resize a previously allocated region of memory )
        ( a-addr \ len -- a-addr' )
        CELL+ SWAP CELL- DUP @ SWAP ROT XRESIZE 2DUP ! DUP ROT CELL+ ! CELL+ ;

: GC-FREE ( Free a previously allocated block )
        ( a-addr -- )
```

(Continued.)

```
        CELL- 0 OVER @ CELL+ ! FREE DROP ;

: END-SCOPE ( End the current scope, discard its storage )
        ( scope-t -- )
        CUR-SCOPE @ SWAP CUR-SCOPE !
        BEGIN
                ?DUP
        WHILE
                DUP CELL+ @ ?DUP IF FREE DROP THEN @
        REPEAT
        ;

RESTORE-STATE
```

GC-ALLOC (size -- a-addr)
Allocates *size* address units of memory, returning its *address*. This is done safely, in that the address of the allocated memory cannot be lost, once it is available. Notice that DRAW-NODE is executed before the XALLOC that allocates the memory. This is because if DRAW-NODE fails, it will abort, and if the address was already on the stack, it would be lost. At most, one node could be lost from the free-list. Even this could be recovered fairly simply, although I haven't written the routine to do so.

GC-FREE (a-addr --)
Frees memory allocated via GC-ALLOC or modified via GC-RESIZE. Although this function isn't really necessary, sometimes it is convenient to be able to free memory immediately, without waiting for garbage collection.

GC-HIDDEN (--)
Changes the search order so that the GC-HIDDEN wordlist is the first in the search order. This is included because I really hate cluttering up my main wordlist. As it is written, Listing Two only adds five words to the CURRENT wordlist, out of a total of 16 words. I tend to use a lot of "HIDDEN" vocabularies for my implementation details. If your system already has a HIDDEN vocabulary, remove the definition of GC-HIDDEN, and replace all occurrences of it with HIDDEN. If you do this, Listing Two will only add four words to the CURRENT wordlist.

GC-RESIZE (a-addr \ size -- a-addr')
Resizes a previously allocated region of memory. Note that the address passed to GC-RESIZE must have been previously returned from either GC-ALLOC or GC-RESIZE. GC-RESIZE may be used on any block of memory returned from GC-ALLOC or GC-RESIZE, as long as it is still valid. This means you can do this:

```
START-SCOPE 1000 GC-ALLOC
START-SCOPE SWAP 2000 GC-RESIZE SWAP
END-SCOPE
```

and expect it to work.

OLD-BASE (-- x)
A value used to contain the value of the variable BASE at the beginning of Listing Two. This is here because I hate gratuitous base bashing, and the Forth system I use most often complains if I put something on the stack during file loading and don't remove it on the same line. It doesn't cause an error, but it clutters the screen. Those of you with a HIDDEN vocabulary might want to create this value there once and for all, and just assign numbers to it periodically (i.e., change line six of Listing Two to: BASE @ TO OLD-BASE HEX).

OLD-CURRENT (-- x)
A value used to contain the result of GET-CURRENT at the beginning of Listing Two. See the remarks under OLD-BASE. I also hate gratuitous search order trashing.

RESTORE-STATE (--)
Restores the previous search order and base.

START-SCOPE (-- scope-t)
Starts a new scope for memory allocation tracking. Returns a token describing the enclosing scope (actually a pointer to a linked list of pointers to allocated blocks).

XALLOC (size -- a-addr)
Attempts to allocate *size* bytes of memory from the system. Raises an exception (via ABORT") if this isn't possible. I use this because I often don't care why I can't allocate memory, but the only reasonable response to a failed request is to abort. If your system provides the exception wordlist and the ior returned by ALLOCATE is meaningful, the ABORT" should be replaced by THROW.

XRESIZE (a-addr \ size -- a-addr')
Attempts to resize a previously allocated block to *size* bytes. Raises an exception (via ABORT") if this isn't possible. There are two possible causes for failure: There isn't enough memory (in which case an abort is usually appropriate) or the address wasn't allocated via ALLOCATE or RESIZE (in which case there's a bug, and an abort is definitely appropriate). As above, if your system provides the exception wordlist and the ior returned by RESIZE is meaningful, the ABORT" should be replaced by THROW.

Downloading: Code available via FTP from: ftp.netcom.com/pub/ja/japs

Jim Schneider — japs@netcom.com — has been programming in Forth for about half his life. He wrote the assembler distributed with Andrew McKewan and Tom Zimmer's 32-bit Forth for Windows 95 and NT. He is currently doing virus research for a major AV and security products vendor.

Back to Forth

An Object-Oriented Forth

Anatole L. Medyntsev
St. Petersburg, Russia

1. Preface

This article describes an experimental object-oriented Forth system for MS Windows. Certain ideas implemented in the system are based on the author's experience in FoxPro 2.5, Visual C++, MS Access, Visual BASIC, and Delphi for database development in the financial field. In my opinion, a combination of an interpreted object-oriented language and the usual C or Pascal is the best environment for database applications development. Certainly, many languages can be used (and are used now) as that interpreted language (Smalltalk, Visual BASIC, or Prolog, for instance). But, unlike others, Forth is not a monolithic language (many basic elements of the language can be changed at the user level), and it provides a simpler, more flexible, and more open technology.

1.1 The tool we need for database development.

In short, the tool we need would be similar to Delphi or Visual BASIC, but with such dynamic features as run-time macro substitution, late binding, etc. Why are the dynamic mechanisms necessary? For example:

1. A real business application has many options (parameters) which can be modified by a user. These options are usually saved in files (databases). They might be not only simple numeric or string values, but more complex values such as arithmetic expression, for instance.
2. In some cases, the ability to save the names of called procedures, or even the procedures themselves, in a database is very useful (for instance, data-validation rules). Besides, this ability is necessary in order to add object-oriented features to the usual relational database.
3. Moreover, an experienced application user should be allowed to modify the code or to add simple routines. Certainly, the core of any system should be protected from user errors; however, some programming facilities (like BASIC in MS Word and MS Excel) should be present in many applications at the user level.
4. Context-dependent behavior: Sometimes, the behavior of an object (instance) should depend on the environment in which the *object* is created (unlike the traditional approach, where it depends only on the environment where the object's *class* is defined). Late binding is necessary to support this feature.
5. Run-time class generation: Usually, special classes (for instance, DBTable and DBField in Delphi) are used to provide database access. If a database structure is unknown at design time, it is impossible to define the appropriate classes (because the field descriptions are unknown). In this case, the class for the database should be generated automatically at run time.

Certainly, features like these can be implemented in classic languages. For instance, some dynamic facilities are accessible in Delphi through a special directive ("published") which provides run-time information (the names of methods and properties) so that the methods and fields of any object can be accessed by name at run time. But it is not sufficient. I think the Forth architecture is potentially more suitable in such cases.

1.2 What changes are necessary?

The situation in which Forth is used as a shell for a high-level operating system environment (MS Windows, for instance) differs cardinally from the usual situation, where Forth is used as a portable shell for a low-level, hardware-closed environment. So, many words from the standard "standalone" vocabulary are unnecessary. But many new words are necessary to work in an event-handled environment such as MS Windows. (It would be better if the "operating system" vocabulary were portable among various operating systems, in the same way the "standalone" vocabulary is portable among various processor architectures. But this problem is beyond the scope of this article.)

The following features should be present, too:

- object-oriented possibilities;
- run-time type checking (like Visual BASIC). This is very useful for automatic typecasting, for instance;
- no application crashes (general protection errors, etc.) are allowed. The user should be protected from such errors;
- Forth programs are not very readable because of the standard Forth parameter-passing mechanism, which

makes the flow of data invisible. It causes dumb errors. Thus, local variables and explicit parameter-passing (as, for instance, in C, Pascal, BASIC, etc.) should be implemented.

2. The object-oriented Forth system: basic mechanisms and language description.

This system does not follow any Forth standards. In fact, there are two unusual, basic mechanisms:

- vocabularies are replaced by modules;
- run-time type checking.

In addition, the system provides two levels of syntax:

- At the system level, standard Forth syntax is used (parameters are passed only via the stack).
- At the user level, there are local variables and explicit parameter passing, run-time stack-checking, and the traditional syntax for expressions (no reverse Polish notation).

2.1 Modules, classes, and vocabularies.

Modules are the basic elements of this Forth system. They are like C++ or Object Pascal classes. In fact, there is no difference between a module and a class in the Forth system. (I use "class" and "module" as synonyms here.)

A module consists of:

1. Entry-points table—analogous to a Virtual Method Table (VMT) and the classic Forth vocabulary. Each element of the table has a fixed format and contains:
   - the name of the word;
   - a pointer to the handler procedure which processes messages;
   - a reference to the threaded code (for colon definitions) or to memory (for variables);
   - any additional parameters required for the handler. (The generated threaded code contains indexes, not addresses, of elements in the entry-points tables.)
2. Static variables and bodies of colon definitions. Indirect-threaded code is used.
3. A module can contain a constructor and a destructor (optional). The constructor is executed when a new instance of the module is generated. (Destructors are not yet implemented.)
4. A module (class) can contain a reference to its parent class (only one parent class is allowed). All words (i.e., procedures and variables) are virtual and can be redefined in a child class (no private, public, or protected directives are yet implemented).

Each module (class) is compiled as a separate unit. Not only backward references, as in standard Forth, but also forward references are allowed inside a module. All words in a module can be accessed only when the module is compiled and an instance of the module is created.

2.2 Instances. The dynamic context tree.

An instance of a class contains a header and memory for non-static variables. The gen function creates an instance of a module (equivalent to "new" in C++). When an instance is created, memory for the header and non-static objects is allocated, and the constructor is executed. Multiple instances of the same module are allowed.

The global variable `CurrentContext` points to the instance from which a word name's search starts. Normally, `CurrentContext` contains reference to the last-created instance of a module. However, it is possible to assign any instance reference to `CurrentContext`. Each instance has a parent instance, which is defined by the value of `CurrentContext` (when the instance is created). Thus, there is a tree of module instances. The tree is used for searching, as in standard Forth. However, unlike the usual, static dictionary tree, this tree of instances is created dynamically. This feature, with the late-binding mechanism, supports the context-dependent behavior of objects.

2.3 Names scope. Binding.

There are three sorts of names:

- global (public)
  These are contained in a special global, entry-point table. They are visible from everywhere. The special word registration is used to add new names to the table (compile-time binding).
- local (private)
  These are defined in the same module (compile-time binding).
- dynamic
  These are searched through the dynamic context tree at run-time (late binding). (Syntactically, the first letter of a dynamic name is capitalized, but the first letter of a non-dynamic name is not.)

2.4 Data types. Stack structure.

The system supports run-time data type checking. Run-time automatic type conversion is supported, too.

The set of types includes: char, int, long, var, string, procedure, object (instance), char*, int*, long*, var*; numeric types are the same as in C. *Var* is like Variant in Visual BASIC, its structure is the same as the structure of the element of the stack (see below); *object* refers to an instance; *procedure* refers to any word in some instance; the name of a module can be used as a type, too.

An element on the stack has the following structure:

word type	type of data;
word count	for a pointer, determines the limit up to which the pointer may be advanced (i.e., the amount of memory before reaching the end of the memory area);
long value	holds the whole object (if it fits), or a pointer to the object.

2.5 Pointers.

Pointers are typed. A pointer structure contains not only the address, but size, too. Only special words can be used to work with pointers.

```
>>  ( pointer n -- pointer )
```

Move the pointer by *n* positions.

```
<<< ( pointer -- pointer )
```
Set backward direction for moving the pointer.

```
>>> ( pointer -- pointer )
```
Set forward direction.

Arithmetic operations cannot be used with pointers. Thus, it is impossible for a user to have illegal access to memory.

2.6 Examples

```
10 ints A
```
Define array, named A, of 10 integers.

```
A 2 >> @
```
Get the third element of array A.

```
A <<<  @
```
Get the last (tenth) element of array A.

```
A <<<   1 >>  @
```
Get the ninth element of array A (i.e., the second element from the end of the array in the backward direction).

```
A <<<  dup @  swap >>> +!
```
Adds the value of the last element of A to the first element.

2.7 Method invocation.

A method of any class can be invoked with the words -> and =>. Static invocation (compile-time binding):

```
<object> -> <name of a method>
```

Dynamic invocation (dynamic searching by name):

```
<object> => <name of a method>
```

2.8 Small examples (extended syntax).

In the examples in Figure One (page 14), the word , (comma) is used to mark the end of an expression, and the word ` (tick) is used to mark a function call.

(I use an extended "user-level" syntax in these examples; some standard Forth words are not available on this level—, for instance. Thus, I use the same names, but the semantics is different.)

3. Implementation remarks.

This system is implemented in C. By virtue of this, new words implemented in C or Pascal—procedures from any static or dynamic (DLL) library, for instance—can be added easily to the system dictionary. Only small C functions should be implemented in the Forth system environment to call a function from any library.

The current version includes:

- Forth classes to work with Windows (implemented),
- classes to work with databases (implemented partially),
- DDE support (not finished), and
- interactive debugger (draft version, not finished).

The size of the .EXE file (with Windows and database support) is about 250K. The source code of the core of the system (without Windows and databases support) contains about 5000 lines in C and about 500 lines in Forth.

Certainly, the C version of Forth is not very fast—about 30,000 steps/sec (on a '486 DX, 50 MHz), sufficient to work with databases and to operate with Windows through MS Windows API.

4. Discussion.

Some areas where this Forth system is potentially useful are described here.

4.1 Forth as the database-server language.

At present, many database servers (such as Oracle, Sybase, Microsoft SQL Server, and InterBase) are based on using SQL as the language for client/server conversation. Universal APIs such as ODBC (from Microsoft) and IDAPI (from Borland) support SQL technology, too, but I think there are some restrictions in the approach:

1. The power of SQL is not sufficient, and some procedural extensions are usually added to SQL by the server producer. I think that architecture is fairly odd. In my opinion, the approach where SQL is implemented as an extension of any powerful procedural language is more suitable (as it is in FoxPro, for instance).

   Moreover, in many cases, not only a database server but an application-oriented server is necessary. This means that, not SQL, but an application-dependent language (or remote procedures) should be used for client/server interaction so that the server can deal not only with database files, but with the entire operating system environment (such as COM ports, for instance). This is impossible without a powerful, procedural server language.

2. Not only relational databases, but network databases (db_Vista from RAIMA Co., for instance) are used now. Moreover, in many cases, the network paradigm is used to work with relational databases (where a unique automatically generated key, like ROWID in Oracle, is used to address records). SQL is often fairly good at working with the network representation (for large report generation, for instance), but I think the native, pointer-oriented approach should be considered as the basic level in this case, and this basic level should be available to client applications.

3. The default optimization strategy of an SQL server is often fairly good. But, sometimes, explicit optimization of SQL requests is required, too. It is necessary to understand the very complex basic optimization mechanisms of the SQL server to optimize SQL requests, and this is not easy. I think the navigation (xBase) approach provides a simpler way for explicit optimization.

4. Logically, there are no differences between database access and interprocess communication (in the same way that, in Prolog there is no difference between the database-based stream and the procedure-generated

(Text continues on page 26.)

Figure One. (See section 2.8 for syntax notes.)

Factorial

```
module test
  // factorial
  // ":" - in a procedure header separates input and output parameters
  : fact (in:out){
        loc in1
        in if{
              in1=in-1,
            ` fact(in1:out)
              out=out*in,
           }else{
              out=1,
           }endif
  };
  // constructor, () - no input or output parameters
 : init(){ loc out ' fact(8:out) MsgBox(out) };
end
// Create an instance of last compiled module
genLast
```

Rectangle drawing

```
// Point and Rectangle class definitions ;

  class Point
        base RootClass // parent class reference (RootClass - default ancestor)
        int X
        int Y
       //Cleaning the variables
     : init(){ X=0, Y=0, };
   end
   class Rectangle
        base Root
        Point TopLeft
        Point BottomRight
   : Draw(){
        // Draw rectangle
   `  Line(TopLeft->X, TopLeft->Y, BottomRight->X, TopLeft->Y )
   `  Line(BottomRight->X, TopLeft->Y, BottomRight->X,BottomRight->Y)
   `  Line(BottomRight->X,BottomRight->Y,TopLeft->X,BottomRight->Y)
   `  Line(TopLeft->X,BottomRight->Y,TopLeft->X,TopLeft->Y)
    };
    end
    module test
     Rectangle rect
       //Coordinates setting
     : init(){ rect->BottomRight->X = 9 ,
               rect->BottomRight->Y = 25 ,
               rect->Draw //Draw rectangle
      };
    end
    // Create an instance of the last compiled module
    genLast
```

Zen Floating Point

C.H. Ting
San Mateo, California

When I was young and innocent, I learned programming in Fortran. The world was simple. You operated a punch card machine, submitted a card deck to the computing center, and waited for the printout. All numbers were floating unless they were prefixed with I-M. All operations were floating, unless the compiler decided otherwise.

Then came mini-computers. The simple world was turned upside-down. All numbers were integers, without exception. To get back to the more familiar floating-point numbers, you had to buy a big and slow Fortran compiler, which generally did not work very well. Or you could spend $50,000 to buy a floating-point processor and hook it up to your mini. You would be lucky if the system actually worked.

Now, you buy a '486 PC and a floating-point unit (FPU) is included free of charge. I am much older and, hopefully, wiser since learning Forth. What can I do with the FPU?

If we strip out the register-addressing instructions, a floating-point package can be built very simply and elegantly.

The most obvious approach is to have a simple floating-point package in Forth which allows me to access the FPU interactively.

The '486 FPU was designed to be used in Forth, because the floating-point registers therein are arranged as a stack. The floating-point operations merge naturally into the Forth framework. The '486 FPU instruction set is unnecessarily complicated because Intel designers allowed the registers to be accessed as a regular register set. If we strip out these register-addressing instructions, a floating-point package can be built very simply and elegantly.

Most of the FPU machine instructions can be easily transcribed into simple Forth code words. We do not even need a floating-point instruction assembler, although it is not difficult to implement by extending the 80x86 Forth assembler, as done in the HFLOAT package in F-PC. The FPU instructions are simple enough that we can enter them into the code definitions using their binary codes. Here we define a special defining word FPU to take care of all these simple cases. Only a few special cases need to be handled explicitly in assembly code.

Martin Tracy published a floating-point package in his ZenForth to provide some elementary floating-point capability to an integer Forth system. This ZFPC package follows his design concept, but gives the user the full power of an industry standard floating-point processor.

Limiting our access to the top two numbers on the floating-point stack, the 80x87 instruction set can be greatly simplified. This implementation gives you all the power of 80x87 with very simple Forth syntax. You can use it interactively as a powerful, high-precision scientific calculator, or write elaborate programs to do scientific and engineering computations.

This package is aligned to the optional floating-point wordset in ANS Forth.

Code begins on next page.

Dr. C.H. Ting has long been a noteworthy figure in the Forth community. He has been very active in the Silicon Valley FIG Chapter and owns Offete Enterprises.

```
empty clear_all_labels
create fBuffer 100 allot
2variable dTemp
hex

: FPU ( b1 b2 -- , defining word to create FPU code words)
    assembler CODE SWAP C, C,
    NEXT END-CODE
    ;

D9 E0 FPU FCHS ( F: r -- -r , change sign )
D9 E1 FPU FABS ( F: r -- |r| , absolute value )
D9 EE FPU FLDZ ( F: -- 0.0 , load 0.0 )
D9 E8 FPU FLD1 ( F: -- 1.0 , load 1.0 )
D9 E9 FPU FLDL2T ( F: -- log2/10/ , load log base 2 of 10 )
D9 EA FPU FLDL2E ( F: -- log2/e/, load log base 2 of e )
D9 EB FPU FLDPI ( F: -- pi , load pi )
D9 EC FPU FLDLG2 ( F: -- log10/2/, load log base 10 of 2 )
D9 EE FPU FLDLN2 ( F: -- loge/2/, load log base e of 2 )
D9 EA FPU FSQRT ( F: r -- \/r , square root )
DE C1 F+ ( F: r1 r2 -- r1+f2 , add )
DE C9 F* ( F: r1 r2 -- r1*f2 , multiply )
DE E9 F- ( F: r1 r2 -- r1-f2 , subtract )
DE CE F-R ( F: r1 r2 -- r2-r1 , reversed subtract )
DE F9 F/ ( F: r1 r2 -- r1/f2 , divide )
DE F1 F/R ( F: r1 r2 -- r2/r1 , reversed divide )
D9 C0 FDUP ( F: r1 -- r1 r1 )
D9 D9 FDROP ( F: r --  )
D9 C9 FSWAP ( F: r1 r2 -- r2 r1 )
D9 C1 FOVER ( F: r1 r2 -- r1 r2 r1 )
D9 FF FCOS ( F: r -- cos[r] )
D9 FE FSIN ( F: r -- sin[r] )
D9 FB FSINCOS ( F: r -- cos sin )
D9 F2 FPTAN ( F: r -- y x , y/x=tan r )
D9 F3 FPATAN ( F: y x -- r , r=arctan y/x )
D9 FD FSCALE ( F: r1 r2 -- r1 r2*2**r1 , scale r2 by 2**r1 )
DB E3 (FINIT) ( -- , hardware initiation )

\ miscellaneous operations

code F0< ( f: f -- f, -- flag )
    D9 c,   E4 c,       \ FTST
    DF c,   E0 c,       \ FSTSW AX
    and ax, 4700 #  \ get condition
    cmp ax,  100 #  \ F0< ?
    jz  1 $
    xor ax, ax
    push    ax
    next
1 $:    mov ax, -1 #
    push    ax
    next    end-code

code F< ( f: f1 f2 -- f1 f2; -- flag )
    D8 c,   D1 c,       \ FCOM ST(1)
    DF c,   E0 c,       \ FSTSW AX
```

```
    and ax, 4700 #  \ get condition
    cmp ax,  100 #  \ F< ?
    jz  2 $
    xor ax, ax
    push    ax
    next
2 $:    mov ax, -1 #
    push    ax
    next    end-code


code F0= ( f: f -- f; -- flag )
    D9 c,   E4 c,       \ FTST
    DF c,   E0 c,       \ FSTSW AX
    and ax, 4700 #  \ get condition
    cmp ax, 4000 #  \ F0= ?
    jz  3 $
    xor ax, ax
    push    ax
    next
3 $:    mov ax, -1 #
    push    ax
    next    end-code

code F@ ( f: -- f; a -- )   \ Intel temp-real 80 bits
    pop     bx
    DB c,   2F c,       \ FLD 0[BX]
    next    end-code

code F! ( f: f -- ; a -- )  \ Intel temp-real 80 bits
    pop     bx
    DB c,   3F c,       \ FSTP 0[BX]
    next    end-code

code (D>F) ( f: -- f )
    DB c,   06 c,       \ FILD long-integer
    dTemp ,
    next    end-code

: D>F ( d -- ; f: -- f )
    swap            \ reorder bytes
    dTemp 2! (D>F) ;

code (F>D) ( f: f -- )
    DB c,   16 c,       \ FIST long-integer
    dTemp ,
    next    end-code

: F>D ( -- d ; f: f -- )
    (F>D) dTemp 2@
    swap ;          \ reorder bytes

\ floating-point number input and output

decimal
variable fsign
variable fexponent
variable fdigits
```

(Continued.)

```
: PRECISION ( -- n )     fdigits @ ;
: SET-PRECISION ( n -- )     fdigits ! ;

5 set-precision

: justify ( f: 10.0 f1 - 10.0 f2, 1.0<=f2<10.0 )
    F0< if FABS -1 else 0 then fsign !
    0 fexponent !
    F<
    if  begin   FOVER F* F<
        while   -1 fexponent +!
        repeat
        FOVER F/
    else    begin   FOVER F/ F<
        1 fexponent +!
        until
    then ;

: F. ( f: f -- )
    F0=
    if ."  0.0" FDROP exit then
    10. D>F FSWAP
    justify
    fsign @
    if ." -" else space then
    fdigits @ 0
    do FOVER F* loop
    F>D <# fdigits @ 0 do # loop ASCII . hold #S #> type
    fexponent @ ?dup
    if ." E"  1 .R then
    FDROP FDROP ;

: FLOAT ( d --; f: -- f )    \ 0.001234, 1234.5678 as input
    10. D>F          \ base
    D>F
    DPL @ 0
    do FOVER F/ loop
    FSWAP FDROP ;

code FSAVE ( -- )
    DD c,   36 c,        \ FSAVE fBuffer
    fBuffer ,
    next    end-code

code FRSTOR ( -- )
    DD c,   3E c,        \ FRSTOR fBuffer
    fBuffer ,
    next    end-code

: FDUMP ( -- )
    FSAVE CR
    fBuffer 14 +
    8 0 do  dup F@ F.
        10 +
    loop    drop
    FRSTOR ;

: FINIT (FINIT)
    8 0 do  0. float loop ;
```

A Stack-Based Dataflow Operating System

Barry Kauler
Joondalup, WA, Australia

Dataflow and CASE

For a couple of years now, I've been using a programming language, called *LabView*, that is a 100% visual dataflow programming environment for Windows machines, Macs, and Sun Workstations. A "hello world" program developed in LabView won't even fit on a 1.44 Mb floppy disk, and it runs incredibly slowly. It does, however, offer a high level of programmer productivity, and can be used by people who know little about conventional text-based languages. It was the visual *dataflow* paradigm that greatly intrigued me and started me on an investigation of dataflow software design techniques.

Moving down to a much lower level, I became interested in operating systems for embedded applications, that mesh better with *Structured Analysis* CASE tools, in particular the dataflow and controlflow diagrams.

I conceived of a "tiny" dataflow operating system that targeted real-time embedded applications, and that took care of all messaging and synchronisation between modules of the dataflow/controlflow diagram. I wrote version 1.0 for the 8051 microcontroller family, based on a new scheduling principle that I coined *Signature Scheduling*. One of the features of this scheduling system is that it has very powerful run-time control over execution, unlike more primitive dataflow models that only have what is called *static* scheduling.

Stack Machines

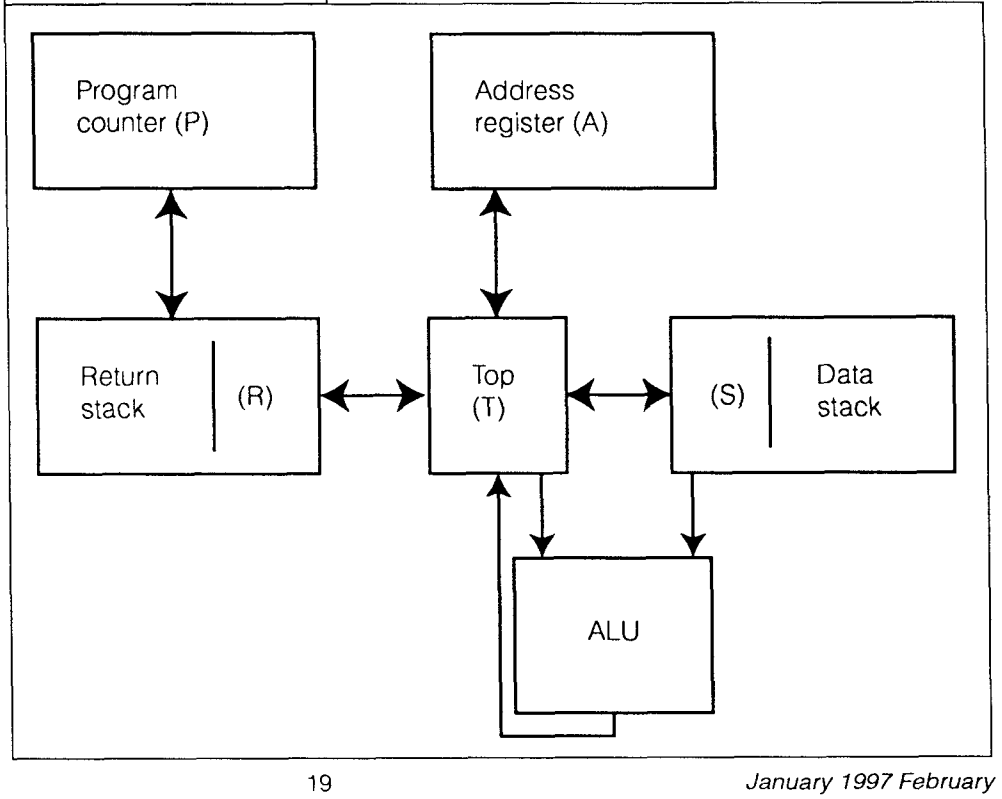
I've been following *Forth machine* architectures over the years, and most recently, of course, Charles Moore has been working on the MuP21, F21, etc. I studied these and was interested enough to purchase the *MuP21 Development System* from Dr. Ting.

It occurred to me that dataflow and a stack machine could be natural partners. In a nutshell, a dataflow system is just a network of code modules that pass messages between each other. Each message consists of a header portion which contains its destination processor, node, and terminal on the node; and a data portion.

The operating system, again in a nutshell, delivers messages and fires a node (code module) when its inputs have all arrived. The system is very elegant, as it automatically takes care of synchronisation between modules, and distribution of modules over a network of processors can be taken care of completely transparently.

Anyway, the currently active messages could sit in a stack and the operating system could examine whatever is on the stack to determine if any node is ready to fire. If so, it would pull the messages off the stack or, better still,

Figure One. MuP21



just bring them to the top of the stack and fire the node.

To be able to implement my OS efficiently on a stack machine, I need to be able to rapidly scan through the active messages, and insert and extract messages. I need random access; but more than that, if I extract a message, I need to be able to "close the gap."

MuP21 greatly intrigued me, as it has a split-stack arrangement, and each half can be independently rotated.

With MuP21, PUSH and POP treat the return and data stacks as one, while it is possible with certain instructions to manipulate each half individually. Also (I don't know if this will stay in future designs), the return and data stacks each rotate their contents—that is, the top element wraps around to the bottom element.

Table Lookup Version of TERSE51

Version 1.0 of my OS, which I call *TERSE* (Tiny Embedded Real-time Software Environment), employs table lookup techniques and needs three different tables, one being a scheduling table; another describing all messages, active or not; and a token table, to flag the status of each message. Basically, TERSE 1.0 keeps a history of execution by calculating a "signature" based on all nodes previously executed, and uses this signature to look in the scheduling table for a list of nodes that can be scanned as potential candidates for next-firing.

Then TERSE will scan through the other two tables, and find which of the eligible nodes has received all input messages.

I managed all of that in about 600 bytes, but it didn't rest easy with me. I liked my overall concepts, but not the implementation. I realised that a complete redesign, based on a machine like MuP21 with a split stack, could make TERSE lean and blindingly fast.

However, I also realised that the stack principle is applicable to conventional architectures, like the 8051—just as Forth will run on an 8051, but it's nothing like having the right stack-based CPU!

Stack-Based TERSE, Conventional CPU

So I completely rewrote TERSE for the 8051, based on keeping the set of currently active messages "floating" in a stack-frame, and it now weighs in at 300 bytes (including networking software). The new version is 1.11, and I have written a description of the background reasoning behind its development, and an outline of how to use it, in another article, "A Tiny Microcontroller Dataflow Kernel," published in *Microprocessors and Microsystems* (April 1996, pp. 105–110, Elsevier Science, U.K.).

I also have an anonymous FTP site for TERSE, at http://scorpion.cowan.edu.au/science/terse/index.html, with the full source code for the 8051 version, and I intend to put a print-file of the magazine article there as well, pending copyright permission.

TERSE51 computes a signature, based on the current set of messages on the stack and on the order of those messages, giving a measure of execution history, as well as the current state. The signature is a number that is unique for each pattern of messages on the stack, and a lookup table is required to relate the signature to which node to fire next. The design is totally deterministic, and no scanning of nodes is required. It also has a safety-net feature, in that any wrong permutation of messages on the stack automatically causes a fall-through to an error handler.

Potential of Split-Stack Approach

Before TERSE51 calls a node, it extracts all messages destined for that node from the stack and puts them into registers R0–R7. On a stack machine with a split stack, the stack could be rotated and the appropriate messages pulled out into the other stack. Then, when a node is called, the messages would be on the stack.

I conceived that MuP21 could keep the currently active set of messages on the return stack, and TERSE could rotate it and extract all messages for the node over to the data stack so, when the node is called, there they are on the data stack, ready to be used.

Note that my signature technique for version 1.11 requires all messages on the return stack to be kept in the same order, as a record of execution history, so whatever algorithm is used to search the messages must not mess up that ordering—which is why I like a stack that rotates.

When a node exits, with TERSE51, all a programmer has to do is push the output messages onto the stack, then exit. I have used a macro to take care of pushing the messages with the correct formatting. On a stack machine, whatever is on the data stack when the node exits becomes the output messages, and TERSE just PUSHes them all across to join the others on the return stack.

Here are some further intriguing thoughts:

Although I have suggested that the active set of messages resides on the return stack and messages to be delivered to a node be extracted onto the data stack, this does not prevent a node from fully utilising both stacks—as long as the node restores them before exiting. This does, of course, mean that the return stack, in particular, will have to be quite deep.

I do, however, envisage a paradigm shift in programming philosophy, in that the return stack will *not* be used to hold subroutine return addresses—individual code modules will not have subroutines/procedures/words, as the nodes are themselves the smallest level of decomposition. The subroutine/procedure/word concept of being called from different places is implemented by clone-nodes (objects) that have different node numbers but are the same code. Thus, apart from interrupt-saving of registers, there won't be much on the return stack, other than the messages.

From Easy Concept to Realisation

There's an incredible amount that I haven't explained, in the limited space of this article. Figure Two illustrates the natural synergy between a dataflow notation and the physical system, and a dataflow OS enables direct realisation of this diagram as code.

It even allows the OS to automatically take care of the classical *mutual exclusion* problem, as represented by two Nodes 1 and 2, in Processor-3, that access Gizmo-A. Node-0 is the error-handlers, and I have shown these daisy chained.

Invitation

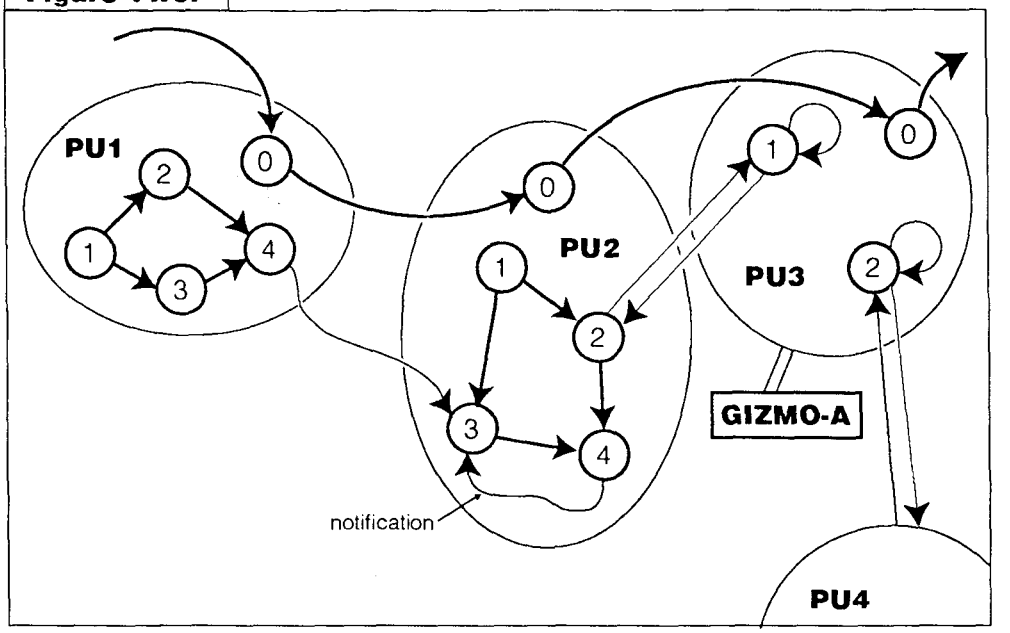
These are my ideas. TERSE51 version 1.11 uses an eight-bit signature, which has a problem with distributed applications, and I will be moving to a bigger, maybe 16-bit, signature. In the immediate future, I'm committed to further developing the 8051 version, but will be moving onto the MuP21 "sometime."

TERSE and its principles, such as signature scheduling, are public domain and I would like it to stay that way.

If you find this interesting, please read my documentation via FTP. I'd like to see someone tackle the equivalent of TERSE51, but on a stack-based machine, such as MuP21. This could be tackled at the basic assembly language level, or the Forth level. At the time of writing, I haven't actually tried implementing anything on the MuP21—maybe a reader can suggest another variation on the above split-stack idea, or some problem with it. I am also keen to see TERSE ported to other conventional CPUs, and further improvements.

A wild thought is that Charles Moore could design the signature generator, and maybe all of TERSE, into "T32"—this would then allow "dataflow utopia," in which dataflow is *fine grained*—that is, even the smallest operation becomes a dataflow node, just like LabView, except it will be compact and fast.

Figure Two.



Barry Kauler's new book, *Flow Design for Embedded Systems* (ISBN# 0-87930-469-3), is being published by R&D Books (an imprint of Miller Freeman, USA). It describes his radical, unified approach to embedded-systems design, and also describes how the TERSE RTOS works. Included with the book will be his "GOOFEE Diagrammer," a graphical CASE tool for Windows; the author believes, "It would be extremely interesting for someone to explore mapping of GOOFEE designs to Forth." He welcomes feedback to b.kauler@cowan.edu.au.

Can POSIX Threads Be Used as a Standard Forth Multi-tasker?

Dr. Everett F. ("Skip") Carter, Jr.
Monterey California

First presented at the FORML Conference of 1996. —Ed.

The IEEE Portable Operating System Interface (POSIX) standard P1003.1c specifies a model for computing with "threads." Threads are multiple processes running within the same memory image. In the Forth vernacular, threads are multiple Forth virtual machines running simultaneously within the same dictionary and data space. The POSIX threads API consists of a set of calls that maps very closely to that of a traditional Forth cooperative multi-tasker. This conceptual similarity could be used to leverage a possible ANS Forth multi-tasker by declaring conformance to the POSIX standard.

What Are POSIX Threads?

The IEEE Portable Operating System Interface (POSIX) standard P1003.1c (also known as ISO/IEC 9945–1:1990c) specifies a model for computing with "threads." The Ada tasking model was the basis of a language-independent model for "lightweight processes" (LWP) in the early 1980s, which ultimately led to threads. This kind of process is used in contexts where the standard Unix model of separate cooperating tasks (generally created through a *fork*) is overkill and, therefore, inefficient. The efficiency is obtained by allowing the different LWPs or threads to be essentially independent program counters or process structures running simultaneously and with the same data space. There are currently several popular threading systems (e.g., for OS/2, Windows-NT, and Solaris, in addition to POSIX) but they are all similar in their essential aspects. Because the POSIX threads are a defined international standard, I will focus upon those here.

How POSIX Threads Relate to Forth Multi-tasking

For all their novelty in the rest of the programming world, a description of what threads are is remarkably familiar to Forth programmers. Threads are so similar to traditional Forth (cooperative) multi-taskers, that the practical, negative impact of Forth absorbing the POSIX threading model appears to be negligible. (See Table One.)

What is Different?

The POSIX standard does not specify whether the task-switching mechanism is cooperative or not, this is up to the implementor to decide. This does imply that any calls to PAUSE would be implicit.

Another difference is that, although there is an equivalent to Forth USER variables, there is a performance hit when accessing them, so one is discouraged from making extensive use of them.

Some Forth multi-taskers can explicitly control the state of *another* task (through WAKE and SLEEP), but PThreads can only control themselves. Actions equivalent to Forth's can be

Table One. Selected POSIX threads functions.

Pthread	function
pthread_create	create a new thread of control (TASK)
pthread_equal	compare two thread handles for equality
pthread_exit	terminate the current thread (STOP)
pthread_join	wait for another thread to terminate
pthread_self	obtain thread ID of the calling thread
pthread_mutex_init	create a semaphore
pthread_mutex_destroy	remove a semaphore
pthread_mutex_lock	block until a semaphore lock is obtained
pthread_mutex_trylock	as above *with blocking*
pthread_mutex_unlock	release a semaphore lock if owned
pthread_cond_init	create a "condition" variable
pthread_cond_destroy	remove a condition variable
pthread_cond_wait	wait on a condition variable
pthread_cond_timedwait	as above but with a timeout
pthread_cond_signal	signal a setting for a condition variable
pthread_cond_broadcast	as above, but unblocks *all* waiting threads
pthread_once	assure that a routine is only called once across all threads
pthread_getspecific	get the address of a thread-specific (USER) variable

achieved through use of condition variables.

Experimenting with Threads in Forth

I have made some exploratory experiments with implementing a PThreaded Forth. I used PFE and Until as testbeds. My conclusions about these tests are:

- It appears that PThreads *is* a practical model to apply to Forth.
- Forth's use of implicit variables via the stack makes the use of threads particularly efficient: this kind of variable access is not available in other languages.
- Threading would not be efficient to supply as an "add-on" to a compiler, but must be built-in by the implementor.

Skip Carter — skip@taygeta.com — is a scientific and software consultant. He is the leader of the Forth Scientific Library project, and maintains the system taygeta on the Internet. He is also the President of the Forth Interest Group.

Code. Simple demonstration of a use of POSIX threads.

```
VARIABLE index

: do_increment ( -- )
    6 0 DO index @ . CR 1 index +! 1000 MS LOOP

    pthread_exit        ( -- )

;

: thread_test ( -- )
    0 index !
     \ start a thread
    ['] do_increment pthread_create      ( xt -- thread flag )

    ." back from thread create " .S CR

    ABORT" unable to create thread"

    1 index +!

    \ wait for that thread to end
    pthread_join                         ( thread -- status )


    ." join status = " . CR

    \ final display value from the variable index is 7
    ." final value of index = " index @ . CR
;
```

ADVERTISERS INDEX

"Experimenting with the ANS Forth Standard"

FORML 1996

Richard Astle
La Jolla, California

For the thirteenth year in a row I spent Thanksgiving weekend at Asilomar, a state park and conference center in Pacific Grove, California, attending the FORML Conference. This was the eighteenth annual conference: it's a shock to realize how few I missed. There are other Forth conferences and events, including the application-oriented Rochester conference, but this is the big one, the "modification laboratory," where changes and experiments are discussed, where Forth is, as far as it can be in public, formed.

That said, attendance could be described as "off." Asilomar is hard to get to by plane and on that weekend in particular (there was talk of whether to move FORML, in place or time). One new attendee combined FORML with another Asilomar conference just after it, avoiding travel on Sunday, optimizing frequent flier miles. But most people drove, from the Bay Area, Southern California, Oklahoma. For me the drive is part of it, leaving San Diego full of turkey, sleeping at a rest stop, then the empty coast road in the morning, coffee at Nepenthe past Big Sur, the view clear to the horizon. I arrived tired, thinking, I'm too old for this, but I thought that last year, too, and the year before, and still come that way.

The lobby at Asilomar was less full of familiar faces than in the past, but to me those who were there were more familiar, and that made up for it somewhat. I always expect some who don't show (where were you, Tom Zimmer?), but Wil Baden was back, with hugs that break eyeglasses, and Klaus Schleisiek, who came the furthest. After lunch at the organizational meeting we found there were almost no papers to present, and it looked like a disaster in progress.

But FORML always surprises, not only with technical innovations, but also the energy of the participants. Short on formal papers (the two in the notebook were by people who weren't there), the conference coalesced into themes, or threads, which surfaced throughout the sessions, from papers, to impromptu talks to discussion groups.

FORML's not famous for fidelity to theme, and this year's announced topic was "Experimenting with the ANS Forth Standard." Nevertheless, several threads touched on the Standard, which is no longer a promising horizon but a working document with enough history to be critiqued from the point-of-view of practice.

The topic of the Standard was broken down, in the working group session, into three threads: "fixing," "using," and "extending." These threads pervaded the conference, as did a fourth, that of "documenting." This last concerned both the traditional sense of explanation, and the more concrete sense of providing a reference implementation, as FIG used to do with fig-Forth, and as Laxen and Perry and Kelly did with Forth-83. An ANS reference implementation would provide a *de facto* answer to questions of interpretation, or at least give those with different interpretations something concrete to complain about.

The ANS Standard document exists: an official version can be purchased for about $200 from ANSI by those serious about defending their own implementations, and a pre-final draft is available from a number of sources on the Internet. But the ANS document, while definitive, is not a tutorial, and in parts is difficult even for experts, as occasionally shown by discussions in comp.lang.forth. For the rest of the world there has only been Jack Woehr's (and where were you, Jax?) now out-of-print *Forth: The New Model*, which was based on a superseded draft of the Standard. This situation is about to change, however: Wil Baden announced that he is revising Leo Brodie's *Starting Forth* to make it ANS compatible.

As far as anyone at the conference knew, the only commercial Forth that claims full ANS compatibility is Power MacForth from FORTH, Inc., though there are several public domain and/or shareware Forths for various platforms that are probably at or near it. The most popular of these, judging from the number of times it was mentioned, is Tom Zimmer and Andy McKewan's public-domain Win32Forth for Windows NT and 95. Win32Forth comes out of the zip file with little documentation other than the pre-final draft of the ANS document and a few examples. In my opinion, most of the trouble people have with Win32Forth is not with Forth, ANS or otherwise, but with the Windows API, and there is plenty of documentation available on this topic: all you need is to know how to translate a C function call to a Forth stack diagram. Still, Win32Forth, like F-PC before it, is full of all kinds of wondrous things that could bear illumination beyond bare

source code and it was felt by many that, as Windows is, for good or ill, the currently dominant computer platform, a documentation project would be a good thing. One attendee, Howard Shapiro, even volunteered to coordinate such a project.

The "using" thread got somewhat overwhelmed by the "documentation" thread, but one theme of it was the collection and publication of a set of "programming pearls," clever and/or efficient implementations of things we all need at some time or another, like the Forth Scientific Library. Few such pearls were offered, though we did get a good Julian date routine. The "pearls" project will be ongoing, and presented sometime in the future to the Forth community as a whole.

Beyond or beneath documentation and use were the questions of the Standard's shortcomings, with respect both to past practice and to future directions (i.e., "fixing" and "extending"). These were separate threads, or at least separate discussions in the working groups session, but they surfaced throughout the conference, and one person's bug fix is often another's new feature, so I'll discuss them together.

POSTPONE was, of course, rehashed, but those it makes uncomfortable seemed resigned: at least there was no loud call for its abolition.

Conference Chairman John Rible pointed out that WORDLIST was not intended to be used by itself but was a compromise, intended as a building block to allow the implementation of various versions of VOCABULARY (83, 79, fig, etc.), much as INVERT and 0= are included to allow the two mutually exclusive implementations of NOT. Despite this explanation, several present said they preferred WORDLIST to any form of VOCABULARY, but András Zsótér noted that the limited number of WORDLISTs provided in the standard makes the concept useless.

The file access wordset was also designed to be a set of primitives for implementing more useful, higher-level, words, but, while WORDLIST is a compromise, Skip Carter asserted that the lack of higher-level file words (such as ATOI) was a deficiency, resulting in a proliferation of disparate solutions to common tasks.

Like the limited number of required WORDLIST slots, the limited required depth of the optional floating-point stack was seen as a shortcoming, though Chuck Moore pointed out that requiring too great a depth would handicap those who implemented Forth in low memory environments.

In general, though not unanimously, participants seemed to favor making the floating-point and control stacks the same as the data stack.

Wil Baden pointed out that obsolescent words, such as #TIB, EXPECT, CONVERT, etc., are intended to be "read only," that is, you shouldn't use them, but you should be able to know what they mean when you see them. Still, several members of the "fixing" working group asserted that the inclusion of these words in the Standard was confusing, and a vote was taken to recommend their removal.

One of the prepared papers from a non-participant (Chris Jakeman, capably channeled by Guy Grotke) bemoaned the inability to manipulate return addresses in a Standard way. >R and R> aren't enough, since there's no guarantee either that an execution address is a single cell or that it resides on the return stack. This is not so much a fix as an extension, since in F-PC, for example, though the execution addresses are on the return stack, they are two cells wide (segment and offset). On the other hand, we used to have the ability to manipulate dictionary structure, but the Standard has no LATEST or LAST or anything with equivalent functionality. This means, for example, you can't find the execution address of a word while you're defining it, so that if RECURSE didn't exist it could not be implemented. I'm particularly interested in this point, since I can't implement RETRY (see *FD*, vol. XVII, number 4) in pure ANS Forth (at least without a kludge incurring a run-time penalty or reliance on a tricky optimizing compiler). I doubt I can get RETRY into the Standard—it's just too quirky for most folks—but I'd be happy just to have LATEST back.

Of the pure extensions (ones that could not be considered bug fixes), the best received was Skip Carter's forthcoming proposal to standardize a Forth multi-tasker with a wordset based on POSIX threads. Other suggestions were for interpretable control structures (as many of us use for conditional compilation), for a standard array wordset (probably unnecessary), and for an object orientation wordset. This last seems important, considering the way much of the rest of the world is going, but András Zsótér pointed out there are just too many ways to do it to expect agreement right now. Perhaps the history of CASE will repeat itself: there were, and still are in practice, many different CASE statement structures, but one is now "Standard." This doesn't prevent other CASE structures from being used, it just means they have to be spelled differently if they are.

Another thread was Forth in hardware, to which one of the Saturday morning sessions and a working group were devoted. Dr. Ting presented a couple of chip designs simulations, Klaus Schliesiek presented a paper on efficient instruction sets, and Chuck Moore discussed his current CPU project, the i21, which is the basis of iTV's forthcoming TV set-top world wide web browser. Chuck said he was "pleased that hardware restrictions are becoming relevant again, and people are going to have to write tight code," a sobering thought to those of us used to the big stack frames that implement CATCH and THROW.

Saturday afternoon the Forth hardware working group, chaired by John Hart, reported that "Forth is the essence of the simplest computer that can be made," and that its essential elements at the hardware level are stacks, nesting, zero operand instructions, list processor (inner interpreter), and a minimal instruction set.

Wine and cheese parties are always a feature of both conference evenings, time for further discussion and showing off. A highlight this year was Skip Carter's six-legged walking robot, which ambled hesitantly about,

guided by sonar, infrared, and mechanical whiskers. When, with an awkward step, the robot broke a leg, Skip replaced it with a spare. Afterwards, towards midnight, as Skip was putting the robot back in his car, two deer, walking more smoothly but with no more urgency, ambled out of a motel parking lot across from us and down the street towards the Asilomar grounds.

Another thread that surfaced throughout the conference, but mostly on the last day, concerned jobs. After years of apparent decline, mirrored in the shrinking FIG membership, people are talking about career opportunities, and not just as some hope based on the presumed future of Forth in hardware.

One direction is towards Open Firmware, a Forth-based, hardware-independent boot code language pioneered at Sun and adopted by IBM, Motorola, and Apple. Conference attendees Randy Leberknight and John Hall spoke about working conditions and job possibilities at Motorola and Apple, respectively.

Forth-in-hardware lives at iTV, which had a significant presence at FORML. That company's web browser has already been mentioned. The president of the company, Gary Langford, came to the last conference session to meet people and to announce that he hoped to double iTV's programming staff in the near future.

The last session of the conference traditionally includes awards (leftover bottles of wine), brief impressions of the conference by first-time and long-time attendees, and a panel discussion. The panel this year consisted of John Hall, Lloyd Prentice, and Trace Carter, who spoke from different perspectives on the non-programming skills useful to Forth programmers. Trace, who took over the daily management of FIG from the Hall brothers in January, suggested that we need to know what we're not good at (e.g., organizational skills) and to be able to hire people to do those things for us. Lloyd, who runs a small Forth-based company on the East Coast, suggested we had to hone our communication skills, and to remember we're selling solutions, not Forth. And John Hall, from his perspective as a former employee at Sun and a current employee at Apple, suggested we had to learn to keep our individuality in the face of corporate culture. The main theme of discussion, however, became that of teamwork and the Forth programmer: how do we, who most often work alone, and quite productively, learn to work together, as current career opportunities often require?

After Sunday lunch (the traditional turkey pot pie) many of us were able to extend our stay in the area by a few hours at Skip and Trace Carter's house in Seaside, just north of Monterey, where we had food and drink treats, a visit with another robot and a couple of cats, and the sight of John Rible and Chuck Moore, among others, working on a 3-D jigsaw puzzle with a view of Monterey Bay beyond them.

Richard Astle has been programming in Forth for about eight years, most of that time developing and maintaining a rather large database-management set of programs. In the process, he has re-written the underlying Forth system more than once for speed and capacity. He has a bachelors degree in mathematics from Stanford University, a master's in creative writing from San Francisco State, and a Ph.D. in English literature from the University of California (San Diego).

(Back to Forth, from page 13.)

stream), and it would be better to use the same interface convention in both cases. (I mean, a standard API, like ODBC or IDAPI, is necessary. In fact, this is a step towards stream-oriented, or data-flow, programming.) From this point of view, the transaction mechanism should be supported not only for database modifications, but for some other, application-dependent actions, too.

In short, not an SQL-centric server, but a procedural language-centric server is necessary. I think Forth is a suitable basis for implementation of that approach.

Besides, in this case, the same language—Forth—can be used for both the client and server parts of an application.

References.

1. Leo Brodie. *Starting Forth.* Prentice-Hall, 1981.
2. Dick Pountain. *Object-Oriented Forth.* Academic Press Limited, London, 1987.
3. *Microsoft Visual Basic v 4.0. Programmer's Guide.*

Anatole Medyntsev — MedyntsevA@Novavox.Ru — learned about Forth in 1985. (Several Forth systems have been implemented in the former USSR, where Forth remains very popular.) He was a software engineer in the Department of Computer Science at the St.Petersburg Electrotechnical University. While there, he used Forth, among other things, to implement an efficient, object-oriented Prolog. The Forth system described in this paper began in 1994, as part of a solution to business-application development in the banking industry.

Squareroot and Golden Ratio

Wil Baden
Costa Mesa, California

The Stretching Forth article that comes after this one wants the golden ratio to 32 binary places. The golden ratio is $(\sqrt{5}-1)/2$ or $\sqrt{1.25}-0.5$. This will require taking the squareroot of a 65-bit value.

The rest of this article will develop practical definitions for squareroots of single and double numbers, as well as a special definition to calculate the golden ratio.

My approach is to imitate the pencil-and-paper method of extracting the squareroot. Using binary arithmetic rather than decimal arithmetic simplifies the process.

Here is a worksheet for pencil-and-paper extraction of $\sqrt{1.25}$ using decimal arithmetic. If you don't understand what's being done, skip ahead. Come back to it after you've read the rest of the article. I did the example to help me understand what I was trying to do in Forth.

Squareroot of 1.25, using paper-and-pencil method:

```
    1. 1  1  8  0  3 ...

_   1.25 00 00 00 00 ...
1   1
    -
2_  0.25
21   .21
    ----
22_     4 00
221     2 21
        ----
222_        1 79 00
2228        1 78 24
            -------
2236_            76 00
22360             0
                 -----
22360_           76 00 00
223603           67 08 09
                 --------
```

Figure One shows the Forth.

Taking `BITS/CELL` as the number of bits per cell, this yields the 16-bit squareroot of an unsigned 32-bit integer, or the eight-bit squareroot of an unsigned 16-bit integer. Our interest is in 32-bit integers.

There is also a dependency on 2's complement arithmetic.

While preparing this article I fortuitously came across the same algorithm written in C (Figure Two).

At first glance these seem to be very different, but a closer look shows that the Forth could have been derived in three steps from the C.

Step One. Direct transcription.

As the C code is transcribed, we observe that we can dispense with variable `i`. (Figure Three.)

Figure One.

```
32 CONSTANT BITS/CELL

: SQRT                ( radicand -- root )
    0 0                             ( remainder . root)
    BITS/CELL 2/ 0 DO               ( remainder . root)
        >R D2* D2* R>    \ Shift remainder left 2 bits.
        2*               \ Shift root left 1 bit.
        2DUP 2* U> IF    \ Check for next bit of root.
            >R R@ 2* - 1- R>     \ Reduce remainder.
            1+                   \ Add a bit to root.
        THEN
    LOOP
    NIP NIP              ( root)
;
```

Figure Two. The C squareroot code.

```
/* Author: Ken Turkowski, Apple Computer */
unsigned long SquareRoot (unsigned long v)
{
    unsigned long root = 0; // square root
    unsigned long remHi = 0; // high part of partial remainder
    unsigned long remLo = v; // low part of partial remainder
    unsigned long testDiv;
    int i;

    for (i=16; i>0 --i) // Loop through the 16 bits of the value
    {
        remHi = (remHi << 2) | (remLo >> 30); // Shift the 64-bit
                                              // partial remainder left
        remLo <<= 2;
        root <<= 1;                           // Shift the calculated
                                              // root to make room
        testDiv = (root << 1);
        if (remHi > testDiv)
        {
            remHi = remHi - testDiv - 1; // Take error out of the
                                         // partial remainder
            root++;                      // Add a one bit to the root
        )
    }

    return root;
}
```

Figure Three.

```
VARIABLE root
VARIABLE rem-hi
VARIABLE rem-lo
VARIABLE test-div

: square-root                ( radicand -- root )
    rem-lo !                     ( )
    0 rem-hi !
    0 root !
    16 0 DO
        rem-hi @ 2 LSHIFT rem-lo @ 30 RSHIFT OR rem-hi !
        rem-lo @ 2 LSHIFT rem-lo !
        root @ 1 LSHIFT root !
        root @ 1 LSHIFT test-div !
        test-div @ rem-hi @ u< IF
            test-div @ 1+ NEGATE rem-hi +!
            1 root +!
        THEN
    LOOP
    root @
;
```

Figure Four.

```
VARIABLE rem-hi
VARIABLE rem-lo

: square-root                    ( radicand -- root )
    rem-lo !                         ( )
    0 rem-hi !
    0                                ( root)
    16 0 DO
        rem-hi @ 2 LSHIFT rem-lo @ 30 RSHIFT OR rem-hi !
        rem-lo @ 2 LSHIFT rem-lo !
        1 LSHIFT
        DUP 1 LSHIFT rem-hi @ U< IF
            DUP 1 LSHIFT 1+ NEGATE rem-hi +!
            1+
        THEN
    LOOP
;
```

Figure Five.

```
: Q2*                          ( n . . . -- 2n . . . )
    ( Dependency on 2's complement arithmetic. )
    D2* >R >R
    DUP 0< IF
        D2* R> 1+ R>
    ELSE
        D2* R> R>
    THEN
;

: DSQRT                        ( radicand . -- root )
    0. 0.                          ( radicand . . . root .)
    BITS/CELL 0 DO                 ( radicand . . . root .)
        2>R Q2* Q2* 2R>
        D2*
        2OVER 2OVER D2* 2SWAP D< IF
            2DUP 2>R D2* D- -1 M+ 2R>
            1 M+
        THEN
    LOOP
    DROP NIP NIP NIP NIP           ( root)
;
```

Figure Six.

```
MARKER NONCE
: golden-ratio                 ( -- golden-ratio )
    0 [0x] 40000000 0. 1.
    BITS/CELL 0 DO                 ( radicand . . . root .)
        2>R Q2* Q2* 2R>
        D2*
        2OVER 2OVER D2* 2SWAP D< IF
            2DUP 2>R D2* D- -1 M+ 2R>
            1 M+
        THEN
    LOOP
    2SWAP 2DROP 2SWAP 2DROP        ( root .)
    [0x] 80000000 0 D- DROP        ( golden-ratio)
;

golden-ratio      \ 9E3779B9
NONCE
    CONSTANT golden-ratio
```

Step Two.
Reduce the number of variables.

Next we keep `root` on the stack and do the trivial computation of `test-div` as we need it. (Figure Four.)

Step Three. Eliminate all variables.

Finally we keep `rem-hi` and `rem-lo` as a double number on the stack. We recognize that the shift operations involving `rem-hi` and `rem-lo` are just to shift the double number remainder left two bits.

This gives us `SQRT` above.

We can now do squareroots of single integers. For squareroots of double integers we "promote" the constants and operations in `SQRT`.

Only one new utility operation has to be defined: `2*` for quadruple numbers. (Figure Five.)

We can now take squareroots of double numbers. With 32-bit cells we can take squareroots of 64-bit double numbers. We can see the pattern to take squareroots of longer numbers.

Instead of promoting operations again we will make a definition to be used once and thrown away. This is used to give us the value of the golden ratio, $\sqrt{1.25} - 0.5$ to 32 binary places as a 32-bit integer. (Figure Six.)

Appendix

`[0x] <hex-number>` is how I get hex literals in a definition.

```
: [0x]
  S" [ HEX ] "
  PARSE-WORD S+
  S"  [ DECIMAL ] "
  S+
    EVALUATE
; IMMEDIATE
```

Wil Baden is a professional programmer with an interest in Forth. wilbaden@netcom.com

Using Forth to manipulate the real world

Forthware

Closing the Loop — PID Controllers

Skip Carter

Monterey, California

Introduction

Now that we know how to make measurements from the outside world and how to generate various control signals, let's look at how to combine them in order to regulate a process. An example of the kind of thing we want to do is controlling the speed of a DC motor. Based upon a previous column (*FD* XVIII/2), we know we can set the motor speed with a PWM control signal. If we had perfectly calibrated the system so that, say, a 75% duty cycle produced a motor speed of 1000 RPM, we would have an *open-loop* controller. The downfall of an open-loop controller is dealing with all the real world problems like inaccurate calibration, variations in supplied power, or variations in load on the motor. These complicating factors can only be handled if we actually monitor the motor's speed and somehow use the difference between the commanded and actual speed to correct the control signal. This is a *closed-loop* controller.

How we actually implement a closed-loop controller depends upon what we are trying to control (e.g., motor speed or motor position), and its relationship to what we are measuring back from the controlled system (e.g., motor RPM or motor angular position). As we will see, the implementation of the controller is also driven by the behavior we can tolerate when the system is trying to correct itself when it is suddenly perturbed.

The Proportional Controller

The easiest closed-loop controller scheme is one where the error signal is fed directly back into the controller and the new control output is determined by some fraction of the error.

$$u_{new} = u_{old} + K_p \varepsilon$$

For closed-loop controllers, it is easiest to think in terms of the *error*, the difference between the commanded and measured values, of some quantity (like RPM) instead of the measured value directly. So we write the above as,

$$z = K_p \varepsilon$$

where z represents the correction value to apply to the controller.

This *proportional* controller is very easy to implement, and is frequently used with good results. The controller does have a flaw in it: if the system is sitting at the desired output level (the *set point*) and it gets perturbed, the controller has a tendency to respond by oscillating about the set point. These oscillations cause the system to be at the set point *on average*, but it never settles down to stay there.

Improving the Controller

We can reduce the tendency of the control signal to oscillate about the desired position by giving the controller some memory. We accomplish this by adding a term that keeps track of the accumulated error over time,

$$z(t) = K_p \varepsilon(t) + K_i \int_0^t \varepsilon d\tau$$

The integral in this equation provides the memory. This type of controller is known as the *proportional-integral* or *P-I* controller.

Since the integral accumulates the error from the beginning, there is the possibility of an overflow in this term, especially if an integer-only implementation is used and/or the sample rate is high. For integer implementations, two common tricks are used to reduce the overflow problem: (i) whenever an overflow is detected, reduce the accumulated sum by half and increase the gain by two; and (ii) put limits on the integral term and just stop accumulating when these bounds are reached.

Another improvement we can add to the basic proportional controller is to make it more responsive to variations in the control signal. We can do this by adding a term that is large when the control *changes*, i.e., we add a derivative term,

$$z(t) = K_p \varepsilon(t) + K_d \frac{d\varepsilon(t)}{dt}$$

If we just add the derivative as above, we get the *proportional-derivative*, or *P-D* controller.

The PID Controller

We can use all three terms together and get the *proportional-integral-derivative* controller. The equation that fully describes the PID control is:

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

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

For a practical implementation on a digital system, the above equation is replaced with a discrete time approximation:

$$z(k) = K_p\varepsilon(k) + K_i\sum_{j=0}^{k}\varepsilon(j) + K_d[\varepsilon(k) - \varepsilon(k-1)]$$

where we have used a rectangle approximation to the integral and a backward difference approximation for the derivative. These are rather crude numerical approximations, but we will generally be able to compensate for this by making the discrete sampling rate fast and by tuning the controller. The simple numerics has the great virtue of being very easy to implement.

Fine-tuning a PID Controller

It is very rare to see a discussion of PID controllers without also learning how important it is to properly tune the controller. Much of the literature talks about empirical tuning of the controller. The PID equation is based upon an integro-differential equation, and the "gain" factors are determined by a combination of the desired gain plus the effect due to numerically approximating the different parts of the equation.

Part of the preoccupation with empirical tuning PID controllers is cultural and part of it is practical. I have noticed that it is a rare engineer (as opposed to scientists) that actually spends the time to analytically solve differential equations, and not many scientists or engineers have the skills in their bag of tricks to solve integro-differential equations. Fortunately the equations are linear, which means they are solvable with such techniques as Laplace transforms. A second reason why PID controllers need to be empirically tuned is that often, for real-world systems, the exact equations aren't really known, especially given the uncertainties inherent in a system containing mechanical devices and noise-prone sensors.

It is also possible to design self-tuning controllers by adding an adaptive tuning section to the control software. We will leave the topic of adaptive systems for another time.

Experimenting With the Sample Code

The code example in Listing One implements an integer PID controller. The code uses scaled integer arithmetic, scaled such that 128 is effectively a scaled 1. This is a useful scaling for eight-bit sampling (by putting 1 in the mid-range). I am simulating the sensor input that would be coming from, say, a frequency counter measuring motor RPM. This simulation consists of a sensor that returns a noise-contaminated version of the actual setting (to turn off the random noise, set `noise_range` to zero). One sets the PID gains with a sequence such as:

```
64 32 10 gains!
```

which will set, respectively, the proportional gain to 64, the integral gain to 32, and the derivative gain to 10 (effectively, 0.5, 0.25 and 0.08). After the gains are set, a run of the controller is done by typing,

```
88 pid-run output.1
```

This will run the controller, with the current gain settings for a desired output set point of 88, and send the results to a file called `output.1`. The code is set up so the controller can be continued, with output to another file for comparison, e.g.,

```
160 pid-run output.2
```

The output is in a form convenient for plotting with the program `gnuplot` by using a command like,

```
gnuplot> plot 'output.1'
with linespoints,'output.2'
with linespoints
```

[Ed. note: the command should be typed on a single line, it is shown broken for typographical purposes.]

If you run the program with different gain settings, you will see several things going on. First, with just the proportional gain non-zero, the steady state is an oscillation around the ratio between the set point and the (fractional) gain (`set_point @ kp @ 128 */`). Also, the oscillations damp out at a rate that is inversely proportional to the gain. Consequently, the ability to quickly damp out the oscillations competes against the ability to reach the desired output setting. The DC level of the output can be tweaked by adjusting the initial output value, but simply using the integral term takes care of that automatically. Setting the P and I terms to non-zero makes the output actually settle out at the desired setting. If you now add the derivative term (by setting its gain to non-zero), the controller will respond more quickly to changes in the set point. The problem with the derivative term is that it is very sensitive to noise in the data; it looks pretty helpful if the noise is zero, but it amplifies the noise if it's not zero.

Play around with different settings and see how the controller responds. For extra credit, replace my sensor/controller simulator with the real thing, try replacing the sensor with the frequency counter from my last column (measuring the RPM of a motor by counting revolutions) and the controller with the PWM motor controller from *FD* XVIII/2, giving a speed-controlled motor that will automatically adjust for varying loads.

Conclusion

We have left out one possible combination, the I-D controller. I have never seen this subset used for any practical application (perhaps one of our readers has?), so I will not consider it in this introduction to controllers.

Another topic, which I have only mentioned the existence of, is how to make *adaptive* PID controllers. This type of PID controller can actually use the performance of the system to tune the gains while the controller is running. These controllers are implemented by combining aspects of PID controllers with adaptive digital filters. This is definitely a topic for the future, since it requires an understanding of digital filtering, something we have so

far covered in these columns only in a cursory way.

My last column on measurement of frequency containing data got a lot of positive responses, including such comments like, "give us *more* math!" (I actually spared you a bit on this request, a full analysis of PID controllers can be very mathematical). Thank you all for your input. If you have comments, suggestions and criticisms, please don't hesitate contacting me through *Forth Dimensions* or via e-mail at skip@taygeta.com. I receive about 100 e-mail messages per day, so I can't individually answer each, but I *do* read them all.

Downloading: pid.fth is available at ftp://ftp.forth.org/pub/Forth/FD/1997

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

Listing One.

```
\ pid.fth   A PID controller in Forth

\ This is an ANS Forth program requiring:
\         1. The File wordset
\         2. The Forth Scientific Library ASCII file I/O words
\         3. The Forth Scientific Library random number
\            generator R250 (algorithm #23)

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

\ $Author:   skip  $
\ $Workfile:   pid.fth  $
\ $Revision:   1.1  $
\ $Date:   11 Dec 1996 08:29:34  $
\ ================Support code for the simulation==================



: d* ;                  \ compilation stub for R250, not used anywhere
: umd* ;                \ ditto
: umd/mod ;             \ ditto


S" /usr/local/lib/forth/fsl-util.fth" INCLUDED
S" /usr/local/lib/forth/fsl/r250.seq" INCLUDED
S" /usr/local/lib/forth/fileio.fth"   INCLUDED

-1 VALUE fout               \ output file handle

9 CONSTANT tab_char         \ the TAB character

CREATE crlf   2 ALLOT
CREATE tab    1 ALLOT

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

: print_tab ( fout -- )
    tab
```

```
    1
    ROT write-token
;


8 VALUE noise_range         \ the total range of the noise
                            \ set to zero for no noise

100 VALUE max_samples       \ number of samples per run

VARIABLE #samples           \ the number of samples obtained
VARIABLE setting            \ the current sensor input value
VARIABLE tcount             \ the "time" count

TRUE VALUE sensor_once?

: sensor_init ( -- )
    0 #samples !

    sensor_once? IF     \ stuff to do only once
               1. r250d_init
                -1  tcount !
                 0  setting !
          THEN

    FALSE TO sensor_once?
;


: noise+ ( -- x ) \ get noise sample +- noise_range/2

        noise_range 1 > IF
                         r250d D>S noise_range MOD
                   noise_range 2/ -
                 ELSE
                   0
                 THEN

;

\ the simulated (noisy) sensor
: sensor@ ( -- x flag )         \ flag is FALSE if no more data

    1 #samples +!
    #samples @ max_samples > IF
                              0 FALSE
                            ELSE
                              noise+ setting +!
                          setting @ TRUE
                            1 tcount +!    \ increment the "time"
                              THEN
;

\ the simulated external control system output
: controller! ( x -- )
```

```
    setting !

    tcount @ fout write-int
             fout print_tab

    setting @ fout write-int
              fout print_endline
;



\ ==================The PID demo code================================


 32000 CONSTANT ulimit
-32000 CONSTANT llimit

   128 CONSTANT scale_factor

VARIABLE kp             \ proportional gain
VARIABLE ki             \ integral gain
VARIABLE kd             \ differential gain

VARIABLE isum           \ integral sum
VARIABLE old_err        \ previous error value

VARIABLE set_point      \ the INPUT control setting

TRUE VALUE pid_once?

: initialize ( -- )

    10 crlf   C!  13  crlf 1+ C!
    tab_char tab C!

    pid_once? IF
                0 isum !
                0 old_err !
          THEN

    FALSE TO pid_once?

    sensor_init
;

: gains! ( kp ki kd -- )
    kd !
    ki !
    kp !
;

: integrate ( err -- sum )

    isum @ +          \ accumulate the sum

    ulimit MIN        \ apply limits
```

```
    llimit MAX

    DUP isum !
;

: differentiate ( err -- diff )

    DUP old_err @ -

    SWAP old_err !
;


\ an integer PID controller, using a scaling factor
: pid ( xin -- xout )

    \ calculate the current error
    set_point @ SWAP -              ( error )

    DUP kp @ scale_factor */                    ( error p )

    OVER integrate ki @ scale_factor */ +       ( error pi )

    SWAP differentiate kd @ scale_factor */ +   ( pid )

;



\ Note: set the gains, e.g.:  64 32 10 gains!
\       BEFORE running pid-run

: pid-run ( set --<outfile>-- )      \ give the setpoint as input

    BL WORD COUNT       \ get the output file name

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

    CR

    set_point !

    initialize

    BEGIN
      sensor@     \ get the sensor data
    WHILE
      pid             \ apply PID
        controller!     \ set controller
    REPEAT

    fout CLOSE-FILE DROP

;
```

17th Annual Rochester Forth Conference

hosted by the Institute for Applied Forth Research, Inc.

Held at:	*For more information:*	*Or write:*
University of Rochester	716-235-0168 (*voice/fax*)	Rochester Forth Conference
June 25–28, 1997	lforsley@jwk.com	Box 1261
		Annandale, Virginia 22003 USA

Take Advantage...

You are *already entitled* to benefits you might not know about!
The Forth Interest Group's aggressive plan of membership benefits is more than a subscription to the world's most widely read Forth publication.

Membership includes *Forth Dimensions*—now here is what else:

- 10% discount on books and software in the extensive mail-order catalog of the Forth Interest Group.
- 10% discount on advance registration for the FORML conference, FIG's annual forum for the study and improvement of the Forth language.
- Full access to the foremost Forth site on the World Wide Web at FIG's www.forth.org, *including:*
  • free Web page • free e-mail forwarding • your résumé on FIG's "programmers" Web page • access to special interest groups on the "members-only" Web page • FTP access to FIG's Forth Software Library
- Discount on FIG's Internet domain registration service.
- Referral to Forth-related job openings.
- Direct networking with other Forth programmers through FIG's many chapters worldwide.

Help yourself to the benefits that best serve you. And remind colleagues and employers to join the Forth Interest Group, too—see the centerfold order form for details, or contact the FIG office:

408-37-FORTH (408-373-6784)
office@forth.org

Forth companies and staff can benefit, too.

Keep your Forth team up-to-date and connected to the discoveries, explorations, and experience of some of the best minds you'll find in *any* computing discipline.

Corporate membership ($125 per year) includes all the individual benefits, plus:

- Four extra copies of *Forth Dimensions*—less need to share copies among staff members.
- A fifty-word "Corporate Member" listing in *Forth Dimensions* to describe your products or services.
- 10% discount on advertising rates in *Forth Dimensions.*
- A link to your Web site from FIG's home page.

Support your library—or build one.

A Forth reference section is a great asset to any corporate, public, or academic library. A library membership in the Forth Interest Group makes it simple to build your collection of models, techniques, useful code, tutorials, and aids to program (and programming) efficiency.

Library membership ($125 per year) includes all the individual benefits, and:

- 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.

New release!

FORML Conference Proceedings

1995: "Forth as a Tool for Scientific Applications" 1994: "Interface Building"
Two years' conference contents in one convenient volume.
$50 — or order four FORML volumes and receive the lowest-priced one *free...*
Call FIG now for this limited-time offer: 408-37-FORTH *(408-373-6784)*
(Special offer expires March 31, 1997)