VOLUME X, NUMBER 5 JANUARY/FEBRUARY 1989

FORTH DIMENSIONS

SELF-CHECKING NUMBERS

APPLE II $FORTH

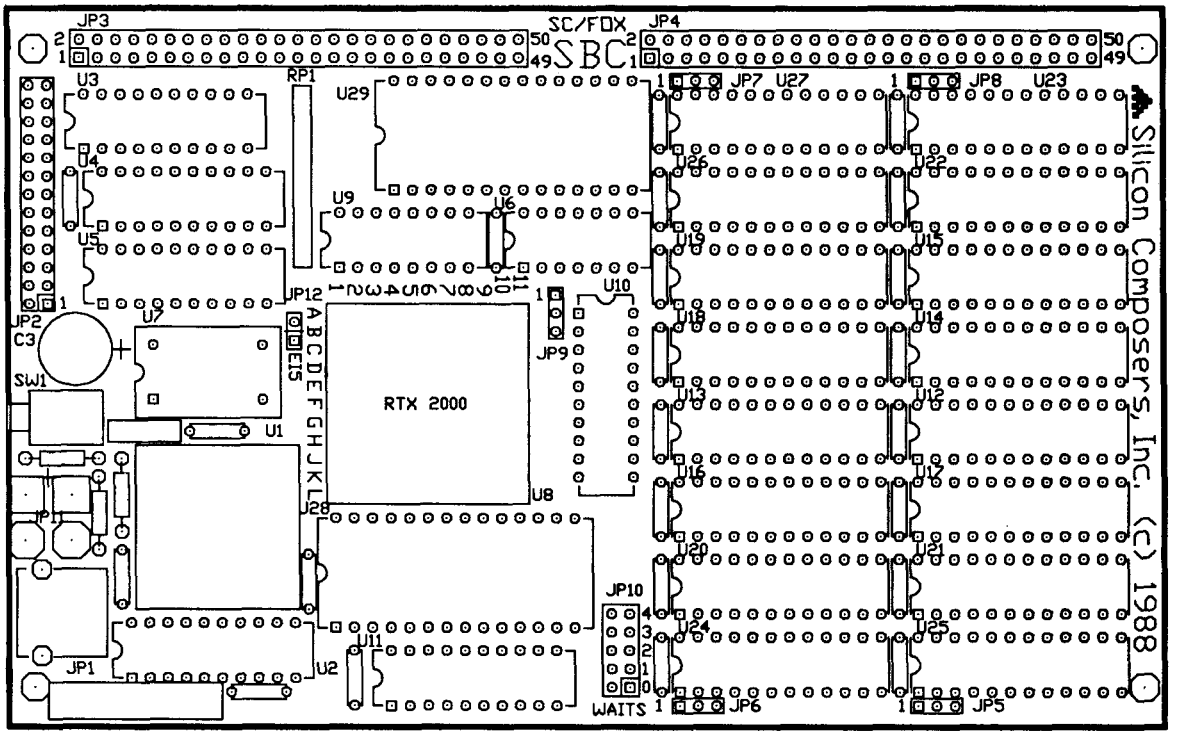
REAL-TIME PROGRAMMING

THE VALUE OF FIG CHAPTERS

SILICON COMPOSERS

Introduces the

SC/FOX™ Single Board Computer



(actual size)

SC/FOX (Forth Optimized eXpress™) SBC:

- 8 and 10 MHz RTX 2000 options
- 32K to 512K bytes 0-wait state SRAM
- 64K bytes of shadow-EPROM space
- Application boot loader in EPROM
- FCompiler™ Forth software included
- Code converter for EPROM programs
- RS232 serial port with handshaking
- Centronic parallel-printer port
- Single +5 volt board operation
- Two 50-pin application headers
- Eurocard size: 100mm by 160mm
- SC/FOX Coprocessor compatibility
- Retail from $1,195 with software

Harris RTX 2000™ Forth CPU Features:

- 1-cycle 16 x 16 = 32-bit multiply
- 1-cycle 14-prioritized interrupts
- one non-maskable interrupt
- two 256-word stack memories
- three 16-bit timer/counters
- 8-channel 16-bit I/O bus
- CMOS in 85-pin pin-grid array

Optional SC/FOX SBC Products:

- SC/Forth™ Language in EPROM
- SC/Float™ Floating Point Library
- SC/SBC/PROTO™ Prototype Board

Ideal for embedded real-time control, high-speed data acquisition and reduction, image or signal processing, or computation intense applications. For additional information, please contact us at:

Silicon Composers, Inc., 210 California Ave., Suite K, Palo Alto, CA 94306 (415) 322-8763

FORTH DIMENSIONS

EDITORIAL

Forth Dimensions
Published by the
Forth Interest Group
Volume X, Number 5
January/February 1989
Editor
Marlin Ouverson
Advertising Manager
Kent Safford
Design and Production
Berglund Graphics

The Forth Interest Group recently appointed a new coordinator for its international network of chapters. Jack Woehr, who has already established a presence on far-flung electronic networks, now makes his first official contribution to *Forth Dimensions*. He clearly understands the benefits of strong chapters, and we welcome him to these pages—please read and respond. Let Jack know about your own thoughts and experience. The straightest path to a vital organization is direct communication...

* * *

One of the articles in this issue forced me to remember an important fact. Previous-generation machines are still used for important work around the world, wherever results and functions are more important than the currently preferred makes and models of the commercial marketplace. In a way, mastery of Forth makes it possible for one to support his own orphaned computing machines of any generation. It's applied hackery at its best, like Luke Skywalker recycling smart parts and refining the elegant light saber.

Forth strings for the Apple II is, admittedly, rather specific for this magazine, but we are reminded of all those laboratories with old Apples still dutifully clicking away after all these years. Maybe I was influenced by my own mechanically sound but task-starved hardware: TRS-80 Model III and Model 100, parallel Diablo, a 128K Mac, and miscellaneous cables and modems (sorry, the Apple II sold a few years ago). Mostly, we present this string stack as a complement to material from the preceding issues; readers will find it useful as a source of secondary illumination about strings, stacks, and Forth. If it also saves a secondhand computer from incidental oblivion, so much the better.

* * *

The public-domain F83 implementation of Forth has an interesting quirk (or feature). It enigmatically prints, "The rest is silence." For some years, I had assumed this to be a Zen-like iceberg in the living room or just a favorite mantram. But while reading *The Hitchhiker's Guide to the Galaxy*, I came upon the episode wherein two deadly nuclear missiles, by virtue of having been caught up in an Improbability Field before detonating, are turned into a bowl of petunias and a sperm whale, respectively. Snagged by the gravity field of an alien planet, the whale has enough falling time for an introspective monologue in which it becomes aware of itself and its surroundings before the author informs us, "And the rest, after a sudden wet thud, was silence."

I thought to ask Mike Perry if that was, after all, the origin of F83's philosophy-in-a-nutshell, a kind of general disclaimer from No Visible Support Software. He denied it, and I believe him. It's probably just a variation on what another friend of mine would call the sound of one statement looping.

—Marlin Ouverson
Editor

About 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. Many research hardware and software designs that will advance the general state of the art. 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 $24/36 per year by the Forth Interest Group, 1330 S. Bascom Ave., Suite D, San Jose, CA 95128. Second-class postage paid at San Jose, CA. POSTMASTER: Send address changes to *Forth Dimensions*, P.O. Box 8231, San Jose, CA 95155."

LETTERS

Local Variables Revisited
Dear Editor,

In a previous letter (*FD* IX/5), I suggested a simple implementation of local variables in high-level Forth. I also suggested assembly coding for faster execution. I was happy to see the letter by Michael Barr (*FD* X/1) in which he presents assembly code that works for quantities as well as for variables.

I would like to point out that the speed can be improved considerably, even without assembly language. With the code I provide with this letter (see Figure One), the declaration and restoration of a local variable executes in 0.71 ms. on my PC/Forth system, compared to 2.0 ms. for my earlier code. Mr. Barr's assembly code executes in 0.29 ms. This has been tested by adding an unused local variable to the definition of `N!`. `LOCAL` must be used with variables, but I have added `LOCAL>>` to work with quantities as well, following Michael Barr's approach.

Because words are compiled directly into the body of `LOCAL`, one level of word nesting is avoided. And as an extra bonus, this makes access to the return stack simpler, and a pair of `>R` and `R>` can be omitted.

Forth can very easily be extended with words like `2LOCAL`, `CLOCAL`, `FLOCAL`, etc. to provide local variables for other simple Forth data types. The accompanying code is for PC/Forth; for PC/Forth+, include `ADDR>S&O` where shown.

Sincerely,
Henning Hansen
Technical University of Denmark
Building 116
2800 Lyngby, Denmark

```
: RESTORE  R> R> ;

: LOCAL  ( addr -- )  \ local variable
  COMPILE DUP COMPILE @ COMPILE SWAP
  COMPILE >R COMPILE >R
  ['] RESTORE >BODY ( ADDR>S&O ) [COMPILE] LITERAL
  COMPILE >R ;  IMMEDIATE

: LOCAL>>                    \ local quantity
  ' >BODY [COMPILE] LITERAL
  COMPILE DUP COMPILE @ COMPILE SWAP
  COMPILE >R COMPILE >R
  ['] RESTORE >BODY ( ADDR>S&O ) [COMPILE] LITERAL
  COMPILE >R ;  IMMEDIATE
```

Figure One.

Label Types for Faster Debugging
Dear Editor,

In Forth, a few hundred lines of source code gives no problem in remembering that a `VARIABLE` must be handled as a variable, a `CONSTANT` as a constant, a `LOCAL` as a local, a `GLOBAL` as a global, an `FVARIABLE` as a floating-point variable, an `FCONSTANT` as...etc.

With ten or twenty thousand lines of source code, remembering whether `<name>` is a constant, variable, array, floating point, etc. is hard. It is worse with the higher versions of Forth, which come with structure, one- and two-dimensional arrays, constants, variables, floating point, locals, globals, deferred actions, tokens, etc.

A few bytes in source code allows labelling the names by type. This is done a little, now. By convention, `+<name>` means something that adds, often an offset. But it sometimes means adding a menu or something else! I'm reducing confusion by using the conventions shown in Figure Two.

Forms can be combined if it will increase the information content. For example, `F~A1<name>` is surely a one-dimensional array of floating-point numbers. Of course, versions with small name fields could use `<name>ETC .` since this is only to improve readability.

Anything to improve readability of source code should be looked at. This habit has greatly reduced time spent digging out the definition of various names in order to decide how it should be handled!

Douglas Hvistendahl
P.O. Box 250
Ellendale, ND 58436

String Stack & F-PC
Dear Marlin:

I am just dropping a note to let people know the current status of the string package described in "Using a String Stack" (*Forth Dimensions* X/3 and X/4).

In the year since I gave that article to you, the string package has greatly evolved.

(Continued on page 7.)

YES, THERE IS A BETTER WAY
A FORTH THAT ACTUALLY
DELIVERS ON THE PROMISE

HS/FORTH

POWER

HS/FORTH's compilation and execution speeds are unsurpassed. Compiling at 20,000 lines per minute, it compiles faster than many systems link. For real jobs execution speed is unsurpassed as well. Even non-optimized programs run as fast as ones produced by most C compilers. Forth systems designed to fool benchmarks are slightly faster on nearly empty do loops, but bog down when the colon nesting level approaches anything useful, and have much greater memory overhead for each definition. Our optimizer gives assembler language performance even for deeply nested definitions containing complex data and control structures.

HS/FORTH provides the best architecture, so good that another major vendor "cloned" (rather poorly) many of its features. Our Forth uses **all** available memory for both programs and data with almost no execution time penalty, and very little memory overhead. None at all for programs smaller than 200kB. And you can resize segments anytime, without a system regen. With the GigaForth option, your programs transparently enter native mode and expand into 16 Meg extended memory or a gigabyte of virtual, and run almost as fast as in real mode.

Benefits beyond speed and program size include word redefinition at any time and vocabulary structures that can be changed at will, for instance from simple to hashed, or from 79 Standard to Forth 83. You can behead word names and reclaim space at any time. This includes automatic removal of a colon definition's local variables.

Colon definitions can execute inside machine code primitives, great for interrupt & exception handlers. Multi-cfa words are easily implemented. And code words become incredibly powerful, with multiple entry points not requiring jumps over word fragments. One of many reasons our system is much more compact than its immense dictionary (1600 words) would imply.

INCREDIBLE FLEXIBILITY

The Rosetta Stone Dynamic Linker opens the world of utility libraries. Link to resident routines or link & remove routines interactively. HS/FORTH preserves relocatability of loaded libraries. Link to BTRIEVE METAWINDOWS HALO HOOPS ad infinitum. Our call and data structure words provide easy linkage.

HS/FORTH runs both 79 Standard and Forth 83 programs, and has extensions covering vocabulary search order and the complete Forth 83 test suite. It loads and runs all FIG Libraries, the main difference being they load and run faster, and you can develop larger applications than with any other system. We like source code in text files, but support both file and sector mapped Forth block interfaces. Both line and block file loading can be nested to any depth and includes automatic path search.

FUNCTIONALITY

More important than how fast a system executes, is whether it can do the job at all. Can it work with your computer. Can it work with your other tools. Can it transform your data into answers. A language should be complete on the first two, and minimize the unavoidable effort required for the last.

HS/FORTH opens your computer like no other language. You can execute function calls, DOS commands, other programs interactively, from definitions, or even from files being loaded. DOS and BIOS function calls are well documented HS/FORTH words, we don't settle for giving you an INTCALL and saying "have at it". We also include both fatal and informative DOS error handlers, installed by executing FATAL or INFORM.

HS/FORTH supports character or blocked, sequential or random I/O. The character stream can be received from/sent to console, file, memory, printer or com port. We include a communications plus upload and download utility, and foreground/background music. Display output through BIOS for compatibility or memory mapped for speed.

Our formatting and parsing words are without equal. Integer, double, quad, financial, scaled, time, date, floating or exponential, all our output words have string formatting counterparts for building records. We also provide words to parse all data types with your choice of field definition. HS/FORTH parses files from any language. Other words treat files like memory, nn@H and nn!H read or write from/to a handle (file or device) as fast as possible. For advanced file support, HS/FORTH easily links to BTRIEVE, etc.

HS/FORTH supports text/graphic windows for MONO thru VGA. Graphic drawings (line rectangle ellipse) can be absolute or scaled to current window size and clipped, and work with our penplot routines. While great for plotting and line drawing, it doesn't approach the capabilities of Metawindows (tm Metagraphics). We use our Rosetta Stone Dynamic Linker to interface to Metawindows. HS/FORTH with MetaWindows makes an unbeatable graphics system. Or Rosetta to your own preferred graphics driver.

HS/FORTH provides hardware/software floating point, including trig and transcendentals. Hardware fp covers full range trig, log, exponential functions plus complex and hyperbolic counterparts, and all stack and comparison ops. HS/FORTH supports all 8087 data types and works in RADIANS or DEGREES mode. No coprocessor? No problem. Operators (mostly fast machine code) and parse/format words cover numbers through 18 digits. Software fp eliminates conversion round off error and minimizes conversion time.

Single element through 4D arrays for all data types including complex use multiple cfa's to improve both performance and compactness. Z = (X-Y) / (X + Y) would be coded: X Y - X Y + / IS Z (16 bytes) instead of: X @ Y @ - X @ Y @ + / Z ! (26 bytes) Arrays can ignore 64k boundaries. Words use SYNONYMs for data type independence. HS/FORTH can even prompt the user for retry on erroneous numeric input.

The HS/FORTH machine coded string library with up to 3D arrays is without equal. Segment spanning dynamic string support includes insert, delete, add, find, replace, exchange, save and restore string storage.

Our minimal overhead round robin and time slice multitaskers require a word that exits cleanly at the end of subtask execution. The cooperative round robin multitasker provides individual user stack segments as well as user tables. Control passes to the next task/user whenever desired.

APPLICATION CREATION TECHNIQUES

HS/FORTH assembles to any segment to create stand alone programs of any size. The optimizer can use HS/FORTH as a macro library, or complex macros can be built as colon words. Full forward and reverse labeled branches and calls complement structured flow control. Complete syntax checking protects you. Assembler programming has never been so easy.

The Metacompiler produces threaded systems from a few hundred bytes, or Forth kernels from 2k bytes. With it, you can create any threading scheme or segmentation architecture to run on disk or ROM.

You can turnkey or seal HS/FORTH for distribution, with no royalties for turnkeyed systems. Or convert for ROM in saved, sealed or turnkeyed form.

HS/FORTH includes three editors, or you can quickly shell to your favorite program editor. The resident full window editor lets you reuse former command lines and save to or restore from a file. It is both an indispensable development aid and a great user interface. The macro editor provides reuseable functions, cut, paste, file merge and extract, session log, and RECOMPILE. Our full screen Forth editor edits file or sector mapped blocks.

Debug tools include memory/stack dump, memory map, decompile, single step trace, and prompt options. Trace scope can be limited by depth or address.

HS/FORTH lacks a "modular" compilation environment. One motivation toward modular compilation is that, with conventional compilers, recompiling an entire application to change one subroutine is unbearably slow. HS/FORTH compiles at 20,000 lines per minute, faster than many languages link — let alone compile! The second motivation is linking to other languages. HS/FORTH links to foreign subroutines dynamically. HS/FORTH doesn't need the extra layer of files, or the programs needed to manage them. With HS/FORTH you have source code and the executable file. Period. "Development environments" are cute, and necessary for unnecessarily complicated languages. Simplicity is so much better.

HS/FORTH Programming Systems

Lower levels include all functions not named at a higher level. Some functions available separately.

Documentation & Working Demo	(3 books, 1000 + pages, 6 lbs)	$ 95.
Student		$145.
Personal	optimizer, scaled & quad integer	$245.
Professional	80x87, assembler, turnkey, dynamic strings, multitasker RSDL linker, physical screens	$395.
Production	ROM, Metacompiler, Metawindows	$495.
Level upgrade, price difference plus		$ 25.
OBJ modules		$495.
Rosetta Stone Dynamic Linker		$ 95.
Metawindows by Metagraphics (includes RSDL)		$145.
Hardware Floating Point & Complex		$ 95.
Quad integer, software floating point		$ 45.
Time slice and round robin multitaskers		$ 75.
GigaForth (80286/386 Native mode extension)		$295.

HARVARD SOFTWORKS

PO BOX 69
SPRINGBORO, OH 45066
(513) 748-0390

(Continued from page 5.)

~<name>variable	
'<name>constant	
F~<name>	floating-point variable
F'<name>	floating-point constant
1A<name>	one-dimensional array
2A<name>	two-dimensions array
STR.<name>	structure, but...
$.<name>	...string address
DEF.<name>	deferred action
TOK.<name>	token
G~<name>	global variable
G'<name>	global constant
L~<name>	local variable
L'<name>	local constant
	(Inside a definition, locals don't need these; but consistency is what makes any convention work.)
^<name>	Offset which is automatically added. (Usually an offset inside a structure, but may be elsewhere.)

Figure Two.

Its most current incarnation is for F-PC by Tom Zimmer, et al., and is largely in assembler for speed.

I believe F-PC is the most significant Forth ever released into the public domain and that Tom, Bob Smith, Dr. Ting, and everyone else involved in bringing it to fruition deserves our deepest gratitude.

In light of the impact that F-PC will make on the Forth community over the next few years, I am in the process of converting all of my tools and utilities over to it. I will be glad to share whatever is currently ready (which should be substantially more than the string package by the time you read this) for $10 to cover my costs. This code is in the public domain and may be freely distributed.

Ron Braithwaite
Software Engineering
11501 N. Poema Place #101
Chatsworth, California 91311

Notes on Forth Style

Dear Editor,

Over the past few weeks, a number of discussions have been held with our programmers concerning problems learning the existing code. These discussions have brought up a number of ideas, concepts, and solutions. I will try to document some of them, as well as point out some subjects to consider.

The Code as it Exists Now...

The code structure currently represents the manner in which it was created. The older way to describe this code was the term "spaghetti code." Not a pleasant name, but correct in this case. How did it get this way?

Most of the code was produced over a long period of time by a number of different programmers. Each programmer had a particular style, which was further modified due to time constraints. Much of the coding was done on site, with deadlines. The experience level of the programmers was also very high, and these people assumed all other programmers would be equally well versed on coding principles.

In fact, the actual code produced worked well but lacked structure and comments. The problem arising now is the maintainability of the code. The ability to maintain code is what has given Forth a bad rap, and is causing this organization a considerable number of hours to bring new programmers up to speed.

What is Maintainability?

The ability to maintain code is more than having an experienced programmer able to modify it. Most organizations' current feeling is that, once the code structure is understood, the programmer can then do the required programming. The operation, then, is to provide some hand-holding with a new programmer until they learn the system. This assumes that the original programmer is still around and available whenever a new programmer needs him.

In reality, the old programmers are always working on something else, even if they are still an employee. If most organizations only keep programmers for two years (an industry average for life with a company), any project more than two years long will have several different programmers working on it. For maintainability to exist, all code must be able to stand by itself. That means, should the programmer die, a new programmer should be able to pick up where he left off. All comments and documents should be available as the work progresses.

Maintenance and Forth

Forth has several features that, if implemented, can provide for this type of documentation. By Forth's nature, any tool not available can be created and most often is. This in itself is a major problem: too many tools. Some of those tools are indexes, comment screens, locate words, and code comments.

The screen orientation of Forth enables the program to be broken down into individual sections. Each section can then contain one operation. An example of this is `DISKING` operations in polyFORTH. By typing `DISKING LOAD` all the functions associated with disk operations are loaded. This in itself is a good and quick way to handle the selection of options. After disking is loaded, a new help screen appears and the programmer knows what he can and cannot do.

This, however, assumes you know that what you want to do in on the disking screens. If you actually wanted to play with a hard disk instead, there is no way to know that you need to load `PARTITION`. If you look at the book about the subject, yes, you will learn the right word; without the book, your only hope is the comment screens.

For every screen of code, a screen of comments has been set aside. Programmers, however, are in a hurry, already know what they want to do, and usually leave such screens blank. This means that, without a book or comments, you must go through all the screens to find out what commands are necessary to perform the operation you need. For this function, Forth has the `INDEX` command. This lists the first line of each screen. If the programmer put sufficient information on the first line, indexing will give you some idea where to start searching for the command you need.

Many programmers in a hurry, however, quite often just put a simple word or two describing the whole screen. This can

at times be more cryptic than the actual code. Assuming the programmer had, in fact, not used the comment line at all, indexing would be useless.

Some Solutions

The first and foremost solution to solve the problems is to fully use the resources Forth does provide. The most important of these is the comment screen. If a standard were to be set, I personally would push for no screen code until some basic statement is put in the screen. This is ideally suited for the beginning of a group of operations. In Forth, it is typical to group operations and to have the first screen load the other parts needed by the particular task. The total of all the screens loaded may not be in order, but they are all noted and loaded from that one screen. The comment screen for this load screen should, without fail, describe the actual desired results, input options, and output conditions.

This does two things. First, it sets the stage for what is being programmed. True, many programmers will either have a clear idea of what is going on in these screens or will already have flowcharted the procedure. In either case, what goes into the first comment screen is the pre-planning ideas. Major changes to the code may occur, but rarely to the overall design idea behind the operation.

The index line on the code screen is often not used. A well-done code screen lists on the index line all the words defined in that screen. Never is any actual code listed on the index line. The INDEX option prints this entire line; when done properly, it thus provides an index listing of words used in the system. Again, when the first screen is set, it should only load other screens; its index line would state what functions are to be performed by the set of screens it loads. Then the following indexes would be of the words used for those functions.

On using load screens, some changes from the polyFORTH way might be considered. Currently, they have one load screen located at nine that loads all other groups of screens. It will provide a help screen that explains what other load screens are possible. What I might suggest is changing this structure somewhat. If the program is something like a tree structure, the loaded screen can happen only after a certain number of screens are loaded. Then these can load other screens. Currently, the location of these screens can be anywhere in the entire file structure. All that is maintained is a pointer to the first load screen.

A possibly cleaner method would be to have all the load screens in one location. That means nine becomes the top of the tree. It would load pointers to 10, 11, 12, and 13. Nine pointers can be anywhere, 100, 145, 160, or 260. The speed of the machine makes it quite easy to move between these screens. What is not available, however, is structure. By being able to look at the first eight or 10 screens and see all possible options and load conditions, the ease of maintenance has been improved many times over.

Thinking in terms of comment screens, this means the entire structure of the program would be explained in the first eight or 10 screens. When printed listings are made (as they always are), the entire structure is displayed in the first few pages of the listing. No longer does it become necessary to flip through many pages in order to see what function a load operation will invoke.

I have yet to find a programmer who can drop a project and return several years later (even six months) and pick up exactly where he left off. All programmers are usually too busy moving on to new topics; returning to an old program takes some mind refreshing. Structure and comments are absolutely necessary for this.

Some Other Points

The ability to maintain code, therefore, depends on how well the comment screens have been maintained and how well other documents have been retained. It is most important that all worksheets and flowcharts be kept in a separate file. Even if this were done, it is seldom that a user could understand it some time later. This means that comments in the source code are still the highest priority.

An area that needs more structure is the code comments themselves. Many programmers who do use the comment screen don't put anything in the actual code. Most often, when code comments are made it is simply in the form of stack comments. On many applications, the stack comment is most important. In assembly coding, however, the stack comment alone becomes meaningless. Considerably more information about the code is needed.

Assembly coding is the high-speed operation and, as such, a number of shortcuts and special uses may occur. We do not want to say that these are not to be used; instead, it is pointed out that the objective is to make these special sections of code maintainable. This can only be done by having a very full comment screen (almost a flowchart), or by putting lots of short comment statements in the actual code. A case in point was setting a flag. This operation was done by incrementing a register. If a value had been pushed into the register, it becomes very obvious what that number is. When the numbers are incremented to the next value, however, it becomes very hard to see the operation.

The objective is to allow other programmers (and you, some years later) to see what is happening. Programming time costs money. Commenting time costs money. Maintenance time costs money. Loss of an employee (even expansion) forces you to bring someone else up to speed. That speed, which is money being spent, is controlled directly by how much and how well the previous code has been documented. The poorer the documentation and comments, the less likely a new programmer is going to be able to come up to speed on the program quickly. It may even happen that an existing program must be completely rewritten if no documentation exists on the program.

A Parting Shot

In the last few years, considerable pressure has been placed on setting standards in the programming community. Most of the pressure has been on the structure of Forth. My feeling is that too little, if any, pressure is being put on establishing style conventions. The style of Forth programmers has caused many complaints and loss of contracts. The solution to the problem is not better or more-structured Forth kernels, but better control over documenting and commenting what has happened. If Forth is to become more important, it needs to have more structure in the way people use it daily. This means using all the tools that can provide that information. Structure the code, but don't forget to structure the comments as well.

Bill Kibler
Kibler Engineering
1850 McCourtney Road
Lincoln, California 95648

SELF-CHECKING NUMBERS

MICHAEL HAM - SANTA CRUZ, CALIFORNIA

People make mistakes when transcribing data. Typographical errors in words are usually easy to see, since our written language includes considerable redundancy. "Febrary" is recognizable as a typographical error for "February" because the remaining letters are enough to identify the intended word. But when numbers are involved, typographical errors are almost always difficult to detect. Normally, you cannot misspell a number: 3578 represents a number as much as 35678 does—but not the same number. Numerals, unlike written words, have little redundancy, so errors are not self-evident.

Worse, numbers are particularly prone to error when copied or entered. The most common errors in transcribing numbers are single-digit substitution errors (36548 instead of 36578), single transpositions (exchanging adjacent digits: 35678 instead of 36578), and double transposition errors (exchanging digits across a column: 35687 instead of 38657).

Because numeric information tends to be important (bank account numbers or amounts of money, for instance), various schemes are used to guard against errors. The simplest tactic is to punctuate the number so that it is broken into shorter, more manageable chunks. Commonly used punctuation includes the comma (5,345,387) and the hyphen (telephone numbers: 423-4175; Social Security numbers: 123-45-6789). Note that these punctuation marks, unlike the period that acts as a decimal point, carry no meaning so far as the number is concerned: they are simply a convenience to the user and a guard against errors.

When numbers are represented digitally, there are schemes that allow for both error detection and error correction. The phrase "single-bit error correction, double-bit error detection" refers to digital encoding systems that do exactly what they say: errors of a single bit are not only detected but corrected; errors involving two bits can at least be detected, though not corrected.

Most schemes that involve human handling of numeric information are satisfied with error detection and do not go so far as error correction. These schemes rely on redundancy; for example, from the redundant information in the phrase, "Sunday, August 10, 1987" you can know that *something* is awry: the specified date is a Monday, not a Sunday. The recipient thus knows that there is a problem and better information is needed. When sending numeric information via Telex, one technique is to spell the number as well as send it numerically: twenty-three (23) widgets. The same approach is used in writing checks: the amount of money is spelled out as well as represented in numerals. The redundancy is explicit.

The routines are ... simple and the protection is significant.

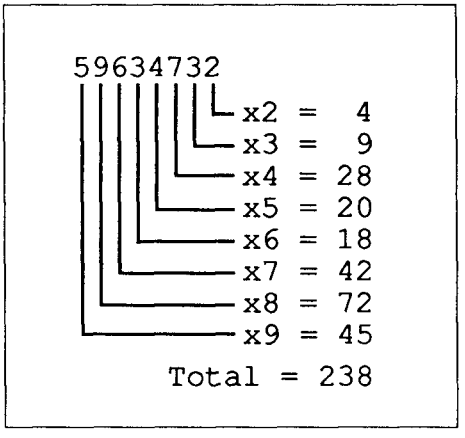
This article describes another method for detecting errors when the number is an identification number rather than an amount: the "check digit," a single digit created from the number and then embedded in the number, typically as the units digit. The recipient can recompute the check digit from the other digits of the number. If the computed check digit does not match, there is definitely an error; if the computed check digit does match, there may or may not be an error, but probably not. Because the added digit changes the value of the number, this system is not generally useful when the number represents amounts, but it works fine as part of identification numbers, such as account numbers, part numbers, student I.D. numbers, and the like.

I include Forth code to compute and check the check digit, along with some observations on special requirements imposed by the tools available in standard Forth products.

As a weak example of a check digit, try the units digit of the sum of the digits: the check digit for 12840 is then 5 (1+2+8+4+0 = 15, with units digit 5). The number with the check digit is then 128405. If the number is received as 138405, there is a problem: the check digit *computed* from the received number is 6, but the *received* check digit in the number is 5. This particular check digit method is hopelessly weak, however: it does not catch transpositions, the most common type of error: 12840 gives the same check digit with this method as 21840.

A much more powerful check digit, which is widely used, is the mod-11 check digit. "Mod 11" (or "modulus 11") refers to the remainder after division by 11. In this method, you multiply each digit of the number by a weight, sum the products, and find the remainder upon dividing that sum by 11. If the remainder is zero (i.e., the sum is evenly divisible by 11), then the check digit is zero; otherwise, the check digit is the difference between 11 and the remainder. The weights, sequentially assigned

from the right-most digit, are 2, 3, 4, ..., 9, 10, as shown below:



238/11 = 21 with a remainder of 7 or, equivalently, 238 mod 11 = 7. So the check digit is 11-7 = 4.

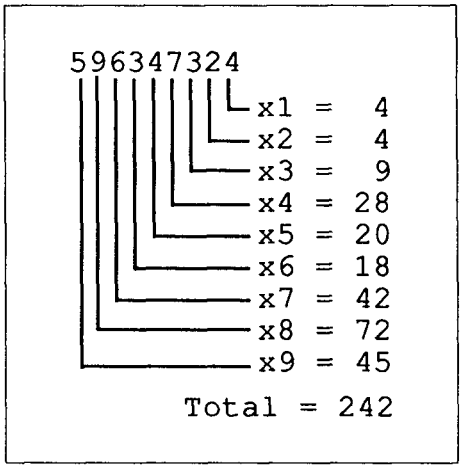
The resulting number (with the check digit appended) is 596347324, which might also be punctuated in some way: 5963-473-24. The original eight-digit base number has become a nine-digit number. Very seldom does one start with a base number of more than nine digits (for which weights above 10 would be needed), but in such cases one could start the weights again at 2.

One question immediately arises: what happens when the remainder is 1? The difference to be appended in that case is 10—two digits instead of one. Usually, numbers that give a check digit of 10 are simply discarded, which means about 9% of the numbers in the range are unavailable. Sometimes these numbers are used with a check digit of some special symbol; the International Standard Book Number (ISBN) code, for example, is a nine-digit code with a mod-11 check digit, and it uses the letter "X" as check digit when the check digit value is 10. But most programs want only numeric values in numeric fields and, therefore, discard numbers that would have a check digit of 10.

Another question: why bother subtracting non-zero remainders from 11? Why not just use the remainder itself, discarding numbers with remainder 10 instead of those with remainder 1?

The difference between the remainder and 11 is exactly the amount that must be added to the sum of the weighted digits to get a multiple of 11. That is, the sum of the weighted digits *including* the check digit (defined as 11 minus the remainder and multiplied by a weight of 1) will be exactly divisible by 11 if the number was correctly received. This property simplifies the checking of the check digit. In most situations, the check digit is checked many more times than it is assigned, so that one wants to simplify the checking rather than the assignment.

To check the number above, for instance, you compute again, this time with the weight 1 for the right-most digit, which is now the check digit:



242/11 = 22 remainder 0 or, equivalently, 242 mod 11 = 0; the check digit checks out okay.

The mod-11 check digit is used for the ISBN code mentioned above, and is also commonly used for bank account code numbers. Whenever you design a system in which people must enter identification numbers—part numbers, student ID numbers, employee numbers, project numbers—it makes sense to use a check digit so that entry errors can easily be detected, ideally at the time of entry.

Since the mod-11 check digit discards only about 9% of the numbers from the planned range, you might think that a random error would result in a valid number (even with the check digit) about 91% of the time. But note that the added check digit makes the range of numbers an order of magnitude greater.

For example, suppose that you are assigning part numbers. There are about 20,000 parts currently to be tracked, so a five-digit number is ample. You decide to start with 10000 as the first part number to avoid leading zeroes (because not having leading zeroes causes fewer errors). All part numbers will then be five digits (before adding the check digit). The range 10000 through 99999 gives 90,000 numbers, and even discarding 9% of them (when the check digit is 10) leaves 91% available, or 81,900.

That, however, is 91% of five-digit numbers. With the check digit added, you in fact have a six-digit number. The first valid part number (with check digit and punctuation) will be 100-005 and the last 999-997. Thus, the part numbers are spread across an effective range of 100,005 through 999,997, and of the 899,993 numbers in this range, only 81,900 will be used, or about 9%. So a random error will hit a valid number only about 9% of the time.

The check digit offers more protection than that, however, because the mistakes people make are not random. The most common error is a single-digit substitution error; single transpositions and double transpositions are the next most common. Table One tells the story.

It is true that random errors (the "other" errors in the table) will slip by 9% of the time, but random errors comprise only about 5% of all errors.

So much for background; now for application. My preferred programming language is Forth, and in implementing words for the above scheme, we face Forth's typical dearth of double-precision operators.

Double-precision numbers ("doubles") are necessary in this application because single-precision numbers have a range of only 0 through 65,535—not enough even to implement check digits on a four-digit base (which becomes five digits with the check digit added, and thus can range larger than 65,535).

Doubles go as high as 4,294,836,225, so they can readily be used with eight-digit base numbers (which become nine digits with the check digit). In practice, one seldom needs code numbers longer than nine digits (including the check digit): a nine-digit code number produces around 82,000,000 valid code numbers. Marking errors increase as numbers get longer, so the designer generally wants to use as short a code number as possible. In my experience, six-digit code numbers (about 82,000 valid possibilities) are often enough.

Given that doubles will be used, one naturally wants operators that permit one to chip off the digits one by one to compute the check digit and stick it onto the end. `D/MOD` and `D*` (accepting doubles for

both arguments and returning double results) would be very useful. Most Forths lack such numbers. Standard Forth has some few mixed operators that allow a double and a single argument, but these return single results.

One Forth that has provided a complete solution to the need for higher-precision operators is MMSFORTH (published by Miller Microcomputer Services). MMSFORTH includes an extension called N-LEN#. This collection of operators parallels all the usual number and stack operators, and allows you to define the precision (single, double, triple, quadruple, or whatever) with the word #PREC, which defines the number of cells to be used in the arithmetic operations. By setting #PREC to 2 and using #* and #/MOD, one can easily write a set of check-digit words for double-precision numbers.

Working within the bounds of the

```
: DIGITSUM      ( d -- n )
  <# #S #>      ( convert double to ASCII string, )
                ( leaving address and count on stack: adr n -- )
  TUCK          ( same as SWAP OVER; stack is now:  n adr n -- )
  1- +          ( the address of the right-most digit in the string; )
                ( stack is now: n adr+n-1 -- )
  0 ROT         ( sum will be accumulated on the stack; )
                ( the sum starts with 0; stack: adr+n-1 0 n -- )
  0 DO          ( the limits of the DO loop are n and 0; )
                ( stack inside the loop starts as: adr+n-1 0 -- )
  OVER          ( copy address to top of stack )
  I -           ( move one digit leftward on each iteration )
  C@            ( fetch the ASCII character )
  ASCII 0 -     ( convert it to its numeric value )
  I 1+          ( this is the weight: 1, 2, 3, etc. )
  *             ( multiply digit by weight )
  +             ( add to accumulating sum )
  LOOP          ( loop back for next )
  NIP  ;        ( same as SWAP DROP; get rid of the address )
```

Figure One-a. This word converts a double number to ASCII for single-digit (character) manipulation.

```
: DIGITSUM  ( d -- n ) ( weighted sum of digits for mod-11 check )
  <# #S #>  TUCK  1- +  ( adr of rtmost digit )  0  ROT 0
  DO OVER  I - C@  ASCII 0 -  I 1+ *  +  LOOP  NIP ;
```

Figure One-b. Less verbose version of DIGITSUM.

```
: MAKEDIGIT     ( d -- d' f )
                ( T if okay, F if no good )
  D10*  2DUP    ( two copies of 10 times the original number )
  DIGITSUM      ( one copy is consumed by DIGITSUM, )
                ( which leaves the weighted sum on the stack )
  11 MOD        ( replace sum with the remainder on division by 11 )
  DUP           ( two copies of the remainder )
  1 =           ( see whether remainder is 1 )
  IF  DROP      ( if it is, drop it )
    FALSE       ( and leave "false" flag )
  ELSE DUP      ( otherwise, duplicate remainder to be a flag )
    IF
      11 SWAP -         ( subtract non-zero remainders from 11 )
    THEN
      US>D              ( convert to double precision )
    D+                  ( add to 10 * original number, still on stack )
    TRUE                ( and put "true" flag on top )
  THEN ;
```

Figure Two-a. This word leaves a flag and a double on the stack.

```
: MAKEDIGIT  ( d - d' f )  ( T if okay, F if no good )
  D10*  2DUP  DIGITSUM  11 MOD  DUP
  1 = IF  DROP FALSE
    ELSE DUP  IF 11 SWAP - THEN
    US>D  D+ TRUE THEN  ;
```

Figure Two-b. Less verbose version of MAKEDIGIT

Error type	Typical error incidence	Expected detection with mod-11 check digit
Single-digit substitution	86%	100%
Single transposition	8%	100%
Double transposition	1%	100%
Other	5%	91%
Total all errors	100%	99.5%

Table One. Incidence of various types of errors.

Forth-83 Standard, one must find other ways to skin this particular cat. In the following, I'll develop the check-digit words step-by-step from standard Forth.

I first want a word to accept a double on the stack and return the weighted sum of the digits. I'll use this sum one way when I am creating a new code number (appending the check digit) and another way when I am checking the check digit (for a code number entered by some user). I will presume that the units digit of the number I am working with is either the check digit or a placeholder of 0 for the check digit, and this in either case can be multiplied by 1 and added to the sum. In keeping with Forth practice, this word will consume its argument, replacing the double on the stack with the weighted sum of its digits.

The method I employ is to convert the double number to a string of ASCII digits and then retrieve the digits one at a time as I need them, converting each back into its numerical value. The word DIGITSUM works as documented in Figure One-a.

The word ends with the sum of the weighted digits on the stack as a single-precision number. The definition is not so verbose in my actual source-code file; it looks like Figure One-b, with spacing indicating the phrases.

The phrase ASCII 0 - would deserve a separate definition just for readability, if its purpose were not so obvious. Sometimes, though, one wants to create a definition to improve readability. For instance, to append the check digit to the base number, we want to add the check digit (single precision) to ten times the base number (double precision). To use D+, both numbers must be double-precision, so we need to convert the single-precision unsigned number to a double. I can just put 0 on the stack above the single-precision number, but because the purpose of the 0 in the line of source code might be unclear, I prefer to define the operator US>D as 0 CONSTANT US>D.

Using a constant has another advantage: I will save room in the source code if I use the definition several times. The real motivation, however, is readability.

I need to multiply the (double-precision) base number by 10 to append the check digit. It is easy to fake a 10. D* using addition:

```
: D10* ( d -- d' )
  2DUP 2DUP D+ 2DUP
  D+ D+ 2DUP D+ ;
```

I prefixed D to the name so that it is clear that the word operates on doubles. As you can see, I simply duplicate and add in the proper sequence to get 2 times the number, 4 times the number, 5 times the number and, finally, 10 times the number.

It is easier to check the check digit than create it, so let's first write the check word:

```
: CHECKOK?  ( d -- f )
  ( T = okay, F = doesn't check )
  DIGITSUM 11 MOD 0= ;
```

The check digit checks out okay if the weighted sum (including 1 times the check digit) is divisible by 11—that is, the number mod 11 is zero. The 0= at the end converts the number to a true Boolean (0 or -1) and also reverses the truth value: I wanted "true" to mean that no error was detected. You can substitute 0<> for 0= if you want "true" to signify that an error was

(Continued on page 37.)

ProDOS

APPLE II $FORTH

CHESTER H. PAGE - SILVER SPRING, MARYLAND

Much as I love and admire Forth, whenever I need a program that involves string manipulation, I have been going back to a combination of BASIC (Applesoft) and assembly code. This regressive step injured my pride, so I designed a Forth application that handles strings as fluently as does BASIC.

$Forth is built on the following key points:

The string text (undimensioned) is stored downward in high memory. That is, the first character of a string is stored at a higher address than the last character.

A string variable has a two-address value. The operation `$A $@` returns a pair of addresses—the addresses of the first and last characters of the string identified as `$A`.

Strings are loaded from the input stream in the normal way of entering words, and then is copied in reverse order to the highest available memory space, just below any previously stored strings. This reverse-order storage of pure text allows simple, elegant operations such as extracting portions of strings and concatenating strings (for example, `n RIGHT$`, `n LEFT$`, `n1 n2 MID$`, and `$+`). The string-extraction operations involve only changing the pointers in the string-variable values.

A separate string stack contains the pairs of pointers. Thus, `$A` returns the normal single address of a variable (to the parameter stack); `$@` goes to this address, reads the two addresses stored there, and puts this pair onto the string stack.

Garbage collection: as strings are redefined by manipulation or text replacement, their former versions still occupy memory. When available memory is almost used up, garbage collection is performed. This discards all obsolete text and moves the active text upward so that memory is the same as if the final versions of the strings had been the original entries.

To simplify the mechanics of garbage collection, string variables are linked in a simple chain for fast inspection. This is done by having all string variables—and nothing else—in a separate vocabulary named `STRINGS`. Defining a string variable (e.g., by entering `$VARIABLE $A` or `n $ARRAY $B`) automatically puts the defined variable in the `STRINGS` vocabulary. Note that the length of a string is not specified in the definition, as it is in Pascal and some string versions of Forth. This results in major memory savings when dealing with an array of strings, of which one or two may be much longer than the others.

Strings are saved in sequential text files.

When writing a colon definition which refers to string variables, it is necessary to insert `[ STRINGS ]` in the definition, preferably as the first component. This is because the colon sets the `CONTEXT` vocabulary to be the `CURRENT` vocabulary, normally `FORTH`, and the string variable name would not be recognized. For example:

```
: TEST
  [ STRINGS ]
  $A $@  $B $@
  $+  $C $!  ;
```

The basic words in $Forth are given in Figure One. Keyboard string input is handled by an analog of the `. (` operator:

```
: $(  ( -- ; -- A1 A2 )
  ?EXEC  29 WORD COUNT
  $WRITE  ;
```

Thus, entering `$( ABC)` will record the text ABC (from `FRETOP` downward), leave the address holding C on top of the string stack, and the address holding A as next on the string stack. Strings included in a definition being compiled use the immediate `$" ABC"` analog to `." ABC"`.

The key command `$WRITE` copies n characters starting at address A (which is `HERE 1+`) and records them in high memory from A1 down to A2.

`$.` prints the text from A1 down to A2, inclusive.

`$VARIABLE A$` defines `A$` as a string variable *in the `STRINGS` vocabulary.*

`n $ARRAY A$` defines `A$` as an n-component array string variable, `1 A$` to `n A$`.

The difference between `$DUP` and `$COPY` should be noted. `$DUP` simply duplicates the pointers, whereas `$COPY` duplicates the text, locating the text copy at the next available text location.

`$STACK` provides a non-destructive printout of all the addresses on the string stack.

The remaining stack operations are self-evident analogs of parameter stack manipulations.

Disk Files

Strings are saved in sequential text files, separated by a string-end marker. This marker can be a printable character, such as *; a period, to make each sentence a string; a return, to make each paragraph a string; or even a non-printable character,

such as Ctrl-A, to serve as an invisible delimiter. Note that strings can contain any characters desired. The mark is entered by $MARK which requests entry of the mark. If this is done after a file is opened, the file can be read one string at a time by IN$. For writing files, a string on the stack (pointed to by the address on top of the string stack) is moved to the massbuffer by OUT$ which automatically adds the delimiter.

Files are opened (and created, if not already present) by OPEN <name>. Using OPEN as an immediate keyboard command automatically calls $MARK to set the end-of-line information for the open file. Files can also be opened by a word, e.g.:

```
: TEST
  OPEN <filename>
  $MARK <char> ;
```

which can be defined in a screen or from the keyboard.

IN$ will now read the first string in the file; repetition will read the remaining strings. For writing, WRITE must be entered after a file is opened. After all desired strings have been "printed" with OUT$, entering CLOSE will flush the output buffer and close the file. (When the string text fills the massbuffer, it will automatically be saved to disk. CLOSE is only needed for a "mop-up.") To avoid errors arising from forgetting to close a file after reading or writing, OPEN automatically closes the previously accessed file as its first step, flushing only if there is an open file and if WRITE has been entered after that file was opened.

Realization for the 6502

The preceding section on reading and writing disk files is, obviously, completely installation dependent. The other parts of $Forth, however, can be made available in a crude form for testing. If you like the features, they can be incorporated into a complete Forth realization.

Providing non-Apple-specific code for experimental use is a little awkward, because the key words for stack manipulation are primitives; thus, specific addresses must be supplied (unless a Forth assembler is to be used). I am appending ordinary assembly listings for the 6502 CPU, with address equates which should be understood as being illustrative, although two of them are determined by the version of Forth onto which the string routines are to be grafted. These are the addresses of NEXT

String-stack operators

```
$(        ( -- ; -- A1 A2 )
$@        ( A -- ; -- A1 A2 )
$!        ( A -- ; A1 A2 -- )
$SWAP     ( A1 A2 A3 A4 -- A3 A4 A1 A2 )
$DROP     ( -- ; A1 A2 -- )
$DUP      ( -- ; A1 A2 -- A1 A2 A1 A2 )
$COPY     ( -- ; A1 A2 -- A3 A4 )
$STACK    ( -- ; A1 ... An -- A1 ... An )
$.        ( -- ; A1 A2 -- )
$WRITE    ( A n -- ; -- A1 A2 )
```

String manipulators

`$+`	`( A1 A2 A3 A4 -- A5 A6 )`	Concatenation.
`$LEN`	`( -- n ; A1 A2 -- A1 A2 )`	Non-destructive operation to put the length of a string onto the parameter stack.
`RIGHT$`	`( n -- ; A1 A2 -- A3 A4 )`	Returns the last n characters of a string.
`LEFT$`	`( n -- ; A1 A2 -- A3 A4 )`	Returns the first n characters of a string.
`MID$`	`( n n1 -- ; A1 A2 -- A3 A4 )`	Returns n1 characters beginning with the nth character.

Figure One.

```
STRINGS SCR # 1
 0 \ General Notes                                          25JUL88CHP
 1 \ In screen #2, the 90FF value for FRETOP0 can be changed to
 2 \ any convenient high-memory location.  Strings are stored
 3 \ from this location downward.
 4
 5 \ Boot FORTH
 6
 7 \ If your FORTH does not recognize the dummy word NEXT, find
 8 \ the address, nnnn, to which all primitives jump as a final
 9 \ step, and replace GONEXT in all screens with nnnn JMP,
10
11 \  Define : DUMMY ; to be used for linking to discard the
12 \           ASSEMBLER vocabulary
13 \ Enter HEX 2000 ALLOT, load ASSEMBLER, then load STRINGS
14
15   -->


STRINGS SCR # 2
 0 \ Special VARIABLES and CONSTANTS                        25JUL88CHP
 1 HEX
 2 E6 CONSTANT N   \ This must be a zero-page location
 3 \ It is a temporary constant for the assembler
 4 ' DUMMY 4 + DP !
 5 \ From here on, add all new words to the FORTH dictionary
 6 90FF CONSTANT FRETOP0     VARIABLE FRETOP
 7 : FRETOP! FRETOP0 FRETOP ! ;
 8 VARIABLE XSAVE     VARIABLE TEMP      VARIABLE TEMP1
 9 VARIABLE STRING.X   80 CONSTANT SS0   5 CONSTANT SS1
10 VARIABLE S.STACK    80 ALLOT
11 VARIABLE GARB      100 ALLOT
12 : SSP! SS0 STRING.X C! 0 0 ! ;
13 : SSP@ STRING.X C@ ;
14 -->
15
```

and a new variable FRETOP which is to be created as the first step in adding the string routines. The FRETOP address is found by reading your dictionary pointer immediately after booting, or by entering VARIABLE FRETOP and using FRETOP to find its address. Incidentally, this is a good example of the virtue of using an assembler written in Forth: FRETOP can be defined as a variable and its address made available for the assembly code simply by using the word FRETOP.

XSAVE and STRING.X are any available cells; N requires a sequence of seven zero-page cells, N-1 to N+5. (In fig-FORTH, this was above the parameter stack.) TEMP and TEMP1 are two pairs of cells for temporary number and address storage.

In my CORE.FORTH, I use $900 (instead of $800) as the origin of the Forth nucleus. This makes $880–8FF available for the string stack, and $800–87F for a floating-point stack. I have my input massbuffer at $9100, so set $90FF as FRETOP0, the initial value of FRETOP. In CORE-FORTH, FRETOP is a system variable and the added system constants are SS0, SS1, FRETOP0, S.STACK, STRING.X, and GARB. GARB is the address of a one-page (256 bytes) buffer used in fast garbage collection procedures. The value of $9500 puts it just under the ProDOS buffer.

Some words comprise a simple test, followed by a primitive for the actual operation. For example, $DROP requires testing for an empty string stack first, hence:

```
: $DROP
  ?$EMPTY $DROP% ;
```

where $DROP% is the corresponding primitive. This notation is convenient.

The assembled primitives—as given, or with your own values for the address equates—are to be written as screens by your favorite method, using an assembler written in Forth or copying the binary data by hand as in the attached screens.

S.STACK was placed at $4000 in the hope that this would not interfere with an existing Forth and, hence, could be used experimentally; although it is a very poor choice for actual implementation.

```
STRINGS SCR # 3
 0 \ REV.MOVE  $@%                                          23JUL88CHP
 1 ASSEMBLE REV.MOVE 6 # LDA, N 1- STA, 101 0 ,X LDA, N ,Y STA,
 2 INX, INY, N 1- CPY,  101 BNE, 0 # LDY, XSAVE STX, STRING.X LDX,
 3    DEX, DEX, DEX, DEX, N 3 + LDA, S.STACK 3 + ,X STA, N 2+ LDA,
 4    S.STACK 2+ ,X STA, STRING.X STX, 0 # LDX, 102 N CPX,
 5    106 BEQ, 103 N 4 + )Y LDA, N 2+ )Y STA, N 2+ DEC, N 2+ LDA,
 6    FF # CMP, 104 BNE, N 3 + DEC, 104 N 4 + INC, 105 BNE,
 7    N 5 + INC, 105 INX, 102 BNE, 106 N 1+ DEC, 103 BPL,
 8    STRING.X LDX, N 2+ LDA, FRETOP STA, CLC, 1 # ADC, S.STACK
 9    ,X STA, N 3 + LDA, FRETOP 1+ STA, 0 # ADC, S.STACK 1+ ,X STA,
10    XSAVE LDX, GONEXT END
11 ASSEMBLE $@% 4 # LDY, 101 0 X) LDA, PHA, 0 ,X INC, 102 BNE,
12    1 ,X INC, 102 DEY, 101 BNE, INX, INX, XSAVE STX, STRING.X
13    LDX, 4 # LDY, 103 DEX, PLA, S.STACK ,X STA, DEY, 103 BNE,
14    STRING.X STX, XSAVE LDX, GONEXT END
15 -->


STRINGS SCR # 4
 0 \ ?$EMPTY  ?$FULL  $@  $!                                23JUL88CHP
 1 : ?$EMPTY SS0 3 - SSP@ U< ABORT"   Empty stringstack"  ;
 2
 3 : ?$FULL SSP@ SS1 U< ABORT"  Full stringstack" ;
 4
 5 ASSEMBLE $!% XSAVE STX, STRING.X LDX, S.STACK 3 + ,X LDA,
 6    PHA, S.STACK 2+ ,X LDA, PHA, S.STACK 1+ ,X LDA, PHA,
 7    S.STACK ,X LDA, PHA, INX, INX, INX, INX, STRING.X STX,
 8    XSAVE LDX, 4 # LDY, 101 PLA, 0 X) STA, 0 ,X INC, 102 BNE,
 9    1 ,X INC, 102 DEY, 101 BNE, INX, INX, GONEXT END
10
11 : $@ ?$FULL $@% ;
12 : $! ?$EMPTY $!% ;
13
14   -->
15


STRINGS SCR # 5
 0 \ $DUP  $DROP     S.STACK->P.STACK                       25JUL88CHP
 1 ASSEMBLE $DUP% XSAVE STX, STRING.X LDX, S.STACK ,X LDA,
 2    PHA, S.STACK 1+ ,X LDA, PHA, S.STACK 2+ ,X LDA, PHA,
 3    S.STACK 3 + ,X LDA, DEX, DEX, DEX, DEX, S.STACK 3 +
 4    ,X STA, PLA, S.STACK 2+ ,X STA, PLA, S.STACK 1+ ,X STA,
 5    PLA, S.STACK ,X STA, STRING.X STX, XSAVE LDX, GONEXT END
 6
 7 : $DUP ?$EMPTY ?$FULL $DUP% ;
 8
 9 ASSEMBLE $DROP% STRING.X INC, STRING.X INC, STRING.X INC,
10    STRING.X INC, GONEXT END
11 : $DROP ?$EMPTY $DROP% ;
12 ASSEMBLE S.STACK->P.STACK XSAVE STX, STRING.X LDX, 4 # LDY,
13 101 S.STACK ,X LDA, PHA, INX, DEY, 101 BNE, STRING.X STX,
14 XSAVE LDX, 4 # LDY, 102 DEX, PLA, 0 ,X STA, DEY, 102 BNE,
15 GONEXT  END      -->


STRINGS SCR # 6
 0 \ $SWAP  $WRITE  $(  $"  CHR$                           23JUL88CHP
 1 ASSEMBLE $SWAP% XSAVE STX, STRING.X LDX, 4 # LDY, 101
 2    S.STACK ,X LDA, PHA, INX, DEY, 101 BNE, DEX, 4 # LDY,
 3    102 S.STACK 4 + ,X LDA, S.STACK ,X STA, PLA, S.STACK 4 +
 4    ,X STA, DEX, DEY, 102 BNE, XSAVE LDX, GONEXT END
 5
 6 : $SWAP SSP@ SS0 8 - > IF ."  Fewer than 2 strings on stack"
 7     SSP! QUIT THEN $SWAP% ;
 8
 9 : $WRITE ( addr n---) [ 2 ALLOT ] ?$FULL FRETOP @ SWAP
10      REV.MOVE ;      \ Space left for inserting ?$GARBAGE
11 : $( ?EXEC 29 WORD COUNT $WRITE ;
12 : ($") PHRASE $WRITE ;
13 : $" 22 COMPILE ($") WORD C@ 1+ ALLOT ; IMMEDIATE
14 : CHR$ ( n---;---A A) HERE C! HERE 1 $WRITE ;
15 -->
```

```
STRINGS SCR # 7
 0 \  $.  $COPY%                                              23JUL88CHP
 1 : $. ?$EMPTY S.STACK->P.STACK 1- SWAP DO I C@ EMIT -1 +LOOP ;
 2
 3 ASSEMBLE $COPY% XSAVE STX, STRING.X LDX, S.STACK 3 + ,X
 4    LDA, PHA, S.STACK 2+ ,X LDA, PHA, S.STACK 1+ ,X LDA,
 5    PHA, S.STACK ,X LDA, PHA, FRETOP LDA, S.STACK 2+ ,X STA,
 6    FRETOP 1+ LDA, S.STACK 3 + ,X STA, XSAVE LDA, SEC, 6 # SBC,
 7    TAX, XSAVE STX, PLA, 4 ,X STA, PLA, 5 ,X STA, PLA,
 8    4 ,X SBC, 0 ,X STA, PLA, 5 ,X SBC, 1 ,X STA, SEC,
 9    FRETOP LDA, 0 ,X SBC, 2 ,X STA, FRETOP 1+ LDA, 1 ,X SBC,
10    3 ,X STA, CLC, 0 ,X LDA, 1 # ADC, 0 ,X STA, 1 ,X LDA,
11    0 # ADC, 1 ,X STA, SEC, 2 ,X LDA, 1 # SBC, FRETOP STA,
12    3 ,X LDA, 0 # SBC, FRETOP 1+ STA, 2 ,X LDA, PHA,
13    3 ,X LDA, STRING.X LDX, S.STACK 1+ ,X STA, PLA,
14    S.STACK ,X STA, XSAVE LDX, GONEXT END
15 -->


STRINGS SCR # 8
 0 \  $COPY  $VARIABLE   $ARRAY                               25JUL88CHP
 1 : $COPY ?$EMPTY $COPY% CMOVE ;
 2
 3  VOCABULARY STRINGS
 4
 5 : $VAR STRINGS DEFINITIONS CREATE 0 , 1 , 0 , 0 , DOES> 2+ ;
 6
 7 : $VARIABLE $VAR FORTH DEFINITIONS STRINGS ;
 8
 9 : ($ARRAY) STRINGS DEFINITIONS CREATE DUP 0 , , 2* 0 DO
10     0 , LOOP DOES> SWAP 1- 4 * 2+ + ;
11
12 : $ARRAY ($ARRAY) FORTH DEFINITIONS STRINGS ;
13
14 -->
15


STRINGS SCR # 9
 0 \ LEFT$  RIGHT$                                            23JUL88CHP
 1
 2 ASSEMBLE LEFT$ 0 ,X DEC, 0 ,X LDA, TEMP STA, FF # CMP,
 3    101 BNE, 1 ,X DEC, 101 1 ,X LDA, TEMP1 STA, INX, INX,
 4    XSAVE STX, STRING.X LDX, SEC, S.STACK 2+ ,X LDA, TEMP SBC,
 5    S.STACK ,X STA, S.STACK 3 + ,X LDA, S.STACK 1+ ,X STA, XSAVE
 6    LDX, GONEXT END
 7
 8 ASSEMBLE RIGHT$ 0 ,X DEC, 0 ,X LDA, TEMP STA, FF # CMP,
 9    101 BNE, 1 ,X DEC, 101 1 ,X LDA, TEMP1 STA, INX, INX,
10    XSAVE STX, STRING.X LDX, CLC, S.STACK ,X LDA, TEMP ADC,
11    S.STACK 2+ ,X STA, S.STACK 1+ ,X LDA, TEMP1 ADC, S.STACK 3 +
12    ,X STA, XSAVE LDX, GONEXT END
13
14 -->
15


STRINGS SCR # 10
 0 \ MID$   $LEN%                                             20APR88CHP
 1 ASSEMBLE MID$ ( n1 n2--- ) 0 ,X DEC, 0 ,X LDA, TEMP STA,
 2    FF # CMP, 101 BNE, 1 ,X DEC, 101 1 ,X LDA, TEMP 1+ STA,
 3    2 ,X DEC, 2 ,X LDA, TEMP1 STA, FF # CMP, 102 BNE,
 4    3 ,X DEC, 102 3 ,X LDA, TEMP1 1+ STA, INX, INX, INX, INX,
 5    XSAVE STX, STRING.X LDX, SEC, S.STACK 2+ ,X LDA,
 6    TEMP1 SBC, S.STACK 2+ ,X STA, PHA, S.STACK 3 + ,X LDA,
 7    TEMP1 1+ SBC, S.STACK 3 + ,X STA, PLA, TEMP SBC,
 8    S.STACK ,X STA, S.STACK 3 + ,X LDA, TEMP 1+ SBC,
 9    S.STACK 1+ ,X STA, XSAVE LDX, GONEXT END
10
11 ASSEMBLE $LEN% XSAVE STX, STRING.X LDX, SEC, S.STACK 2+
12    ,X LDA, S.STACK ,X SBC, PHA, S.STACK 3 + ,X LDA,
13    S.STACK 1+ ,X SBC, TEMP STA, CLC, PLA, 1 # ADC, PHA,
14    TEMP LDA, 0 # ADC, XSAVE LDX, DEX, DEX, 1 ,X STA, PLA,
15    0 ,X STA, XSAVE STX, GONEXT END   -->
```

```
STRINGS SCR # 11
 0 \ $LEN  $+  CL.GARB.BUF                                    23JUL88CHP
 1 : $LEN ?$EMPTY $LEN% ;
 2
 3 ASSEMBLE $+% XSAVE STX, STRING.X LDX, S.STACK ,X LDA,
 4    S.STACK 4 + ,X STA, S.STACK 1+ ,X LDA, S.STACK 5 +
 5    ,X STA, INX, INX, INX, INX, STRING.X STX, XSAVE LDX,
 6    GONEXT END
 7
 8 : $+ $SWAP $COPY $SWAP $COPY $+% ;
 9
10 ASSEMBLE CL.GARB.BUF XSAVE STX, 0 # LDX, TXA, 101 GARB
11    ,X STA, DEX, 101 BNE, FF # LDA, DEX, GARB ,X STA,
12    DEX, GARB ,X STA, DEX, GARB ,X STA, DEX, GARB ,X STA,
13    XSAVE LDX, GONEXT END
14
15   -->


STRINGS SCR # 12
 0 \ $LIST  GARB.COLL                                        20APR88CHP
 1 ASSEMBLE $LIST 3 ,X LDA, PHA, 2 ,X LDA, PHA, 1 ,X LDA,
 2    TEMP 1+ STA, 0 ,X LDA, TEMP STA, INX, INX, INX, INX,
 3    XSAVE STX, 0 # LDX, TEMP LDA, GARB 2+ ,X CMP, TEMP 1+ LDA,
 4    GARB 3 + ,X SBC, 103 BCC, 101 INX, INX, INX, INX,
 5    TEMP LDA, GARB 2+ ,X CMP, TEMP 1+ LDA, GARB 3 + ,X SBC,
 6    101 BCS, DEX, DEX, DEX, DEX, TEMP1 STX, 0 # LDX, 102 GARB 4 +
 7    ,X LDA, GARB ,X STA, INX, TEMP1 CPX, 102 BCC, PLA, GARB ,X
 8    STA, PLA, GARB 1+ ,X STA, TEMP LDA, GARB 2+ ,X STA, TEMP 1+
 9    LDA, GARB 3 + ,X STA, XSAVE LDX, 103 GONEXT END
10
11 : GARB.COLL GARB FC + BEGIN 4 - DUP DUP GARB SWAP U<
12       SWAP @ DUP 0= 0= ROT AND
13       WHILE DUP $@ $COPY $! REPEAT DROP DROP ;
14
15   -->


STRINGS SCR # 13
 0 \ $ACTIVE  GARBAGE  ?$GARBAGE                             27JUL88CHP
 1 : $ACTIVE CL.GARB.BUF STRINGS CONTEXT @ BEGIN @
 2       DUP @ A081 = 0=
 3       WHILE NAME> DUP 4 + DUP 2- SWAP @ 0 DO
 4       4 + DUP DUP @ DUP FRETOP @ U<
 5       IF $LIST ELSE DROP DROP THEN
 6       LOOP DROP >LINK REPEAT DROP ;
 7 : GARBAGE FRETOP! BEGIN $ACTIVE GARB F8 + @ WHILE
 8       GARB.COLL REPEAT ;
 9 : ?$GARBAGE FRETOP @ HERE - 200 < IF GARBAGE THEN ;
10   -->
11 NOTE: NAME> converts name field address to code field address
12      >LINK converts code field address to link field address
13
14 NOTE: $ACTIVE assumes that the lowest word in the STRINGS
15 vocabulary links back to 81 A0 as the dummy name in FORTH


STRINGS SCR # 14
 0 \ $PTR  $STACK  ?$GARBAGE                                 25JUL88CHP
 1
 2 ASSEMBLE $PTR XSAVE STX, TEMP LDX, S.STACK ,X LDA,
 3       PHA, S.STACK 1+ ,X LDA, XSAVE LDX, DEX, DEX, 1 ,X STA,
 4       PLA, 0 ,X STA, TEMP INC, TEMP INC, GONEXT END
 5
 6 : $STACK ?$EMPTY STRING.X C@ [ TEMP ] LITERAL C!
 7         BEGIN $PTR U. [ TEMP ] LITERAL C@ 80 = UNTIL ;
 8
 9 ' ?$GARBAGE ' $WRITE >BODY !
10
11 SSP! FRETOP!
12
13 ' DUMMY >NAME ' FRETOP0 >LINK ! \ Restore linkage
14
15 QUIT
```

Part Four

DESIGNING DATA STRUCTURES

MIKE ELOLA - SAN JOSE, CALIFORNIA

Several approaches to the design of data structures have been introduced previously. This concluding article departs from the concerns for portability and reuse of operations. Previous installments have already shown how object orientation and data abstraction can help address those concerns. With a much more personal treatment this time, I'll reconsider object-oriented programming (OOP) in a variety of ways. One way is through an exploration of my own efforts to create an object-oriented Forth (OOF).

Although much of what is presented here is based on concepts introduced before, it is enough of a digression that it requires a new conceptual framework.

In the first installment, I said that any new object defines a new data type (because it carries with it a certain set of properties which another object could duplicate). I'll introduce the term *data typing system* to refer to the existence of a system for managing operations associated with data types. Likewise, I'll use the term *object typing system* to refer to the management of operations associated with objects. Both are forms of *operation management systems*. The term object typing system was chosen because of its enforcement connotations.

Data typing systems have become well established as components of modern languages, such as C and Pascal. As far as I know, no language before object-oriented languages (OOLs) provided a management function for user-defined objects and their associated operations. Although this advancement is recognized less, it is of key importance. By managing operations that apply to particular types of user-defined objects, languages are deserving of the label "object oriented."

Forth altogether lacks any operation management system. Therefore, there is an opportunity to provide both an object and a data typing system with the same set of routines. This approach can be described alternately as a Forth data typing system and as an object-oriented Forth.

Even without a system for managing the matchups between objects and their associated operations, objects can be said to exist. The Forth programmer is the "management system" that ensures proper matchups between operations and objects. Without an operation management system to get in the way, inheritance is simple: the Forth programmer inserts the name of the desired operation, without any ensuing complaints from the compiler.

A major component ... already exists in the form of vocabularies.

Forth's vocabulary provisions can become the basis for a Forth operation management system. For example, a reference to a variable can automatically establish an `INT` (integer) vocabulary context, helping select operators appropriate for integer objects. This results in a natural classification of objects according to their data types: One `INT` vocabulary is the repository for all operations of type `INT`, and all the objects that set `INT` as the context vocabulary are assumed to be integer objects. Integer objects might be constants, variables, or even the elements of a more complex object.

Since a major component of such an operation management system already exists in the form of vocabularies, the OOF presented here is dramatically brief. Two new words are needed. They are special versions of `CREATE` and `DOES>` that can be used to declare objects.

The use of vocabularies is a substantial departure from the customary OOL model. One difference is the lack of formal messages associated with objects. Upon mention of an object, the Forth compiler is affected because the search order changes. It is difficult to say that this scheme uses messages, since it merely involves a vocabulary context switch that affects compilation. If such context switches can be thought of as messages, then perhaps this can be thought of as a message-based system.

Another important difference from other OOLs is the lack of support for late binding. Late binding seems to require a prefix syntax, and I have been loathe to accept prefix syntax for late binding while postfix operators remain a Forth standard. To provide this form of late binding, an operator table is typically used for each type of object. Each operation table is indexed by a common set of prefix "messages" (see RAY87).

Criticism of OOP

My criticism of OOP centers around two assertions: (1) the OOP model is too limited without multiple inheritance, and multiple inheritance requires great patience and care; and (2) the benefits claimed by OOLs can often be obtained without OOP.

By demanding that real-world phenomena be represented with a single object hierarchy, OOP imposes serious constraints. Which object should be subordinate to another? Often, class hierarchies are created solely to make it possible to inherit certain operations. However, this is a reflection of the programmer's problem space rather than the real-world phenomena the object is helping to represent. Multiple inheritance helps address this problem, but at a cost. It reduces the attractiveness of OOP as a programming philosophy and undisciplines what is otherwise a powerful organizational technique.

In the first installment of this series, I tried to dispel misguided enthusiasm over OOP by saying that the recent preoccupation with object-oriented programming resulted from the ability to reuse operations through inheritance—an ability many programmers lost with strongly typed languages. By skipping data type enforcement, Forth never lost the capability and so never needed to regain it.

The ability to reuse operations is not the only advantage of OOP that can be achieved through means other than OOP.

Among the benefits of OOP are the creation of more portable code; the ability to overload operator names to avoid a horde of new operators such as `W@`, `LONG@`, `FP@`, etc.; and the elimination of operator-object mismatches.

However, more portable code is best solved through abstraction of the host computer (as shown in the previous installment). The code introduced in this article will at least partially provide the third benefit. Overloaded operator names can be provided without OOP, as described in the following section.

Syntactical Means of Operator Symbol Reduction

Operator overloading is one of the desirable advantages of OOP. Let's examine a table that shows the candidates for name overloading:

	Fetch	Store
Char	`C@`	`C!`
Int	`@`	`!`
Float	`F@`	`F!`
Double	`D@`	`D!`

When used this way, eight operators are needed for two operations, and sixteen for four, proceeding exponentially. However, four data type identifiers and two operator identifiers used in combination can permit the same reduction in operator names that messages provide for OOLs:

Fetch	Store
`CHAR GET`	`CHAR PUT`
`INT GET`	`INT PUT`
`FLOAT GET`	`FLOAT PUT`
`DBL GET`	`DBL PUT`

Each new operator adds only one new identifier, and each new data type adds only one new identifier, proceeding arithmetically. In this case, slightly more verbosity imposes little compile-time and no run-time overhead, since `CHAR`, `INT`, `DBL`, and `FLOAT` can simply be vocabularies.

Obstacles to OOF

Before now, object-oriented Forths (OOFs) have had two drawbacks to detract from their appeal: The stretch required to make Forth an OOL is often too great, resulting in a language too unlike Forth; and Forth's parameter stack cripples our effort to create an OOF.

Object-oriented programming relies on a typed language. By storing floating-point numbers, truth flags, bit maps, addresses, integers, and characters all on the parameter stack, Forth seriously discourages any data typing systems. Accordingly, object-oriented Forths have to steer clear of "normal" Forth. Separate stacks for each data type, or a data typing stack must be added.

The problem can be illustrated with a fragment taken from the word `LATEST`:

```
CURRENT   @   @
```

`CURRENT` should be part of a class of objects exhibiting a pointer property. Call it the "pointer class" to indicate that an address is the only allowable interpretation for the value stored in `CURRENT`. After the value inside `CURRENT` is fetched by a "pointer" fetch operator, the stack contains an object exhibiting an "address" property. After the final fetch by an "address" fetch operator, the stack contains a value with a "name-field-address" property.

I Object!

Operator overloading reduces the number of operators that have to be remembered. When combined with an object typing system, the correct operator can be selected without any errors. This is good. However, the approaches we often see are often not completely faithful to this goal.

Suppose the average of two numbers must be determined. Typically, OOFs use the following prefix stack operations and postfix arithmetic operations:

```
GET   A
GET   B
+
2 /
PUT   C
```

Although OOFs may select the correct fetch operation for A and B and the correct store operation for C, they typically do not help select the appropriate plus and divide operators for the values on the stack. This is not so good.

Unless Forth changes, postfix operations will remain. Unless a replacement can be found for the prefix message syntax, OOFs will remain prefix.

Furthermore, more comprehensive operation management systems are required. By making all Forth operators prefix, we'll create a consistent prefix syntax, but we'll lose the advantages of postfix syntax.

A Better Tradeoff

If some inconsistency must be tolerated, let it involve something other than prefix syntax for some operators and postfix for others. Occasionally, programmer intervention will be necessary when using the operation management system I am suggesting. Of the possible inconsistencies, I found this much more acceptable. Here is why:

Explicit references to the vocabularies `CHAR`, `INT`, `DBL`, and `FLOAT` are mostly unnecessary when objects are manipulated without intervening stack operations. On the other hand, this inconsistency can lead to a less strictly enforced object typing system. Unlike most others, this operation system lets us have our object typing system and override it too. This way, the spirit of Forth, as exemplified by complete programmer control over the language, is preserved.

For example, a value can be fetched from a `CHAR` variable and then stored as a cell integer at the `PAD`'s location, using:

```
PAD   INT   !
```

No restrictive object or data typing system gets in our way: We do not have to request the use of a store operation from the set

belonging to the INT data type, nor do we have to convert or declare PAD as an INT type of object for the moment. We do have to help select the correct store operation by stating the INT vocabulary, however.

Most of the time, we retain the option of automatic operation selection. Yet where it is likely to be unreliable, manually selected operations can be dictated. Assuming that CURRENT sets the correct vocabulary to compile the correct GET operator, we could be assured that the object-oriented

```
CURRENT   GET   ADDRESS   GET
```

acts the same as standard CURRENT @ @. In this case, the programmer must be aware that ADDRESS is the appropriate type vocabulary for the value that resulted from CURRENT GET.

By ensuring that GET and PUT (and by extension +, /, -, *, etc.) do not have definitions in the Forth vocabulary, we can detect certain errors at compile time. For example, if in the preceding example CURRENT does not set a context vocabulary that includes a GET operation, then "Not found" or "Huh?" errors would be displayed.

Such an operation management system can also support a form of inheritance, through hierarchically structured search paths (as opposed to hierarchical object classes). For example,

```
NOT ONLY  PTR
BUT ALSO ADDRESS
```

helps compile the GET and PUT operators when both are located in either the PTR (pointer) vocabulary, or in the ADDRESS vocabulary. Although CURRENT could establish PTR as the first vocabulary to be searched, operations from the ADDRESS vocabulary would be available also.

To describe this action in OOP terms, ADDRESS becomes a superclass, or base class, for PTR. For an OOL, however, this relationship would be a permanent condition, not just a temporary one. With the approach I am suggesting, PTR can be the superclass at one point, and ADDRESS the superclass at another point (a form of temporal, multiple inheritance).

One object hierarchy could be:

```
ADDRESS
 PTR
 CELL-ADR
 CHAR-ADR
 DBL-ADR
 CFA
```

Within such an object hierarchy, D@ (or the equivalent) would be placed in the DBL-ADR vocabulary, C@ (or the equivalent) in the CHAR-ADR vocabulary, and so on. For ease of reuse, @ could be placed in the ADDRESS vocabulary, so that the classes PTR, CFA, and CELL-ADR could "inherit" it, rather than having it occur three times in these three different vocabularies.

To implement overloaded operators, you could redefine D@ as @ in the DBL vocabulary; see Figure One. Note that DP must now execute during compilation to set the DBL vocabulary in order to help compile the correct version of @ for itself.

Another possibility is to make a version of GET specifically for the PTR vocabulary. The new GET could reflect some unique properties about PTR objects, as in Figure Two. By switching to the next appropriate vocabulary, the PTR version of GET helps the operator management system keep track of the data type of the item left on top of the stack. This is comparable to Forth's compiler extensibility through the use of IMMEDIATE words. IMMEDIATE words permit compiler actions to be distributed throughout many different routines. Likewise, by distributing portions of the object typing system throughout the operators themselves, operation management is implemented as an extensible collection of functions.

In this way, the reach of the operator management system can be extended. For example, operator selection can be continued for two operations beyond CURRENT, not just one. This permits CURRENT @ @ to be changed into the following object-oriented code:

```
CURRENT   GET   GET
```

The two GETs shown are really two different object-specific operations, selected as a result of the vocabulary that had been previously associated with CURRENT.

The reach of the object-typing system can be extended as far as it is useful to do so, even to extremes. Suppose CURRENT sets the context vocabulary to INDIRECT-NFA-POINTER. Then suppose that the version of GET compiled after CURRENT sets the context vocabulary to NFA-POINTER. The final GET would be the version that had been placed in the NFA-POINTER vocabulary. Suppose it fetches an NFA and sets the NFA vocabulary. Then CURRENT GET GET would finally leave the NFA vocabulary set, so that associated operations such as ID., >LFA, and >PFA could be enabled for compilation or execution.

Besides extending the reach of the object-typing system, this would also resolve collisions of operator names such as >PFA. Two separate versions of >PFA could be included, one in the CFA vocabulary and the other in the NFA vocabulary.

Furthermore, such an object-typing system supports the postfix selection of all operators, including arithmetic operators. For example, the plus operator in the following sequence is the one appropriate for object B, since there is no intervening vocabulary switch after B GET:

```
A   GET
B   GET
+
2   /
C   PUT
```

A prerequisite is the overloading of all arithmetic operators through type vocabularies, such as INT, DBL, and CHAR. Note also that no warning would be issued if A and B were DBL (double) variables while C was an INT variable, even though the double result would only be half-consumed. (Extensions for run-time type checking will be able to detect this type of error.)

The same mixed operator should not appear in two vocabularies. It should only appear in the vocabulary corresponding to the data type that is expected to be on the top of the stack. There will be times when the INT vocabulary is not automatically set before the appearance of a mixed-operator such as M*/. However, the "take-charge" solution is to simply precede M*/ with INT. To suitably extend the operation management system (refer to the way S>D is modified in Figure Three), M*/ should set the double vocabulary.

We have already given considerable attention to an operator like M*/, namely GET. A wide variety of GET operations were suggested to help account for the type of the datum each leaves on the stack.

Large-Scale Development

By making certain declared elements less available (private) and others more available (public), some modern languages assist with large-scale program development. One practice involves assigning to a programmer the "private" operations for

an object, such as its declaration routine, as well as any object-specific operations, such as fetch and store (GET and PUT).

By considering particular object-specific vocabularies as their own private domain, members of a multiple-programmer project can become part of a productive, well-coordinated team. While every team member should be able to execute the routines in an object-specific vocabulary, making alterations to those routines could be restricted (possibly through changes to FORGET and DEFINITIONS).

For example, by locating a generic sort routine in the FORTH vocabulary and by allowing a gang of programmers to develop object-comparison operations, a large, parallel effort can be undertaken effectively. The integrity of the overall program should remain much more stable, even while simultaneous development and testing efforts are underway.

Libraries

Another way to view object-specific vocabularies is as libraries. The preceding generic sort suggestion helps show that "library" vocabularies can play a substantial role. Existing libraries (or suites of routines) are often much less conservative with memory, because they often include many peripheral routines that round out the Forth dictionary. Most of these basic operations should not be part of one library (or vocabulary). The separation of operations into their proper "object" libraries could alleviate this problem. When libraries are required to be more modular, as will be the case for "object" vocabularies, we will be able to load only the required routines, yet still be assured that those routines have no unknown external dependencies.

However, library development such as I am suggesting will require greater care to ensure that "object" libraries can be used in a standalone fashion. This may lead to greater pressure to add more functions to the "standard" Forth kernel. But it will also help reduce the number of operations that permanently reside in the FORTH vocabulary. Because many of them can be associated with a particular data type, they could then be pushed off into an appropriate library.

Another difficulty library developers will face is the need to avoid CPU-specific functionality. However, these difficulties are not insurmountable, as the next section will show.

Parallel Vocabularies

Consider the flexibility that would result if all cell-sensitive operations (which regrettably include IF, LITERAL, and others) were either left out of the Forth vocabulary or were included there as deferred words. Then a single vendor's Forth could engage a 32-bit cell width or even a 64-bit cell width (assuming such a library was available).

To illustrate, suppose CELL is a deferred word that is habitually associated with "cell" vocabularies, such as 16-BIT and 32-BIT. The vocabularies accessed through CELL can be called parallel vocabularies. Naturally, the parallel operations in parallel vocabularies are overloaded.

With an object-typing system like the one under consideration, not only could the cell width be changed for the compilation of one program, but it could even be changed for any of the individual routines within a program.

For such an OOF, the dynamic configuration of data types is not difficult to imagine. Each instance object would "know" what kind of object it is, and accordingly set the correct vocabulary for itself. This can happen even despite the reassignment of CELL any time after the object is declared. When CELL is deferred differently than a specific object, that object still can be associated to the currently inactive parallel vocabulary. Objects declared through the new declaration provisions maintain the ability to reinstate the inactive parallel vocabulary. Once they have done so, requesting the CELL vocabulary resets the context vocabulary back to the parallel vocabulary intended to be active, and away from the supposedly inactive one that was appropriate for the object that set it.

The FORTH vocabulary will require one of the parallel vocabularies to be its own native cell vocabulary. Because other parallel vocabularies can override the default at any time, more care has to go into the definition of cell-sensitive Forth words. For example, IF will need to be defined with a conversion operation that is dependent on the currently active cell vocabulary. The conversion operation is needed to ensure that the bit-width of the flag left on the stack is adjusted to match that of the kernel's idea of the width of a cell. Otherwise, IF may not entirely consume the flag, resulting in incorrect Forth interpreter and compiler operation.

Conversion operations that may have to be automatically selected are 32>16 and 16>32. If the current interpretation environment is regarding the width of a cell as 32 bits, yet the kernel cell width had originally been 16 bits, then the 32>16 operation is necessary to allow the native 16-bit conditionals to consume their context-dependent stack parameters at run time. By incorporating a deferrable word such as >ORIGINAL-WIDTH, late binding can be employed to help select the correct conversion operation (see Figure Four).

To establish a 32-bit-cell vocabulary, a word such as CELLWIDTH can be created in the 32-BIT library, as a parallel operation of those in all the other parallel vocabularies. It should modify certain kernel words in the appropriate way (see Figure Five). This way, 32-BIT CELLWIDTH would make the 32-BIT vocabulary the currently active one. (Note that this code will not work, because IS works properly only during interpretation. I deliberately chose to express the idea this way since it is clearer than it would otherwise be.)

In a similar way, other libraries could be made passive or inactive to support multiple FLOAT data types, perhaps making available 16-bit, 32-bit, 48-bit, and 64-bit floating-point numbers under one Forth. For a particular application, any unused parallel vocabularies could be forgotten in order to recover memory.

Such a treatment of libraries could give birth to library vendors. This would help eradicate the prevailing "one-bit-width-size-fits-all" notion. Since components could be purchased from several different vendors, Forth programming environments would be upgradable in ways that previously would have required abandoning one Forth implementation for another.

The existence of highly modular Forth libraries could accelerate Forth's acceptance by the general programming community

Implementation

To retain Forth's existing syntax for data declarators, the new provisions are modeled directly upon CREATE and DOES>. These new provisions are OBJ-CREATE and OBJ-DOES>. An example of their use is shown in Figure Six.

Suppose DOUBLE is used to create DP. DP must be made an immediate word in order to perform two actions at compile

time: compile a reference to a component part of itself, and set the appropriate context vocabulary. These two actions may be bound to DP through two different code fields:

| DP | cf1 | vocab | cf2 | <value> |

The first code field will be responsible for setting the vocabulary and compiling or executing the second code field, depending on the system state. The second code field will be responsible for the run-time action of a particular object (such as D@ for a double constant).

Let's concentrate on the immediate or compile-time actions associated with the first code field. Since the first code field's actions must be shared by all (typed) objects, it can be made general (see Figure Seven). To conform with our desired syntax, GENERAL-DECLARATOR can be abandoned in favor of OBJ-CREATE. Like GENERAL-DECLARATOR, it can also bind the object-general behavior to the resultant object. So, whereas CREATE builds

| <name> | std cf |

by using the code in Figure Eight as its definition, OBJ-CREATE can go ahead and build

| <name> | object-general cf | vocab |

For now, at least, we have the correct compile-time action. Now let's implement the object-specific behavior which occurs at run time. That's easy. In Figure Nine-a, for example, we make a declarator for double constants.

Now we have the correct run-time action. However, the DOES> which follows OBJ-CREATE patches the one and only code field of DP, so that instance objects only execute D@. The object-general code within OBJ-CREATE is ignored. To fix this, OBJ-DOES> is used as a replacement for DOES>, and is defined as in Figure Nine-b (line numbers have been added for explanatory purposes).

This special version of DOES> retrieves the first code field value created by OBJ-CREATE to the stack (lines 1 and 2), then allows the first code field to be overwritten by DOES> (line 3). Then, line 4 of OBJ-DOES> retrieves the value for the second code field (now incorrectly stored as the first code field), and relocates it to the end of the current instance object (line 5). Finally, it restores the first code field, using the previously stacked value (line 6). When the dust settles, the instance object looks like that shown in Figure Ten.

We have a different problem now. The second code field needs to appear before the four-byte allotment. One fix is to have OBJ-CREATE allocate space for the second code field. Add ADR/, ALLOT to OBJ-CREATE, directly preceding DOES>. Since the initial instance object left by OBJ-CREATE has an invariant layout, OBJ-DOES> can anticipate that the second code field location is one cell past the first code field (see Figure Eleven).

Using the new object declaration provisions, the declarator DCONSTANT can be created (Figure Twelve-a) that builds instance objects in the corrected format (Figure Twelve-b).

For a very broad discussion of dual code fields, see "Dual-CFA Definitions" (ELO86). That article concentrated on executable colon definitions rather than data definitions, so the illustrations differ substantially.

Final Remarks

My original goal for this series of articles has been eclipsed by this installment's preoccupation with OOF. Originally, the goal was to identify the data design strategies that would promote the portability and reusability of objects. I continue to believe that studies of data structures that focus on portability and efficiency issues will prove fruitful, and this has not been an exception. I'll summarize the preferred design style and practices identified:

1. Share layout properties among data ojects to enhance reuse of operations (Part One).
2. Use data abstraction to hide "private" components of data objects from unnec essary, direct manipulation and to ease development (Parts Two and Four).
3. Use host abstraction to enhance source portability (Part Three).
4. Use clear declaration and consistent op eration syntaxes, even though they may be more verbose (Parts Three and Four).

Notably lacking in this new OOF is support for late binding. Another desirable enhancement is an implementation of "structures" that supports object-typing at the structure level as well as the level of components within structures.

Because it can already add features to Forth that are highly touted for other modern languages, this germinal OOF is substantial. Rather than completely overhauling Forth, I have suggested ways to use Forth's vocabulary provisions, and a couple of new data declarators, to realize the missing functionality. I am happy to report that these additions can preserve Forth's promise of complete programmer control of the language.

References

ELO86: Elola, Mike. "Dual CFA Definitions," *Forth Dimensions* IX/1.

ELO88: Elola, Mike. "Designing Data Structures, Part One", *Forth Dimensions* X/2.

RAY87: Rayburn, Terry. "Methods> Object-Oriented Extensions Redux," *1987 FORML Proceedings*, Forth Interest Group.

Mike Elola is a published Forth programmer and a full-time writer at Apple Computer. Over the years, Mike feels, Forth has tricked him into believing he is a computer scientist.

```
DBL  DEFINITIONS
: @    D@  ;

FORTH DEFINITIONS
DOUBLE  DP
: HERE  DP @  ;
```

Figure One. Redefinition of D@ and @.

```
: GET       ( adr -- )  ( for pointers )
  @
  [COMPILE]  ADDRESS
             ( set the address vocabulary)
  ;  IMMEDIATE
```

Figure Two. A version of GET specifically for the PTR vocabulary.

```
INT  DEFINITIONS
:  S>D  (  n  -- d )
           ( compile-timeonly)
      COMPILE  S>D
      [COMPILE]  DBL  ;  IMMEDIATE
FORTH DEFINITIONS
```

Figure Three. Confine mixed operators to the appropriate vocabulary.

```
DEFER  >ORIGINAL-WIDTH
:  IF
   COMPILE  >ORIGINAL-WIDTH
   ... ;  IMMEDIATE
```

Figure Four. A deferred word permits use of late binding.

```
32-BIT  DEFINITIONS
:  CELLWIDTH
   >ORIGINAL-WIDTH IS 32>16
( adjustcompiler words )
   LITERAL  IS  CELL-LITERAL
( adjust INTERPRET )
   CELL  IS  16-BIT
( enable 16-bit cells )
   BYTES/CELL  IS  2  ;
( adjust host-specific values )
```

Figure Five. Modifying kernel words to use a 32-bit-cell vocabulary

```
:  DOUBLE
   [COMPILE]  DBL
( set context vocablary )
   OBJ-CREATE  ADR/,  2 *  ALLOT
   OBJ-DOES>  ;
```

Figure Six. Example use of object-defining words.

```
:  GENERAL-DECLARATOR
( template use only)
   CREATE
   CONTEXT  @  ,
(  compile class-specific code field )
      DOES>
( pointer-to-vocabulary -- )
      DUP  @  CONTEXT  !
      ADR/,  +   ( cfa2 -- )
      STATE  @
      IF  ,  ELSE  EXECUTE  THEN  ;
```

Figure Seven. Code for the generalized (shared) code field.

```
:  OBJ-CREATE      ( -- )
   CREATE
   IMMEDIATE
   CONTEXT  @  ,  ( -- )
   DOES>
      DUP  @  CONTEXT  !
   ADR/,  + ( cfa2 -- )
   STATE  @
      IF  ,  ELSE  EXECUTE  THEN  ;
```

Figure Eight. OBJ-CREATE binds an object-general behavior to the objects it creates.

```
:  DCONSTANT
   OBJ-CREATE   ADR/,  2  *  ALLOT
   DOES>  D@  ;
```

Figure Nine-a. Following OBJ-CREATE with DOES> causes the object-general code to be ignored.

```
0)   :  OBJ-DOES>          (  -- )
1)      LATEST  >CFA       ( cfa -- )
2)      DUP  @  SWAP       ( ObjectGener-
alCfaValue cfa -- )
3)      [COMPILE]  DOES>
4)      DUP  @
       ( ObjectSpecificCfaValue -- )
5)      ,
6)      !     (  -- )  ;  IMMEDIATE
```

Figure Nine-b. Initial (problematic) definition of OBJ-DOES>.

| <name> | obj-general cf1 | voc-ptr | 4-bytes | class-specific cf2 |

Figure Ten. Instance object structure that results from the problematic OBJ-DOES>.

```
:  OBJ-DOES>       (  -- )
   LATEST  >CFA    ( cfa -- )
   DUP  @  SWAP    ( ObjectGeneralCfaValue  cfa -- )
   [COMPILE]  DOES>
   DUP  @          ( cfa ObjectSpecificCfaValue -- )
   OVER  ADR/,  +  !
   !  ( -- )  ;  IMMEDIATE
```

Figure Eleven. Final definition of OBJ-DOES>.

```
:  DCONSTANT
   [COMPILE]  DBL
   OBJ-CREATE ( <name>:<cf1>:<voc-ptr>:<tbs-cf2>:)
   D,
   OBJ-DOES>  D@  ;
```

Figure Twelve-a. Defining an object declarator.

| <name> | obj-general cf1 | voc | class-specific cf2 | 4-bytes |

Figure Twelve-b. The correct format of resulting instance objects.

1988 Forth National Convention

REAL-TIME PROGRAMMING

Anaheim, California— The Forth Interest Group held its annual convention here in the Grand Hotel on November 18–19, the first time the event has taken place outside Northern California. Chairman Martin Tracy convened an outstanding program based on the theme, "Real-Time Programming." The caliber of the presentations made one wish the proceedings were to be published; however, all of the speakers are contributors to the Forth community, and many are published in the related literature. Limited space prevents our reporting the contents of most speeches here; highlights follow, and we hope to present many of the speakers' ideas as future articles.

Attendance at this year's convention exceeded that of preceding years, and the exhibit room was notably busy during all breaks in the conference room. Crowds were especially thick at the New Micros booth, which sold out of their hardware offering early; at the GEnie booth, which was promoting the on-line Forth RoundTable; and at the Forth Interest Group booth, which featured a selection of technical literature not seen before by many attendees. Other vendors of Forth hardware and software systems also enjoyed brisk activity throughout the event.

Ray Duncan, proprietor of Laboratory Microsystems, Inc., set a thoughtful and inspiring tone for the convention with his keynote speech. He pointed out that this year's venue proved that the Forth Interest Group recognizes Forth users outside the San Francisco Bay area, and that the theme shifted our focus back to Forth's true forté: real-time programming.

Forth, Ray reminded us, was invented by a real-time programmer, not by an academic, publish-or-perish writer. When it splintered into various dialects early on, it became difficult to teach, to write textbooks about, and to use the public-domain code that was published in *Dr. Dobb's Journal* and other places. But there has been a recent shift in public attitudes about Forth: now that its proponents aren't claiming every popular programming concept as "perfect for Forth," developers are beginning to admit its genuine strengths. Ray believes that Forth and real-time applications really are well suited to one another. Forth is a powerful probe for new and unstable or untested environments. For example, it gave him a 6–8 month lead time over other authors when he was writing his book about OS/2 for Microsoft Press.

J.D. Hildebrand, editor of the new *Embedded Systems Programming*, delighted the audience with his "Ten Myths of Real-Time Programming":

1. It's a small, specialized niche.
2. "Real time" means as fast as possible.
3. RISC is a panacea (think about interrupts).
4. Software design is subordinate to hardware design.
5. The system-development life cycle is: analysis, design, coding, debug, integration, self-congratulation. (Existing tools facilitate this flow, not the real-world process.)
6. Ada.
7. Real-time programmers use (only) assembly language.
8. Programming is programming, there are no special issues (the Computer Science attitude).
9. We have all the tools we need.
10. Real-time programmers must be self-taught (reinventing circular queues and concurrency every month).

Air Force Major Steven LeClaire was called the rennaissance man of expert systems by *Business Week*. His presentation showed how current efforts in that field could lead to a kind of cognitive companion for managers and researchers. Such a system would offer supplemental analyses and perspectives of a particular data set or environment. This will be particularly valuable in complex fields, like making esoteric medical diagnoses and in areas like manufacturing where scientific rules for decision making have not yet been formulated. His first system was based on Jack Park's Expert5, a Forth product.

The convention event which evoked the most spontaneous enthusiasm was the contest for "the world's fastest programmer." The object of this contest was to be the first to program a "mystery gizmo" using the serial port of a host computer. Some representatives of languages other than Forth were to join the fray, but did not appear as planned. A smorgasbord of systems were used by the participants, some of which were:

Commodore	64FORTH
Zenith	F83
Macintosh II	MacForth+
Sharp	F83
Compaq	PC/Forth
Otrona	Z80 Forth
Samsung	polyFORTH
Grid/New Micros	MaxForth
Amiga	JForth

Contest designer Martin Tracy un-

veiled a working version of the gizmo in front of an enthusiastic crowd of spectators and fourteen programmers who vied for the $1000 prize. The device consisted of a row of LEDs attached to one end of a hacksaw blade. The blade was mounted vertically to a base unit. A solenoid at the base caused the blade to sweep back and forth in a continuous arc, while the LEDs at the top end (capitalizing on the retinal retention of the human eye) displayed a scrolling message in the air: "The rain in Spain falls mainly on the plain."

A gizmo was passed to each contestant. Martin called out, "Start your gizmos!" and the hacking began. A few contestants had frankly puzzled expressions, but most dived in with—apparently—some idea of where they thought they should begin. About half started working on the oscillation (figuring that coming up with a few characters wouldn't be so tough), while the others started sketching character sets on paper (what could be so hard about driving

CHARLES MOORE'S FIRESIDE CHAT

REPORTED BY DENNIS RUFFER

This report cannot be considered a transcript of Chuck's talk, since I am merely attempting to reconstruct it from notes I took during his talk. I do hope, however, that this conveys the intent of his words and that, by reading this report, the rest of you can get a feeling for the current state of our industry.

It is not quite clear why this is called the fireside chat, since they have never had a fire. Chuck wore his Australian hat, since he did not want to look too much like Einstein as portrayed in the advertisements for this convention. He got it at last year's Australian Forth Symposium. In his travels, he finds it very interesting that there is a community of interest in Forth around the world.

On November 19, 1968 the first Forth computer said OK. It was an IBM 1130 computer located in a warehouse in Amsterdam, NY. It was intended to be a pattern designer for carpets, but since its black and white display was totally inappropriate for the job, it was abandoned within a year. It is remarkable how similar that early Forth was to the Forth we have today.

Chuck really enjoyed the "Real-Time Programmers Contest" we had this year. As he walked around the room, he noticed how everyone was using very interactive, modular designs. He wondered if the ones who used multiple windowed Forth found that those tools got in the way, or if the ones who did not have those tools missed them. He suggests that scrolling the parameters for this project would work very nicely with his three-key keyboard. He was asked if he would use the display techniques from the contest in his next computer but, laughing, he said, "No, it has a moving part and no color." Although he did not enter this year's contest, using the excuse that he had no computer to show off, he will try it next year.

Although he designed the Novix with a 16-bit "B" port intended to be a bus for interconnecting multiple CPUs, he has never seen the technique used. So far, he has not seen anything he could not do with just one Novix. It is not clear what computers are, or what software is. All you need is a couple of latches and shift registers. Minimalist to the core. He is working on another chip, but he is having to learn how to use workstations to enter the design.

FORTH, Inc. is a neat company, and the people are getting down to real programs. However, they keep reinventing the wheel, doing the same applications over again. CAD existed many years ago, but few remember how it was done. The 300-foot telescope that crumbled in Green Bay was written in Forth.

Forth should be under 4K bytes, but he hasn't found a lower limit. You don't need a neural net or adaptive program to balance a broom. All you need is a clever programmer, but they are in short supply. He urges the neural net people to show him something significant they have accomplished. Just because you cannot see the inner workings of a system does not mean that it is chaotic, merely non-linear.

The ANSI standard effort is using a ridiculous amount of energy. It behooves us all to see that it is a good standard. It is not easy to standardize the groundwork of Forth. He considers the "little users" the mainstay of Forth, and they are not well represented. Those who it affects (authors, etc.), must make sure they are not disappointed. *"Get involved!"*

The future of Forth looks very promising. He just got done with the science fiction book, *The Regiment*, about a war fought by mercenaries. They don't really care if they are killed, the goal is to fight well. We tend to get too serious. The reason Forth is spreading is that it is fun. He recently saw a book about Forth in Russian.... Novix is still in business, but they are for sale and still have chips in stock.

At this point, Chuck was done with his rambling notes and he opened the discussion to questions from the floor. Following are some of his thoughts that the questions brought out.

The high cost of RAM is only a temporary problem, and soon they will again tend toward zero. However, people are not happy with the gigabyte operating systems, and they often forget the other extraneous cost of memory. Memory will always be the highest-cost item in a system. Chuck believes in distributed systems, where each piece only takes one or two programmers. Every large project is conceived by management. His concept is to have all the programmers in the same room. You will lose some effort in socializing, but you will also get the most work done. He believes the documentation should be written in parallel with the programming and use the same interactive techniques.

What is Forth, a tool to use a machine, or to share algorithms? Chuck suggested to the ANS committee that the standard be a publication standard, which might give us a more effective way to share programs, but people could not just type in the programs. He noted that all the contestants were doing the same thing, each in his own way. He invented something that he could just hack at his computer. There is nothing that Forth cannot do, and it is superior in every application.

It is impossible to define Forth, but it is that which we have in common. He no longer wants his computer to understand his voice, since he found that he has nothing he wants to say to it.

a solenoid?). But the gizmo was not to be easily conquered, thanks to a couple of twists in the design; about an hour into the contest, *everyone* had puzzled expressions except the spectators, who were laughing and rushing from station to station looking for a leader.

It turned out that the rate of oscillation had an important effect; it wasn't all that easy to determine, and the hacksaw blade tended to be spastic, uncooperative. As for the character set, well, you couldn't just pull it out of ROM; it seemed there should be just one more LED to work with... Finally, one hour and 22 minutes after the contest began, the team of Phil Burk and Mike Haas got the oscillation going. The rapid tick-tick-tick of their device drew the spectators rapidly, who clustered around to chant out the words as they appeared, "The rain in Spain falls mainly on the plain." As they anarchically acknowledged the winner, Martin announced, "By Jove, I think they've got it!" The winners were from Delta Research in San Rafael, California, and used their company's JForth for the Amiga. They were presented with a check from the Forth Interest Group for $1000, made possible by contributions from several Forth vendors and convention sponsors.

Parallel sessions were held on Friday by Harris Semiconductor in the form of tutorials featuring the Harris RTX 2000, billed as the Real-Time Express Microcontroller. More than 120 attended the morning seminar, while 35 more enjoyed the afternoon session. The presentations consisted of multi-segment slide shows, questions and answers, and technical and marketing lectures by nearly a dozen Harris staff members.

Harris contrasts their RTX 2000—which builds on earlier Forth hardware—against microcontrollers, which don't meet the performance demands of some applications; CISC and RISC machines, with which one loses integration; and semi-custom controllers, which are usually expensive and complex. The RTX builds hardware solutions based on standard, off-the-shelf "cells." It executes Forth directly, although Prolog and C compilers are expected in 1989. (Contact Harris Semiconductor for technical details, which looked impressive. A 32-bit version is due in "about a year.")

The traditional concluding event at the annual Forth National Convention is a banquet on Saturday evening. Forth Interest Group President Robert Reiling (co-founder West Coast Computer Faire, Homebrew Computer Club) was the Master of Ceremonies. He announced that his term as a FIG officer is drawing to a close, and noted that the organization's growth was most recently marked by reported efforts on the part of several Forth groups in Bulgaria to form a FIG Chapter. Then FIG Vice-President John D. Hall announced that this year's award for outstanding service to the community goes to Dennis Ruffer. Dennis is a first-term Board member and has spent long hours organizing the Forth RoundTable on the GEnie telecommunications service. Fellow sysops on GEnie were later heard (on-line) to approve the choice, noting that the electronic venue for Forth has yet to be fully appreciated.

The final agenda item of the evening was a memorable after-dinner talk by Jef Raskin, served up with a mild, chandelier-swaying earthquake. Jef was instrumental in creating the original Macintosh project at Apple Computer, and was a member of ANSI committee X3J2 to standardize BASIC. More recently, he invented the Canon Cat, a 68000-based machine that takes a big step toward seamless integration of common business functions and which will also execute Forth code.

Jef said he first used Forth 17 years ago, and subsequently programmed several machines with it. He said that Forth inspired some of the Macintosh's features, but especially the Cat. He also suggested that Forth's use of parentheses is exactly backwards: the programmer should write an essay that clearly describes a program's functions, putting the associated Forth code inside parentheses.

Raskin related some of the taken-for-granted ironies of interface design in our (non-computerized) environments. His tale of what happened when his on-demand hot water heater met a flow-restricting shower head had the audience shouting with laughter. Most computer interfaces, he said, are designed by amateurs. That is why we have printers with on-line switches, diskettes that have to be formatted by the user, and functions that can only be accessed via pull-down menus. And paperless communication has not yet arrived because telecommunication is too expensive and complex for most users. An upcoming book from about CD-ROM from Microsoft Press contains some of Jef's criticisms of Ted Nelson's hypertext concept. Jef suggested that anyone designing appliances for use by people should read *The Psychology of Everyday Things* and *The Psychology of Human-Computer Interaction.*

An editor's delight, Jef drove home the fact that word processing is a real-time experience for the user, and that response time at the keyboard has a definite impact on the human user. (Listen up, developers!) In the next five years, he expects the sales figures of machines designed for use in the home office to exceed the sales of any computer ever marketed. While this can only be viewed as a tremendous opportunity, it is partially offset by the fact that the technology required to manufacture many useful devices is simply not available in the United States.

Volume Nine Index

A comprehensive guide to all issues of *Forth Dimensions* published during the Volume IX membership year. If a cogent letter to the editor referred to a previous article, the original article is cited with the letter even if it was published in an earlier volume. See the Forth Interest Group's order form (center insert) for complete sets of back issues. Special thanks to Mike Elola of San Jose, California for compiling this index.

THE BEST OF GENIE

GARY SMITH - LITTLE ROCK, ARKANSAS

I have already devoted one column to efforts of the X3/J14 Technical Committee, their task of defining the future structure of a ANS Standard Forth, and input to that effort on GEnie. The working draft of X3/J14 is called BASISn, where n is the current level. I had commented briefly on some notable differences between BASIS4N and BASIS5—very briefly. A week later, Bob Berkey literally dissected BASIS5 by verse and line. The following message from Bob Berkey, and Greg Bailey's rebuttal, serve to reflect the nature of both better than anything I might say to induce your participation in this endeavor. Please read this exchange, make notes as you agree or disagree, and put them on GEnie's Forth RoundTable in Category 10, for all to benefit.

Category 10, Topic 2, Message 124
Sun Oct 09, 1988 R.BERKEY

Notes, comments, reactions, and opinions—after 24 hours with BASIS5:

p. 2 The Forth-77 and Forth-78 documents are no longer "pertinent."

p. 3 But parsing can be delimited by a string or a set of strings, and doesn't have to throw away leading delimiters. Suggest that the parsing concept here be called "word parsing." Parsing that only looks for a single trailing delimiter could be called "delimiter parsing."

p. 4 "Address, compilation" has become "compilation token" and "execution token." I like the use of "compilation token," because "compilation address" in 83 was a misnomer.

p. 5 Character. Character sets are now implementation defined, and not necessarily ASCII. Wow, this is a really biggie. It's not possible right now for a program to know what characters are in the character set. Numbers can be detected with `CONVERT` , but that's about it. It appears that output has new restrictions—can't print addresses or bit masks.

p. 42 Now that a char is implementation defined, `ASCII char` is a misnomer. If I am on an EBCDIC system, will `ASCII .` give me the value of an EBCDIC "." (as BASIS5 now states)? From my viewpoint, either the implementation-defined character set needs to go, or `ASCII` and `[ASCII]` should go.

All Propose that whenever a phrase from the Definitions of Terms is used that it be italicized. This would help in recognizing the intent of the language. The typesetting looks great. One exception, I'd like to see the Forth words without justification. Also, I find that Forth words without a trailing space are hard to syntactically distinguish from following punctuation. In fact, it might be more correct and useful to have the "word name" definition specify that a word name includes a trailing space—I hope to develop a proposal or a paper on this.

All The word "cell" is back! In Forth-79 it meant a 16-bit word. It took a vacation during Forth-83, and is now back as 16 or more bits. But, has this word been dormant long enough for its former meaning not to cause confusion? I doubt it.

p. 44 `BLOCK` is no longer in the required word set. Give me a 100 squared with a terminal and I will be able to have an ANS standard Forth! Since `FILE` words are also not required, a Standard Program no longer has a mass storage capability.

p. 16 Variables cannot be ticked (`'`). Don't know why.

p. 19 `PARSE` does not appear in the glossary.

p. 24 This is the page with stack parameter abbreviations. Things are starting to get hairy. The wrap-around number type (w) has been eliminated from the standard along with the arbitrary-bits number type (8b, 16b, 32b). A new number type, unspecified, has been added, but using the old label w. Oh no, *don't reuse the old "w" with a new definition! Find another letter!!!* What's wrong with "x"?

The confusion of the Forth-83 "w" and the BASIS5 "w" is pointless and I trust the committee will change it, but the deeper change involved is technically a dynamite issue. Any of the words in Forth-83 that had wrap-around numbers have, from the programmer's viewpoint, been radically altered.

Take + for example. In BASIS5 it has a stack of (n1 n2 -- n3). n has a range of -32767 to 32767 or larger. In Forth-83, any input to + (w1 w2 -- w3) produces a known output. The BASIS5 + allows the program to use fewer than half of these combinations—$7FE,002 compared to $1,000,000 if you are counting.

Oh no, I'm not believing this. `LOOP` is back to a sort of Forth-79 definition. This must be just a mistake. Mustn't it? Maybe not, what with consideration for overflow on a one's complement machine. Yet, now that I look further at the semantics under `DO`, I see that `w DUP DO ... LOOP` will execute at least 65,536 times. The two ideas seem contradictory, so no doubt more changes are coming here.

Here's another implication with the lost "w": Forth can no longer detect a carry. The idea is to use `D+` instead of + to add two numbers, and if the top of the stack is nonzero, a carry has occurred.

p. 24 Addr is now a data type rather than

a number type. This has yet-to-be-discovered implications. Char, bit-mask, and r (real) are also new data types. Specification is needed to know what input, arithmetic, logical, and output operators, etc., can be used with what data types.

p. 28 " ("quote"). New word compiles a string. It leaves addr and count. Good. CONVERT (with a new name) should also have addr and count. One proposal is to use a new name:
NUMBER? (adr count -- flag)

p. 3 (...) , ABORT" ... ", . (...) , and ." ... " will produce unexpected behavior. Only " (quote) has survived the capacity to handle a null string. I think this is a bad change. ABORT" ... " I use regularly, and the other three cases make Forth look flakey and/or unreliable.

p. 56 FIND. This word still refers to compilation address, which phrase is no longer in the definition of terms. Should have been changed along with ', ['], EXECUTE, and >BODY to execution token(?).

p. 42 Code and data are separated in the 1983 standard. , ("comma"), ALLOT, and HERE are properties of the program, that the system can borrow during compilation if desired. COMPILE can ignore HERE.

D2* and 2* are an interesting pair: D2* is a logical shift, and 2* is an arithmetic shift. Does this mean something? And, if you said to yourself "d-2-times" and "d-times" as you read that last sentence, they are now "d-2-star" and "d-star." First they told me it's "star," then they said in Forth-79 that it's "times." So for eight years I've tried to use "times" and now I am supposed to go back to "star." I assume FORTH, Inc. is involved in this, because they let Brodie publish those cutesy and memorable pictures in his book with the non-standard names. Hmmm, come to think of it, one of his more memorable pictures was "slash," and now I see that "divide" has not been changed, just "times." So this is yet a third name for */ and */MOD (?!?).

As I think about how I pronounce these words, I find there is a dichotomy between "star-slash" and "times-divide."

D2*	always "d-2-times"
2*	always "2-times"
UM*	always "u-m-times"
*	sometimes "star," sometimes "times"
*/	more often "star-slash" than "times-divide"
*/MOD	more often "star-slash-mod" than "times-divide-mod"
D2/	usually "d-2-slash"
UM/MOD	usually "u-m-slash-mod"

Is that a pattern? I guess I learned it one way early on and still don't have it switched over. It appears to me that there are two sets of pronunciations in general use. Personally, I'd rather not see the standard pronunciations changed. There is an alternative, however, which may seem radical, but also fits our world—*two* pronunciations in BASIS6, Brodie and 79-Standard. But a third and new pronunciation? I hope that doesn't last.

And another naming decision from Forth-79 has been overturned. The pictured numeric words are no longer "sharp" but "number-sign." As in <# "less-number-sign."

p. 69 New word: UNDO

p. 32–68 Nine new words in the controlled reference section. New words: <= <> >= 0<= 0<> 0>= U<= U>= U>

p. 34 New words: 2>R 2R> The last two concern me because they are not the same as R> R> and >R >R. The above potential bug needs to be covered in the rationale.

p. 62 NEGATE is the radix-complement. NOT is the radix-minus-one's complement. Some of us don't know what these mean, these concepts need to be in the definition of terms.

p. 66 SP@ deleted from the controlled word set. What was wrong with leaving this word controlled?

I sense that another organization, like the contributors to this board, could take up the cause of collecting commonly used names, to avoid needless duplication of names and the miscommunication that

results from one name coming to have two slightly different meanings. A good example of this is the long fetch with segment word used on the 80x86. Propose that `SP@` be the first word in a new Topic here.

I've been trying to figure out how to run a Forth-83 program using a BASIS5 system. In a worst case, one could implement the arithmetic/logical functions using 16 bytes to represent register bits. But probably it could be done with byte slices.

Implementation-requirement-removal proposals:

1. Leave the function of `BLK = 0` as implementation dependent. The older rule makes using block and buffer on a file system most inconvenient.

2. Remove the implementation requirement that : (colon) sets the compilation vocabulary. I believe this behavior is only for programmers with a line editor who would forget to manually switch out of the editor with its `I` word.

One pattern is clear from BASIS5, the committee is working to remove implementation requirements and specifications. While reducing the functional capacity of a Standard Program, the number of Forth implementations that could meet the standard has been increased. At the same time that running a Forth-83 Standard Program on a generic BASIS5-Forth Standard System has become functionally impractical, the committee is allowing that Forth-83 Standard Systems are maximally supported.

Category 10, Topic 2, Message 125
Sun Oct 09, 1988 G.BAILEY1

Howdy, Bob! Let me be the first to thank you very much for your brief (har) remarks on your first reading of BASIS5. If your second is as fruitful, we need to get your input into usable form. The next meeting is too soon to get proposals in that would meet the two-week rule, but if you have some time between now and then I would request that you get the less argumentative of them on paper and into Martin's hands anyway. We have not been two-weeking things introduced at the meetings unless really called for.

Proposals that clearly correct ambiguities or awkwardness of language without having any technical implications have been called "post" and are generally passed to the documentation committee without debate. Any such would be welcome at any time.

Issues of notation choice or implication that may have been overlooked could be simply pointed out, as you have already done, or specific solutions could be formulated (as you have also done). As a general comment, your posting is already very useful but if you have specific solutions that might be debatable, I would suggest formulating each as a proposal.

In both of these areas, it might be useful for you or anyone else who wishes to take the trouble to think about the editorial aspects of the document to coordinate with Ted Dickens who may be reached at 213-477-7287. He chairs the documentation committee and would likely be willing to compare notes on things his group may already be planning to propose.

As a matter of perspective on BASIS5:

Extensive editorial changes were necessary to beat our working document into the shape required by ANSI style. Rather than wrangle out endless detailed proposals, the documentation committee offered to produce a massively edited document that would presumably take less effort to complete than would the original. We accepted this offer, and BASIS4N—the immediate predecessor to BASIS5—resulted. It was adopted "warts and all" with the understanding that it had problems. You have been noticing some of the warts. The riskiest part of Ted's work was that it wasn't supposed to have technical impact. However, some of the changes undoubtedly had such impact. For example, I had proposed that we delete the requirement for null strings, but the support for this proposal did not represent a strong consensus, so the proposal has been committed. Thus, Ted's committee was *not* given authorization to delete this requirement and it should still be there. Likewise, the technical implications of the stack notation changes have not been subjected to deliberation by the committee as a whole except in a few cases when we have been working on specific words. No doubt there are unintended effects. Since there are known or suspected warts, proofreading for them is very useful. Please continue to do so.

I'd like to chat about a couple of se-

Statement of Ownership, Management and Circulation

1) Title of Publication: Forth Dimensions
   Publication Number: U.S.P.S. 002-191
2) Date of Filing: 9/19/88
3) Frequency of Issue: Bi-Monthly
   No. of issues published annually: 6
   Annual subscription price: $24/36
4) Location of known office of publication: 1330 S Bascom Ave., Suite D, San Jose, Santa Clara County, California 95128-4502
5) Location of the headquarters or general business offices of the publisher: Same as above
6) Publisher: Forth Interest Group, P.O. Box 8231, San Jose, California, 95155
   Editor: Marlin Ouverson, Same as above
   Business Manager: Georgiana F. Shepherd, Same as above
7) Owner: Forth Interest Group, Same as above
8) Known bondholders, mortgagees, and other security holders owning or holding
   1% or more total amount of bonds, mortages and other securities: none
9) The purpose, function and non-profit status of this organization and the exempt status for Federal Income Tax purposes have not changed during the preceding 12 months.
10) Extent and nature of circulation

	Average No. copies/issue during preceding 12 mos.	Actual No. Copies of single issue nearest to filing date
A. Total no. copies printed:	3583	3200
B. Paid/requested circulation:		
1. Sales:	0	0
2. Mail subscription:	2471	2132
C. Total paid/requested circulation:	2471	2132
D. Free distribution by mail, carrier or other means: samples, complimentary and other free copies:	59	50
E. Total distribution:	2530	2182
F. Copies not distributed:		
1. Office use, left over, unaccounted, spoiled after printing:	1053	1018
2. Return form news agents:	0	0
G. TOTAL:	3583	3200

11) I certify that the statements made by me above are correct and complete
/s/ Georgiana F. Shepherd

SIG FORTH

Join Now!

acm

the Association for Computing Machinery's Special Interest Group on Forth

Be Part of the Future of Forth...

SIGForth is the ACM special interest group whose members are the programmers, managers, scientists, engineers, and educators that are interested in applying Forth to solve hypothetical and real-world problems.

Become a **SIGForth** member. Get the **SIGForth** quarterly newsletter and receive a 25% discount on publications and conference registration with your membership. Stay up to date on the latest developments in Forth hardware and software. Your dues also help sponsor several projects including: promoting Forth education in our universities and colleges, an annual Forth industry survey, an ANS Forth X3J14 representative and "Forthics," the creation of Forth programming "ethics" and metrics to foster more successful Forth projects. You'll read about the results first only in the **SIGForth** newsletter.

Join SIGForth Now!

Send your name, mailing address and ACM number (if applicable) with payment by check (payable to ACM), money order or credit card to the address below. **SIGForth** membership fees are: Non-ACM members $42. ACM Student members $11. ACM members $20. Library subscriptions $33. Foreign air shipment add: $6 (partial), $8 (full).

For additional information contact:
ACM, 11 West 42nd St, New York, NY 10036 • (212) 869-7440

lected points you raised, just for fun. Wil Baden had proposed the words CHAR and [CHAR] which were amended to be ASCII and [ASCII] (TP88-114) before adoption. He had been specifically interested in pursuing the decoupling from the ASCII character set. The committee was not quite ready to fully support this decoupling at that time. We have—through the definitions of address units and the words CELL+, CELLS, BYTE+, and BYTES—admitted to the possibility of storage allocation exceeding eight bits. However, "byte" as data is still defined as an assembly of eight bits, "character" is still "a single-byte value," and the definition of "character" still references 2.1 (Referenced Standards) which consists of ASCII. I do not recall voting on the words "implementation defined" there, and in fact this seems to have been an editorial (!) change. Suggest that you submit a proposal that forces this issue; we need one.

I hope it isn't too soon to use the term "cell" again. For gosh sake, it's been more than half a decade and will be nearly a full one before this thing has a pretty binding to wear. I'd like to be able to use the term again within my own lifetime...

Wrap around numbers are an interesting concept on a one's complement computer. Put simply, they don't work that way. However, this doesn't mean that your code is broken. *Your code has always had a dependency on two's-complement hardware if you use such numbers; hence, it has always been broken relative to running on a one's-complement machine.* It will still run just fine on the type of hardware that the technique depends on. What is missing here is some good prose to express what we are trying to say. There is absolutely nothing wrong with writing code that inherently depends on a particular ALU type or even cell width (like 0< to test high bit of a Boolean mask). This merely means that the application uses techniques that are hardware dependent, and the last thing Forth should do is forbid people to exploit the hardware they have to work with. However, as soon as one starts doing so, he is writing code that is *hardware dependent* and should happily admit it. All we are trying to do here is to clarify what you can do that is *not* hardware dependent and call a spade a spade. Since there are many such dependencies in conventional usage of the same set of operators for dealing with the various data types (pairs versus doubles; numbers versus flags versus Boolean masks), there is clearly much work left to be done.

The key thing about the "contractions" 2* and 2/ is that we inherited them from Forth-83 defined as shifts, so they are not in fact contractions at all, at least at present. The definition of 2* as an "arithmetic" left shift has always given me a chuckle, since if one is going to insist on two's-complement hardware there is, of course, no need to make the distinction. On the other hand, one who uses this definition for 2* on a one's-complement machine to manipulate bits will be surprised to find that the arithmetic left shift is and must be *circular*. Unless the committee is willing to change this, we will need to add controlled reference words that guarantee specific bit-oriented shifting operations.

How to pronounce our favorite words is almost as good as counting angels for consuming debate time. I am hoping that the committee can avoid getting bogged down in this relatively irrelevant nonsense until the important part of the document makes sense.

SP@ was deleted because its existence directly conflicts with existing Forth hardware, such as the Novix and Harris chips. To the extent that programmer portability is important, dependency on stack addressing techniques forms habits that simply don't work and cannot be reasonably implemented on such hardware. Even though a word is merely "controlled," it is quite often the case that implementors are effectively forced to support everything in the book. If it can be shown that a word cannot be reasonably implemented on hardware *designed* to run Forth, then that word is an excellent candidate for the silent treatment.

Finally, in reference to your remark about trying to figure out how to run a Forth-83 program on a BASIS5 system: I don't know about you, but most of my systems are nearly BASIS5, so they would probably run reasonably well. If you have arithmetic that depends on a 16-bit ALU, you will have a problem. In my own experience, conversion to a 32-bit or larger machine has not led to much of this. However, I will admit that you may have some difficulty transporting some code that depends on 16-bit two's-complement byte addressing to many of the world's architectures *however* you do it. One way to deal with the problem is to assert that you don't plan on using such equipment—then you don't have a problem any more. If, instead, you plan on such broad portability, then you (like all the rest of us) will probably need to clean up your act a little bit. We all play fast and loose with data types!

Thanks again—Greg B.

Gary Smith and Johnnie, his wife of twenty-one years, reside in Little Rock, Arkansas, where Gary is employed as a Senior Customer Engineer for Data-Card Corporation. He began his involvement in Forth as owner of Hawg Wild Software, a vendor dedicated primarily to providing Forth kernels for the Timex-Sinclair home computers. He is founder and continuing coordinator of the Central Arkansas chapter of FIG (CAFIG). A long-time proponent of telecommunications as a connecting link for users of Forth, Gary serves as co-Sysop on GEnie's Forth RoundTable and as fairwitness for the Forth Conference on Wetware Diversions, a Unix BBS.

NEW CHAPTER COORDINATOR

THE VALUE OF FIG CHAPTERS

JACK WOEHR - 'JAX' ON GEnie

The Forth Interest Group is a fraternal organization of a rare sort, the type wherein amateurs and dilletantes concourse freely with the very best in the field of their mutual interest. That this is so is in part due to the nature of our discipline; Forth from the start was meant to be the medium of the talented individual bridling under the restrictions of more formal language specifications. It is further due to the fortuitous circumstances that have rewarded the hard labours of the pioneers of Forth with a body of dedicated students of the Art of Keeping It Simple.

It is my opinion that the continued association of Forth programmers and devotees is essential to the continued economic viability of Forth and to the continued marketability of the skills we have honed and the tools we have shaped together.

I know from personal experience just how powerful that mutual association of like-minded individuals can be, having worked for two of the last three years in full-time Forth programming positions obtained by tapping the human resources of the Forth Interest Group, first meeting a future employer at a Silicon Valley FIG meeting, later answering an ad on the GEnie Forth Interest Group RoundTable and securing the position offered there.

When I deal with clients who are considering using Forth in embedded systems projects, or considering teaching themselves to program in Forth for the benefit of their budget or of their peace of mind, the oft-expressed concern is "Will this be maintainable?" I assure them that whereas Acadaemia for the most part continues to ignore the existence of Forth except as a "study under glass," the Forth Interest Group is alive and well; and that on any given weekend, in any number of cities, friendly local chapters are educating the next generation of Forth programmers who will maintain and enhance the work of today.

Unfortunately, at times I suspect myself of prevarication; it is not difficult to become concerned over the vitality of the Forth Interest Group local chapter organization, both from observation and from induction based on experience with non-profit organizations that have attempted to be all things to a diverse group of highly individualistic persons sharing an esoteric common interest.

Three points occupy my attention as I accept the kind offer of the FIG Board to allow me to serve my professional organization in the capacity of Chapter Coordinator:

I encourage chapters to appoint a 'designated telecommer.'

1) That FIG is little more than the local chapters plus one thin bi-monthly magazine, and that if the local chapters should fade, and the Forth community cease to support and contribute articles to *Forth Dimensions*, there shall be no FIG;

2) That the future of Forth is in carrying it to the youth of today, who are vastly more computer conscious than most professionals suspect, who are as enthused with Digital Conciousness as my generation was with Inner Conciousness, and who are hungry for the sort of one-on-one contact, instruction, and encouragement that FIG has traditionally provided the newcomer;

3) That the tenous connection between the local chapter and the central organization can be vastly enhanced through a diligent and creative use of telecommunications, through local chapter BBSes, through FIG's international RoundTable on GEnie, and through other media such as USENET.

With the latter goal in mind, I have been attempting to encourage local chapters to appoint a "designated telecommer" to remain in frequent email contact with the Chapter Coordinator. I hope that by the middle of 1989 every Forth Interest Group local chapter will have provided the Chapter Coordinator with a path of electronic communication that will allow me to prepare mass emailings that will reach every local chapter in the world within hours.

Currently, I have received little response on GEnie from local chapters, but on USENET, the international network of computers mostly running the Unix operating system, I have received answers from around the United States and from Western Europe.

If you would like to represent your chapter electronically, please contact me at one of the following telecom venues:
jax@well.UUCP
well!jax@lll-winken.arpa
JAX on GEnie
SYSOP of Realtime Control & Forth Board, 303-278-0364

I look forward to sharing your insights and suggestions on keeping Forth in the vanguard of computer science in the 1990's, and for keeping the Forth Interest Group International abreast of the needs of our small but dynamic community.

(Continued from page 13.)

found. In that case, you should rename the word to be consistent with the flag—for example, you could call it CHECKBAD?.

The final requirement is a word that will accept a number and append a mod-11 check digit to it. The program using the word will make sure that the number passed to it does not exceed eight digits, so the word includes no check for that.

I decided to discard numbers that produce a check digit of 10, but I wanted the stack result to be the same whether or not I got a good code number. The word MAKEDIGIT leaves a flag on top of the stack that tells whether or not the code number is usable, and a double beneath the flag. If the flag is "false" (code number no good), the double still on the stack (10 times the base number) is subsequently discarded. The verbose definition is shown in Figure Two-a; sans commentary, it looks like Figure Two-b. The words FALSE and TRUE are simple constants for 0 and -1, respectively. They are defined to increase the readability of the source code.

From the keyboard, you can make a check-digited code number from 10000 in this way:

```
10000. MAKEDIGIT
```

The decimal point tells Forth that the 10000 is double precision. An equivalent expression is:

```
10000 US>D MAKEDIGIT
```

When executed, the phrase leaves on top of the stack a "true" flag and, under it, the double-precision number 100005.

The phrase `100005 US>D CHECKOK?` leaves a "true" flag on the stack. In an actual program, you would, of course, keep a copy of the code number to use after it was checked:

```
100005 DUP US>D CHECKOK?
```

The routines are short and simple, and the protection is significant, so consider using a mod-11 check digit in any situation in which you have to assign code numbers.

Michael Ham currently is Director of Systems and Programming at CTB/McGraw-Hill in Monterey, California. Prior to that, he developed microcomputer application software in Forth and wrote Forth-related articles.

FIG CHAPTERS

The FIG Chapters listed below are currently registered as active with regular meetings. If your chapter listing is missing or incorrect, please contact Kent Safford at the FIG office's Chapter Desk. This listing will be updated in each issue of *Forth Dimensions*. If you would like to begin a FIG Chapter in your area, write for a "Chapter Kit and Application." Forth Interest Group, **P.O. Box 8231, San Jose, California 95155**

U.S.A.

- **ALABAMA**
  Huntsville Chapter
  Tom Konantz
  (205) 881-6483

- **ALASKA**
  Kodiak Area Chapter
  Horace Simmons
  (907) 486-5049

- **ARIZONA**
  Phoenix Chapter
  4th Thurs., 7:30 p.m.
  AZ State University
  Memorial Union, 2nd floor
  Dennis L. Wilson
  (602) 956-7578

- **ARKANSAS**
  Central Arkansas Chapter
  Little Rock
  2nd Sat., 2 p.m. &
  4th Wed., 7 p.m.
  Jungkind Photo, 12th & Main
  Gary Smith (501) 227-7817

- **CALIFORNIA**
  Los Angeles Chapter
  4th Sat., 10 a.m.
  Hawthorne Public Library
  12700 S. Grevillea Ave.
  Phillip Wasson
  (213) 649-1428

  North Bay Chapter
  2nd Sat., 10 a.m. Forth, AI
  12 Noon Tutorial, 1 p.m. Forth
  South Berkeley Public Library
  George Shaw (415) 276-5953

  Orange County Chapter
  4th Wed., 7 p.m.
  Fullerton Savings
  Huntington Beach
  Noshir Jesung (714) 842-3032

  Sacramento Chapter
  4th Wed., 7 p.m.
  1708-59th St., Room A
  Tom Ghormley
  (916) 444-7775

  San Diego Chapter
  Thursdays, 12 Noon
  Guy Kelly (619) 454-1307

  Silicon Valley Chapter
  4th Sat., 10 a.m.
  H-P Cupertino
  Bob Barr (408) 435-1616

  Stockton Chapter
  Doug Dillon (209) 931-2448

- **COLORADO**
  Denver Chapter
  1st Mon., 7 p.m.
  Clifford King (303) 693-3413

- **CONNECTICUT**
  Central Connecticut Chapter
  Charles Krajewski
  (203) 344-9996

- **FLORIDA**
  Orlando Chapter
  Every other Wed., 8 p.m.
  Herman B. Gibson
  (305) 855-4790

  Southeast Florida Chapter
  Coconut Grove Area
  John Forsberg (305) 252-0108

  Tampa Bay Chapter
  1st Wed., 7:30 p.m.
  Terry McNay (813) 725-1245

- **GEORGIA**
  Atlanta Chapter
  3rd Tues., 6:30 p.m.
  Western Sizzlen, Doraville
  Nick Hennenfent
  (404) 393-3010

- **ILLINOIS**
  Cache Forth Chapter
  Oak Park
  Clyde W. Phillips, Jr.
  (312) 386-3147

  Central Illinois Chapter
  Champaign
  Robert Illyes (217) 359-6039

- **INDIANA**
  Fort Wayne Chapter
  2nd Tues., 7 p.m.
  I/P Univ. Campus, B71 Neff Hall
  Blair MacDermid
  (219) 749-2042

- **IOWA**
  Central Iowa FIG Chapter
  1st Tues., 7:30 p.m.
  Iowa State Univ., 214 Comp. Sci.
  Rodrick Eldridge
  (515) 294-5659

  Fairfield FIG Chapter
  4th Day, 8:15 p.m.
  Gurdy Leete (515) 472-7077

- **MARYLAND**
  MDFIG
  Michael Nemeth
  (301) 262-8140

- **MASSACHUSETTS**
  Boston Chapter
  3rd Wed., 7 p.m.
  Honeywell
  300 Concord, Billerica
  Gary Chanson (617) 527-7206

- **MICHIGAN**
  Detroit/Ann Arbor Area
  4th Thurs.
  Tom Chrapkiewicz
  (313) 322-7862

- **MINNESOTA**
  MNFIG Chapter
  Minneapolis
  Even Month, 1st Mon., 7:30 p.m.
  Odd Month, 1st Sat., 9:30 a.m.
  Fred Olson (612) 588-9532
  NC Forth BBS (612) 483-6711

- **MISSOURI**
  Kansas City Chapter
  4th Tues., 7 p.m.
  Midwest Research Institute
  MAG Conference Center
  Linus Orth (913) 236-9189

  St. Louis Chapter
  1st Tues., 7 p.m.
  Thornhill Branch Library
  Robert Washam
  91 Weis Drive
  Ellisville, MO 63011

- **NEW JERSEY**
  New Jersey Chapter
  Rutgers Univ., Piscataway
  Nicholas Lordi
  (201) 338-9363

- **NEW MEXICO**
  Albuquerque Chapter
  1st Thurs., 7:30 p.m.
  Physics & Astronomy Bldg.
  Univ. of New Mexico
  Jon Bryan (505) 298-3292

- **NEW YORK**
  FIG, New York
  2nd Wed., 7:45 p.m.
  Manhattan
  Ron Martinez (212) 866-1157

  Rochester Chapter
  Odd month, 4th Sat., 1 p.m.
  Monroe Comm. College
  Bldg. 7, Rm.102
  Frank Lanzafame
  (716) 482-3398

- **OHIO**
  Cleveland Chapter
  4th Tues., 7 p.m.
  Chagrin Falls Library
  Gary Bergstrom
  (216) 247-2492

  Dayton Chapter
  2nd Tues. & 4th Wed., 6:30 p.m.
  CFC. 11 W. Monument Ave. #612
  Gary Ganger (513) 849-1483

- **OREGON**
  Willamette Valley Chapter
  4th Tues., 7 p.m.
  Linn-Benton Comm. College
  Pann McCuaig (503) 752-5113

- **PENNSYLVANIA**
  Villanova Univ. FIG Chapter
  Bryan Stueben
  321-C Willowbrook Drive
  Jeffersonville, PA 19403
  (215) 265-3832

- **TENNESSEE**
  East Tennessee Chapter
  Oak Ridge
  2nd Tues., 7:30 p.m.
  Sci. Appl. Int'l. Corp., 8th Fl
  800 Oak Ridge Turnpike
  Richard Secrist
  (615) 483-7242

- **TEXAS**
  Austin Chapter
  Matt Lawrence
  PO Box 180409
  Austin, TX 78718

  Dallas Chapter
  4th Thurs., 7:30 p.m.
  Texas Instruments
  13500 N. Central Expwy.
  Semiconductor Cafeteria
  Conference Room A
  Clif Penn (214) 995-2361

  Houston Chapter
  3rd Mon., 7:45 p.m.
  Intro Class 6:30 p.m.
  Univ. at St. Thomas
  Russell Harris (713) 461-1618

- **VERMONT**
  Vermont Chapter
  Vergennes
  3rd Mon., 7:30 p.m.
  Vergennes Union High School
  RM 210, Monkton Rd.
  Hal Clark (802) 453-4442

- **VIRGINIA**
  First Forth of Hampton Roads
  William Edmonds
  (804) 898-4099

  Potomac FIG
  D.C. & Northern Virginia
  1st Tues.
  Lee Recreation Center
  5722 Lee Hwy., Arlington
  Joseph Brown
  (703) 471-4409
  E. Coast Forth Board
  (703) 442-8695

  Richmond Forth Group
  2nd Wed., 7 p.m.
  154 Business School
  Univ. of Richmond
  Donald A. Full
  (804) 739-3623

- **WISCONSIN**
  Lake Superior Chapter
  2nd Fri., 7:30 p.m.
  1219 N. 21st St., Superior
  Allen Anway (715) 394-4061

INTERNATIONAL

- **AUSTRALIA**
  Melbourne Chapter
  1st Fri., 8 p.m.
  Lance Collins
  65 Martin Road
  Glen Iris, Victoria 3146
  03/29-2600
  BBS: 61 3 299 1787

  Sydney Chapter
  2nd Fri., 7 p.m.
  John Goodsell Bldg., RM LG19
  Univ. of New South Wales
  Peter Tregeagle
  10 Binda Rd., Yowie Bay 2228
  02/524-7490

- **BELGIUM**
  Belgium Chapter
  4th Wed., 8 p.m.
  Luk Van Loock
  Lariksdreff 20
  2120 Schoten
  03/658-6343

  Southern Belgium Chapter
  Jean-Marc Bertinchamps
  Rue N. Monnom, 2
  B-6290 Nalinnes
  071/213858

- **CANADA**
  BC FIG
  1st Thurs., 7:30 p.m.
  BCIT, 3700 Willingdon Ave.
  BBY, Rm. 1A-324
  Jack W. Brown (604) 596-9764
  BBS (604) 434-5886

  Northern Alberta Chapter
  4th Sat., 10a.m.-noon
  N. Alta. Inst. of Tech.
  Tony Van Muyden
  (403) 486-6666 (days)
  (403) 962-2203 (eves.)

  Southern Ontario Chapter
  Quarterly, 1st Sat., Mar., Jun., Sep., Dec., 2 p.m.
  Genl. Sci. Bldg., RM 212
  McMaster University
  Dr. N. Solntseff
  (416) 525-9140 x3443

  Toronto Chapter
  John Clark Smith
  PO Box 230, Station H
  Toronto, ON M4C 5J2

- **ENGLAND**
  Forth Interest Group-UK
  London
  1st Thurs., 7 p.m.
  Polytechnic of South Bank
  RM 408
  Borough Rd.
  D.J. Neale
  58 Woodland Way
  Morden, Surry SM4 4DS

- **FINLAND**
  FinFIG
  Janne Kotiranta
  Arkkitehdinkatu 38 c 39
  33720 Tampere
  +358-31-184246

- **HOLLAND**
  Holland Chapter
  Vic Van de Zande
  Finmark 7
  3831 JE Leusden

- **ITALY**
  FIG Italia
  Marco Tausel
  Via Gerolamo Forni 48
  20161 Milano
  02/435249

- **JAPAN**
  Japan Chapter
  Toshi Inoue
  Dept. of Mineral Dev. Eng.
  University of Tokyo
  7-3-1 Hongo, Bunkyo 113
  812-2111 x7073

- **NORWAY**
  Bergen Chapter
  Kjell Birger Faeraas,
  47-518-7784

- **REPUBLIC OF CHINA**
  R.O.C. Chapter
  Chin-Fu Liu
  5F, #10, Alley 5, Lane 107
  Fu-Hsin S. Rd. Sec. 1
  TaiPei, Taiwan 10639

- **SWEDEN**
  SweFIG
  Per Alm
  46/8-929631

- **SWITZERLAND**
  Swiss Chapter
  Max Hugelshofer
  Industrieberatung
  Ziberstrasse 6
  8152 Opfikon
  01 810 9289

SPECIAL GROUPS

- **NC4000 Users Group**
  John Carpenter
  1698 Villa St.
  Mountain View, CA 94041
  (415) 960-1256 (eves.)

NEW PUBLICATIONS

328 - 1988 ROCHESTER PROCEEDINGS
Programming Environments
The 1988 Rochester Forth Conference, sponsored by the Institute for Applied Forth Research, Inc. was held June 14-18, 1988 at the University of Rochester in Rochester, New York. Proceedings include over 50 papers.

809 - MORE ON NC4000
Volume 9 - November 1988
This publication includes: Forthkit-3 non-optimizing compiler by Thor-Bjorn Bladh; Enhanced cmForth decompiler by David Doupe; NC4000 LISP kernel by Ulrich Hoffmann; Instruction stack effects for NC4016 by Paul Lambrix; A novel parallel computing structure using NC4000 by C. H. Ting and more.

350 - F-PC USERS MANUAL
351 - F-PC TECHNICAL REFERENCE MANUAL
F-PC is a public domain Forth system optinized for IBM-PC/XT/AT type computers under MS-DOS. It was developed by Tom Zimmer with contributions from Wil Baden, Robert L. Smith, Charles Curley, Jerry Modrow and others. Disks are available on some Forth Bulletin Boards and from Offete Enterprises, Inc.

306 - ANS X3J14 BASIS DOCUMENT
BASIS is the working document of X3J14, the committee chartered to write a draft proposed American National Standard (dpANS) for Forth. That document, dpANS, might become American National Standard Forth. BASIS is an early precursor of dpANS, a framework for S3J14. BASIS is very fluid. You are encouraged to submit proposals toward development.

NOW AVAILABLE
FROM THE FORTH INTEREST GROUP

Forth Interest Group
P.O.Box 8231
San Jose, CA 95155

Second Class
Postage Paid at
San Jose, CA