$10 Volume XIV, Number 1 May 1992 June

FORTH

DIMENSIONS

SC32 Debugging Tools

Object-Oriented Forth

Curly Control Structure Set (II)

Working with CREATE ... DOES>

Contents

Features

Departments

Editorial

So What's New?

Welcome to a new volume-year of *Forth Dimensions.* To commemorate this new beginning, we have been preparing—in conjunction with our talented and dedicated contributors—an infusion of fresh material.

"On the Back Burner," a new department, is engineer Russell Harris' forum for hardware-software projects that readers can build and program. Its intent, apart from the enjoyment and education inherent in building programmable devices that work, is to offer proof (e.g., to prospective employers and project managers) that Forth and the programmer can get the job done. (The clever "gizmo" from the World's Fastest Programmer contest several years ago is but one example of the genre.) Russell's first installment, "Demonstrating Competency," explains the *raison d'etre* for the department, and invites ideas and submissions from readers—the success of this undertaking will rely greatly on the response and participation of you, the reader.

"Fast Forthward" is another new feature to appear regularly. It offers space for product news and announcements, short profiles of Forth companies, and essays about what makes a Forth business/programmer successful and about the nature of Forth. This synergy of Forth users, vendors, and developers should help us to to collaborate more closely, to communicate about Forth more effectively with the rest of the world, and to focus special attention on the things Forth does well.

We are doing our best to encourage Forth vendors and developers to participate in *FD* in other ways, too. Adding to the valued presence of our advertisers, this issue welcomes editorial contributions from three businesses. A number of readers requested this kind of perspective in *FD,* and the Forth-business community has responded well. We look forward to hearing from other companies about their Forth products and their experiences in the commercial world. If your firm would like to participate, get in touch with me soon to discuss the options. And remember to send us your press releases about upgrades, new products, and your company's background. Our readers want to hear from you!

Tutorials Wanted!

Some things bear repeating, like the basics of `CREATE ... DOES>`. Leonard Morgenstern's article in this issue tackles that perennial nightmare of Forth neophytes. If someone once helped you by explaining a particularly thorny topic, why not return that favor for the up-and-coming generation of Forth programmers?

I recently got a phone call from a gentleman in the Midwest; he appreciates Forth over other languages, but hasn't yet achieved the degree of proficiency required to benefit from many of *FD's* intermediate and advanced articles. Would we ever, he asked, be publishing more tutorials? I told him the truth: we'd love to, but they are too rarely seen crossing the editor's desk.

Please consider this a call for tutorials. Perhaps a topic springs to your mind even now—chances are, some of our readers need to hear about it. And a FIG Chapter looking for a group project should consider putting its collective genius to work developing a list of such likely topics and jointly developing a series of short, written tutorials with succinct coded examples.

As many of you have noted over the years, there is a dearth of Forth learning resources. Won't you help to relieve this need? After all, Forth's success will ultimately depend on new people learning to use it. (And if you know of any Forth classes and workshops, let us know so we can add them to our "reSource Listings.")

Have You Renewed Lately?

As a final note, check to be sure you have renewed your FIG membership recently. This issue may have been sent as a courtesy even if your membership expired with the last issue. We value your continued participation and are looking forward to an exciting year ahead. So, please, don't let *this* issue be your last...

—Marlin Ouverson
Editor

P.S. See our call for papers and contest announcement on page 22!

Forth Dimensions
Volume XIV, Number 1
May-June 1992

Published by the
Forth Interest Group

Editor
Marlin Ouverson

Circulation/Order Desk
Anna Brereton

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

Subscription to *Forth Dimensions* is included with membership in the Forth Interest Group at $40 per year ($52 overseas air). For membership, change of address, and to submit items for publication, the address is: Forth Interest Group, P.O. Box 8231, San Jose, California 95155. Administrative offices: 408-277-0668. Fax: 408-286-8988

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 $40/46/52 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

Letters to the Editor—and to your fellow readers—are always welcome. Respond to articles, describe your latest projects, ask for input, advise the Forth community, or simply share a recent insight. Code is also welcome, but is optional. Letters may be edited for clarity and length. We want to hear from you!

No Commerce, No Forth

Dear Editor,

If there is no commercial Forth, no commercial hype, and no commercials, there is no Forth. I would like to hear about the activities of the firms who use Forth for their livelihood or who provide Forth development systems for a fee. If such firms succeed, Forth will also; if FIG's aim is to promote Forth, then it must promote those who use it. *Forth Dimensions* is a bit of a bore, lots of articles on ideas that have little to do with commercial reality.

Charles Esson
CVS
11 Park Street, Bacchus Marsh
Victoria 3340, Australia

Ideal Time for an 'End Run'

Dear Marlin,

Forth does what no other language can do. It allows the user to map his or her working environment to a computer in a direct and consistent fashion. This allows the user to solve problems using familiar models and terms.

This is of little or no value to professional programmers. They prefer C and C++ because they recognize this language no matter what the environment or problem. That is why they do not and will not use Forth. However, using familiar terms in a familiar environment is very valuable to everyone else. Therefore, I propose that the Forth community do an end run around other programmers.

This maneuver would have two stages. In the first stage, using ANS Forth, we build a graphic, and possibly object-based Forth. Instead of using graphics to hide the machinery of Forth, we use the graphic interface to make the simple Forth machinery visible, accessible, and understandable. Users will be able to assemble small Forth pieces into their own applications and will learn to modify their environment as they get more comfortable. This environment is ported to Macs, DOS, OS/2, and Unix machines, allowing the user to operate in the same way and with the same environment on all of the operating systems.

The second stage builds on the first stage, using the Forth chips now available to build expandable Forth computers that run this environment quickly and more efficiently than existing machines can run it. Since Forth lends itself to multitasking and multiprocessing, a basic unit with one Forth chip could be bumped to, say, four or eight chips as more power became necessary. The additional chips would behave as coprocessors or as dedicated I/O devices. They could be both, since they can be switched from one type of task to another by changing the software they run.

Now is an ideal time to pursue this approach. The new wave of consumer electronics provides a lot of opportunities to make inroads into the non-programming world. The multimedia devices that are being introduced this year require simple, easy to use, low-memory methods of programming. Sounds like Forth to me.

So let's get started. I've been playing around with ways to do what I've proposed and I'm eager to take it further. Remember, "the Future starts tomorrow."

Regards,

Mark Martino
170–11th Avenue
Seattle, Washington 98112

10 Forth Commandments

by Tom Napier • North Wales, PA

1. These commandments are not carved in stone; thou mayst change them if thine application demandeth.
2. He who changeth these commandments shall not do so lightly, and shall document the change in his program.
3. Thou shalt put thine application into words, and these words shall be thy program.
4. The lord Moore has given thee many of the words of thy program, and the remainder shalt thou create.
5. Thou shalt use no word in thy program before that word has been defined.
6. Thy parameters shall precede thine operations, and thine operations shall remove their parameters from the stack.
7. Thou shalt be sparing in thy use of the return stack and shall at all times keep it balanced, lest thy program depart for the land of thy fathers.
8. There shall be no goto found in thy code. Thy program shall use if-else-endif, counted loops, repeat-while, and repeat-until.
9. If thine application needeth a structure or a data type which does not exist, thou mayst create a new structure or data type.
10. Thou shalt tell thy fellow programmers what new structures and data types thou hast created, that the wheel shall not too often be invented.

A Single-Step Debugger

and Other Tools for the SC32 Processor

Rick Grehan
Peterborough, New Hampshire

The SC32 is a 32-bit, stack-based processor designed specifically for executing high-level, Forth-like languages. It can directly execute two gigabytes of code memory and 16 Gb of data memory. Good descriptions of the SC32 can be found in the March-April 1990 issue of Forth Dimensions ("SC32: A 32-Bit Forth Engine" by John Hayes) and in Philip J. Koopman Jr.'s book *Stack Computers, The New Wave* (1989, Ellish Horwood Ltd., Chichester, West Sussex, England).

Silicon Composers' SC/FOX parallel coprocessing system (PCS32) offers an SC32 on a PC XT/AT-compatible plug-in card. The PCS32 runs the SC32 at 10 MHz, achieving execution speeds of 10 to 15 MIPS. Thanks to the SC32's pipelined design, the system can execute an instruction per clock cycle. Furthermore, since multiple Forth primitives can be combined into a single SC32 instruction, a PCS32 operating with a 10 MHz clock can hit "burst" execution speeds of up to 50 MIPS.

On the software side, the PCS32 is supported by Silicon Composers' SC/Forth32, a Forth-83-compliant system with 32-bit extensions added to harness the capabilities of the SC32. The PCS32 uses the host PC as an elaborate I/O server; the host PC gives the PC32 disk storage, keyboard, and video I/O.

Working on a recent project, I produced a massive amount of code on the PCS32 system. As the number of words and their interactions grew, it became obvious to me that some sort of debugger would speed the development process. In spite of all my Forth coding abilities, bugs inevitably crept into my work and the system would crash during a testing cycle. A debugger would help me home in on the crash site more rapidly. Unfortunately, SC/Forth32 included no debugger. I had to build one. (The source code for the debugger is shown in Listing One.)

Requirements

My needs were not extravagant; I didn't require breakpoints or multi-step executions. I simply wanted a way to single-step through a word's component instructions and watch the stack effects. I also needed to be able to exit to Forth to check the states of variables.

I wanted the debugger to display, at each instruction step, the name of the word it was about to execute. In some sense, you could say that the SC32 supports subroutine-threaded Forth; the SC32's "call" instruction (which works much like any other CPU's subroutine call) does the nesting job of the inner interpreter. This meant the debugger had to extract the call's destination address—which pointed to the body of the word being called—and "back up" to the name field address. This is handled by the word `HISNAME` in Listing One.

Debugger Internals

The main debugging loop is within the word `DLOOP` (see Listing One). `DLOOP` is simply a large `BEGIN ... AGAIN` structure that endlessly fetches instructions and executes them in a controlled fashion. The only way out of `DLOOP` is when the debugger executes the final instruction of whatever word is being debugged. Execution of the final instruction will inevitably cause the return stack to be popped, which has the effect of exiting `DLOOP` and the debugger.

While I have some complaints about the SC32's cell-based architecture (it makes string handling a nightmare), it became a real blessing as I struggled to build the debugger. Unlike processors with instructions of varying length, the SC32's instructions are all 32 bits (one cell) long.

The SC32 instruction types fall into eight categories (see Figure One on page 11). The top three bits of an instruction determine its type. It turns out that it was sufficient to have the debugger treat instructions as though they fell into one of four categories: call, unconditional branch, conditional branch, and everything else. Although the debugger handles several different instruction types identically, the system will nonetheless tell the user what the instruction type is.

Call

To handle call instructions, the debugger first fetches the instruction that would ordinarily execute. It masks out the upper three bits, leaving the destination address in that instruction's lower 29 bits. This value is placed on the parameter stack, and the debugger can simply use the Forth word `EXECUTE` to go where the call would have gone.

The debugger keeps track of where it is inside a word being debugged via the global variable `HISIP` (short for "his instruction pointer"). `HISIP` serves as a simulated instruction pointer; upon each loop through the debugger, the system

Listing One. Single-step debugger.

```
( ** )
( ** Single-step debugger for SC/Forth32 )
( ** Copyright, 1991 )
( ** Rick Grehan      )
( ** Hancock, NH      )
( ** )

( ** )
( ** Storage
( ** )
VARIABLE  HISIP             ( His instruction pointer )
VARIABLE  HISFLAG           ( His FL bit )
VARIABLE  HERELOC           ( Location for inline execution )
CREATE NUMBUF 4 ALLOT       ( Buffer for number input )

HEX
6008242C uCODE GFLAG        ( Put FL on stack )

( ** INSTRUCTION TYPES ** )
00000000  CONSTANT  ISCALL    ( Call )
20000000  CONSTANT  ISBRAN    ( Unconditional branch )
40000000  CONSTANT  IS?BRAN   ( Conditional branch )
60000000  CONSTANT  ISALUS    ( ALU/shift )
80000000  CONSTANT  ISLOAD    ( Load )
A0000000  CONSTANT  ISSTORE   ( Store )
C0000000  CONSTANT  ISLAL     ( Load addr low )
E0000000  CONSTANT  ISLAH     ( Load addr high )

DECIMAL

( ** )
( ** Improved dump )
( ** )
( Dump 16 bytes in hex starting at byte address baddr )
: 16HEXBYTES     ( baddr -- )
  DUP 8 HEX .R DECIMAL ." : "
  16 0 DO
     I OVER + C@ 2 HEX .R DECIMAL
     SPACE
  LOOP
  DROP  ;

( Dump 16 bytes in ascii starting at byte address baddr )
: 16ASCIIBYTES ( baddr -- )
  16 0 DO
     I OVER + C@
     127 AND
     DUP 32 <           ( Printable? )
     IF   DROP ASCII .
     THEN
     EMIT
  LOOP
  DROP  ;

( Super byte dump from byte address baddr )
: SDUMP    ( baddr n -- )
  CR
  BEGIN
     OVER 16HEXBYTES 4 SPACES
     OVER 16ASCIIBYTES CR
     16 - DUP
     0>
  WHILE
```

uses the address stored in `HISIP` to determine the location of the next instruction.

Consequently, the portion of the debugger handling call instructions increments `HISIP` by one before exiting.

Unconditional Branch

The debugger takes care of unconditional branch instructions by simply masking out the high three bits of the instruction, thereby leaving only the jump's destination address. The unconditional branch handler then places this address in `HISIP` and passes back to the start of the loop.

Conditional Branch

On the SC32, a conditional branch instruction will take the branch if the `FL` bit is set to zero. This is a processor flag that can be modified by ALU shift instructions. Consequently, for the debugger to know whether a conditional branch should be taken or stepped over, it has to simulate the setting of the processor's `FL` bit.

I accomplished this by creating a machine-code instruction called `GFLAG` (for "get flag") that places the contents of the `FL` bit on the parameter stack. After the debugger executes any instruction in the target code that may affect `FL`, it calls `GFLAG` and stores the parameter stack in the variable `HISFLAG`.

So, when the debugger encounters a conditional branch, it simply examines the contents of `HISFLAG`. If `HISFLAG` is zero, the debugger treats the instruction as an unconditional branch and the branch is taken. Otherwise, the debugger merely increments `HISIP` by one to skip to the next instruction.

Everything Else

The debugger executes all other instructions—arithmetic/logical, shift, and load/store—as is. It does this by fetching the instruction pointed to by HISIP and placing that instruction in-line. The following is the SC/Forth32 code fragment for doing this:

```
VARIABLE HERELOC
...
IFETCH HERELOC @ !
( Put the instruction
( in-line )
[ HERE HERELOC !
( Set HERELOC )
  0 , ]
( Make room in the
( dictionary )
...
```

The word IFETCH retrieves the instruction pointed to by HISIP. The debugger stores that instruction at the address stored in HERELOC. As you can see by the code between [and], HERELOC is set to point to an initially empty cell within the debugger's stream of execution. Simply put, the debugger patches itself on the fly, the patch being the instruction fetched from the location given by HISIP.

Finally, after the in-line instruction has executed, the debugger uses the GFLAG word mentioned earlier to save the state of the FL bit.

User Input

While you're in the debugger, the system gives you the option of entering a variety of single-character commands at each execution step. These commands are:

F Allows the user to temporarily suspend the debugger and go to Forth. This command simply calls the SC/Forth32 word INTERPRET. The debugger defines an additional word, RESUME,

```
      SWAP 16 + SWAP
  REPEAT
  2DROP  ;

( ** )
( ** Debugger )
( ** )
HEX
( Fetch his next instruction )
: IFETCH   ( -- n )
  HISIP @ @  ;

( Mask out jump address for calls and branches )
: JADDR    ( -- n )
  IFETCH 1FFFFFFF AND  ;

( Mask out instruction type )
: ITYPE    ( -- n )
  IFETCH E0000000 AND  ;

DECIMAL

( Safely print the stack.  This won't bomb if the stack
( has undeflowed. )
: SSTACK
  DEPTH 0<
  IF      ." Underflow "
  ELSE    .S
  THEN  ;

( Given the byte address of a name field, print it )
: SHONAME        ( baddr -- )
  DUP C@ 127 AND           ( Get count )
  ?DUP                     ( Anything there? )
  IF
     0 DO
          1+ DUP C@ 127 AND EMIT
     LOOP
     SPACE
  THEN
  DROP  ;

( Given the cell addr. of a code field, do your best to locate )
( the associated name field and print it. Works in most cases.  )
: HISNAME        ( addr -- )
  BYTE                ( Convert to byte address )
  0                   ( Start a counter )
  BEGIN
     SWAP 1- DUP C@ 127 AND          ( Fetch a character )
     DUP 32 <>                       ( Null? )
     IF   32 <                       ( Printable? )
          IF   DUP C@ 127 AND        ( Fetch it again )
               2PICK =               ( Equal to our count? )
               IF   SWAP DROP
                    SHONAME          ( We got it! )
                    EXIT             ( Go home )
               THEN
          THEN
          SWAP 1+                    ( Increment counter )
     ELSE DROP SWAP                  ( Don't increment )
     THEN
     DUP 33 =                        ( Name can't be this big )
  UNTIL
  2DROP  ;
```

```
( Display the current instruction type )
: SHOTYPE
  ITYPE
  SELECT
     CASE ISCALL = OF     ." CALL: " BREAK
     CASE ISBRAN = OF     ." BRANCH: " BREAK
     CASE IS?BRAN = OF    ." ?BRANCH: " BREAK
     CASE ISALUS = OF     ." ALU/SH: " BREAK
     CASE ISLOAD = OF     ." LOAD: " BREAK
     CASE ISSTORE = OF    ." STORE: " BREAK
     CASE ISLAL = OF      ." LAL: " BREAK
     CASE ISLAH = OF      ." LAH: " BREAK
  NOCASE BREAK  ;

( Get a hexadecimal number from the keyboard )
: NUMIN        ( -- n )
  0 NUMBUF !                  ( Clear receiving buffer  )
  NUMBUF BYTE 10 EXPECT       ( User inputs number here )
  BASE @ HEX                  ( Set base to hexadecimal )
  NUMBUF BYTE 1- NUMBER       ( Convert )
  2DROP SWAP BASE !         ; ( Restore base )

( Exit to forth from debugger )
: TOFORTH ( -- )
  ." TO FORTH " CR
  INTERPRET
  ." BACK TO DEBUG " CR  ;

( Return to the debugger )
: RESUME R> DROP ;

( Get user input at each debugger step )
: USERIN
  BEGIN
    0
    KEY
    SELECT
        CASE ASCII F = OF      ( Shell out to Forth )
                               TOFORTH BREAK
                             CASE ASCII Q = OF    ( Abort )
            1 ABORT" ** ABORTED! " BREAK
      CASE ASCII I = OF        ( Display current instruction )
            BASE @ IFETCH ." (" HEX
              . BASE ! ." )" CR BREAK
      CASE ASCII D = OF        ( Dump )
            ." ADDR:" NUMIN    ( Address )
            ." LEN:" NUMIN     ( Number of bytes )
            SDUMP BREAK
    NOCASE DROP 1 BREAK        ( Anything else continues )
  UNTIL  ;

( Main debugger loop )
: DLOOP
  BEGIN
      SHOTYPE                  ( Show instruction type )
      ITYPE                    ( Fetch it and select )
      ISCALL = IF              ( ** CALL ** )
            JADDR HISNAME      ( Show word's name if possible )
            SSTACK             ( Show the stack )
            USERIN             ( Get user input )
            JADDR              ( Get call's destination address )
            EXECUTE            ( Execute the word )
            1 HISIP +!         ( Bump instruction pointer )
```

that returns the user to the debugger where he left off. Currently, these words make no attempt to save and restore the parameter and return stacks. It's up to you to make sure the stacks are in the same state when you execute `RESUME` as when you left the debugger.

`I` Displays in hexadecimal the instruction the debugger is about to execute. I found this handy for ALU/shift instructions, since the debugger simply announces them as "ALU/SH." With the `I` command, you can disassemble an instruction whose operation you are unsure of (provided you have the manual of SC32 instruction formats handy).

`D` Provides quick access to a memory dump. The debugger will prompt you for the starting cell address and the number of cells to dump.

`Q` Executes an `ABORT`, quitting the debugger and returning to Forth.

Entering any other character at the execution steps will cause the debugger to proceed with the next instruction.

Problems and Improvements

Recognizing SC/Forth Primitives

Since the SC32 was designed from the ground up to execute Forth (and thanks to the optimization of the SC/Forth32 compiler), some of the more complex Forth primitives are compiled into a series of obtuse SC32 instructions. For example, if you encounter the Forth word `DO` in the debugger, you won't see a call to the location of `DO`, you'll see a series of SC32 instructions that load the return stack with initial and terminal loop index values. (Actually, the values loaded on the return stack are not the initial and final loop values. The effect is the same, however.)

Step Into

In its current incarnation, the debugger handles call instructions using the SC/Forth32 word `EXECUTE`. Consequently, there is no way to "nest down" a level and step into a word. In order for the debugger to perform that feat, you would have to add code that kept the variable `HISIP` properly tracking the instruction pointer of the debugged code. The debugger would also have to take over the responsibility of managing the return stack. Specifically, whenever the debugger encountered a call instruction, it would push the incremented value of `HISIP` onto the return stack, extract the destination address from the instruction, and store that address into `HISIP`.

Handling a return from subroutine is more difficult, since the SC32 actually embeds the return operation in ALU/shift or load/store instructions. Bit 28 of such instructions is called the "next" bit. If it is set, it loads the top value on the return stack into the instruction pointer. Bits 16 through 19 are called the "stack" bits: They determine whether the parameter and return stacks are pushed or popped. If the next bit is set and the stack bits specify that the return stack is to be popped, the effect is a return operation.

So, for the debugger to manage a return, it would have to watch for a set "next" bit within ALU/shift and load/store instructions. Whenever it sees a set bit, it would mask the bit out, transfer the top of the return stack into `HISIP`, and execute the modified instruction.

```
        ELSE
        ITYPE
        ISBRAN = IF                  ( ** UNCOND. BRANCH ** )
              SSTACK                 ( Show the stack )
              USERIN                 ( Get user input )
              JADDR                  ( Get jump address )
              HISIP !                ( New instr. pointer )
        ELSE
        ITYPE
        IS?BRAN = IF                 ( ** COND. BRANCH ** )
              SSTACK                 ( Show the stack )
              USERIN                 ( Get user input )
              HISFLAG @              ( Get his FL bit )
              IF
                    1 HISIP +!       ( Branch not taken )
              ELSE
                    JADDR            ( Branch taken )
                    HISIP !
              THEN
        ELSE                               ( ** ALL OTHERS ** )
              SSTACK                       ( Show the stack )
              USERIN                       ( Get user input )
              IFETCH HERELOC @ !           ( Put instr. inline )
              [ HERE HERELOC ! 0 , ]
              GFLAG HISFLAG !              ( Save flag after operation )
              1 HISIP +!                   ( Incr. his address )
        THEN THEN THEN
        CR
    AGAIN  ;

( The outermost word.  To unleash the debugger on a word, )
( simply enter   DEBUG <wordname> )
: DEBUG
    BL WORD CELL FIND NOT                  ( Is word in dictionary? )
    IF   ." ** NOT FOUND **" CR QUIT       ( Bail out if not )
    ELSE      HISIP !                      ( Set instr. pointer if so )
        ." WORD AT:" HISIP @ HEX .         ( Show word's body address )
        DECIMAL CR
        DLOOP                              ( Enter the loop )
    THEN  ;
```

Listing Two. Execution trace.

```
: TRACE
  R@ 1-              ( Back up to code field )
  HISNAME CR  ;      ( Display name )

: >>TRACE
  ['] TRACE ,        ( Compile TRACE into dictionary )
  [COMPILE] ]  ;     ( Make colon happy )

: TRACEON
  ['] >>TRACE
  ['] :
  8 +                ( Address where ] was )
  !   ;              ( Overwrite it )

: TRACEOFF
  ['] ]
  ['] :
  8 + !  ;          ( Put ] back where he was )
```

Finally, you would want to add an additional user-input choice that would allow the user to select whether the debugger stepped into the called word, or executed it as a whole, as it does now.

Last Calls To Jumps

SC/Forth32 is an optimizing compiler. Among other things, this means that the compiler is intelligent enough to recognize that if the last instruction in the definition of a word is a call instruction, that call can be converted to an unconditional jump. This saves return stack space, as well as reducing some execution time that would ordinarily be unnecessarily consumed moving addresses between the return stack and the instruction pointer.

From the debugger's point of view, the jump instruction is just a jump; there's no indication that this was a call optimized into a jump. If you single-step into this situation, it will appear that you have nested down into a word, and in some severe cases this nesting can go on for several levels as you repeatedly encounter the last instruction of each word. Ultimately, of course, you will encounter a Forth primitive and pop out the end.

Trace

As a final tool, I built a simple execution trace facility. I based the execution trace words on the trace commands found in old, reliable, interpreted BASIC. To refresh your memory, executing `TRACEON` in BASIC would cause the system to display the number of the current line BASIC was executing. This was handy for locating exactly where the system either did a belly-flop or hung in an infinite loop. I wanted a similar construct for my Forth work. I wanted words to tell me when they were about to execute, and I wanted to be able to turn this behavior on and off. As in BASIC, this would make it easier to pinpoint where the program died.

My solution was a pair of words—`TRACEON` and `TRACEOFF`—that you could use as brackets. That is, words compiled after `TRACEON` would display their names when executed. `TRACEOFF` would disable tracing; subsequent words would act normally. I was satisfied to have only colon words be affected by `TRACEON` and `TRACEOFF`. (I could have extended the trace word to cover defining words, but I didn't need that particular feature.)

Trace Operation

`TRACEON` works by patching the `:` (colon) word. The last word in SC/Forth32's definition of `:` is `]`, which puts Forth in the compiling state. The SC32 instruction that calls `]` is located eight cells into the definition of `:`. `TRACEON` overwrites that location with a call to the word `>>TRACE`.

So, after you execute `TRACEON`, whenever `:` executes, it calls `>>TRACE` as its last instruction. `>>TRACE` will compile the word `TRACE` into the dictionary. Hence, `TRACE` becomes the first word executed by whatever word `:` has just defined. `>>TRACE` then executes `]` so that the compiler enters the proper state at the end of `:`. (A side-effect is that words compiled after `TRACEON` are one cell longer than they would ordinarily be.)

Now, whenever the colon-defined word executes, it immediately calls `TRACE`. `TRACE` fetches the return address from the return stack and decrements that address by one cell. The resulting cell address points to the body of the calling word, and `TRACE` can unleash `HISNAME` (described above) to print the name field.

`TRACEOFF` simply unpatches `:`, overwriting the call to `>>TRACE` with a call to `]`. The source to the `TRACE` system is shown in Listing Two.

Figure One. SC32 instruction types.

Instruction Type	Top 3 bits of instruction	Description
Call	000	The SC32 places the return address on the return stack, and jumps to the location given by the instruction's remaining 29 bits.
Branch	001	Same as a call instruction, only the SC32 doesn't place anything on the return stack.
Conditional branch	010	If the SC32's FL flag is zero, this instruction performs a branch. Otherwise, the processor proceeds to the next instruction.
ALU/shift	011	Executes a variety of arithmetic, logical, and shift operations, depending on the remaining 29 bits.
Load	100	Adds an offset (encoded in the lower 16 bits of the instruction) to the contents of a designated source register. The contents of the resulting address are loaded into a designated destination register.
Store	101	Adds an offset (encoded in the lower 16 bits of the instruction) to the contents of a designated source register. The contents of a designated destination register are stored at that address.
Load address low	110	Adds an offset (encoded in the lower 16 bits of the instruction) to the contents of a designated source register. The result is placed in a designated destination register.
Load address high	111	Adds an offset (encoded in the lower 16 bits of the instruction) to the contents of a designated source register *after* shifting that offset to the left 16 bits. The result is placed in a designated destination register.

Designing Software-Controlled Devices

Carol Goldsmith
Victor, New York

When software is involved in product development, the step of integrating hardware and software is fraught with difficulty. Sophisticated development systems, emulators, and logic analyzers exist to help the debugging process. In the conventional approach to embedded system design, a PC is used to write, cross-compile, link, and load code into emulation memory on the target system. One iteration of the laborious and oft-repeated edit, compile, link, and load cycle can easily take ten or 15 minutes for a complex project. This sequence must be enacted for one error in one line of code or many. The agony really begins if the errors are interactive with the hardware—the correction of one exposes another. System debugging is often done via an in-circuit emulator (another expense) that provides breakpoints and other software debugging support. Ever wonder why project managers go gray at an early age?

Forth to the Rescue...

The solution—familiar to most readers of this magazine but largely unknown to most designers—is to include Forth on the controller card, giving users the ability to deal with code on a word-by-word, or line-by-line basis interactively with the target system. Forth's primary benefit for the developer is that it eliminates the middle-man. Both a language and a programming environment, Forth can be developed and executed directly on the target system, so there is no need for the traditional cross-development system required by C or assembler. Forth is interpretive and highly interactive, giving developers the ability to prototype applications swiftly. It offers the designer the unique opportunity to write, test, and run software in real time and avoid the time-consuming steps of the edit, compile, test, debug loop for each single modification. On-board Forth offers in one entity a real-time programming language, an operating system, and a development environment. The natural extensibility of Forth leads to application-specific words that are self-documenting as they are used. Engineers using Forth can design words to suit their specific work. Embedded control is definitely a place where Forth can make a significant impact and become more widely known.

Embedded control is a place where Forth can make a significant impact and become more widely known.

Compilation occurs one word at a time on the target system itself. Each Forth word can be tested as soon as it is entered; if it does not produce the desired result, you can quickly change the word and recompile. This encourages thorough testing of each piece of code as it is written. In contrast, C and assembler have long edit, compile (or assemble), link, and load cycles that make it difficult to test fragments of code. Debugging can't start until most of the framework is in place. Incremental testing speeds project development, because there is a higher probability that the design will work the first time.

Not at all Tedious...

Two economical and easy-to-use controllers which offer extensive on-board Forth are the TDS2020 and the TDS9092 from The Saelig Company (Victor, NY). Well-known in Europe, and becoming recognized in the U.S.A. and Canada, these boards from Triangle Digital Services Ltd. of London (U.K.) have been sold worldwide in their thousands. Both of these nearly-pin-compatible 4" x 3" boards provide a complete Forth design environment—the TDS2020 operating at 20 MHz comes complete with eight channels of A/D, and the slower and cheaper TDS9092 runs at 1 MHz, more suited to simpler control situations. The TDS2020 is a powerful CMOS controller card, based on the Hitachi 16-bit H8/532 microprocessor, and runs at about 3 MIPS. It has 16 Kbytes of Forth as well as a full symbolic assembler, eight channels of ten-bit A/D, three channels of D/A, serial RS232 and I^2C protocols, too. There is 45K for program storage, and up to 512 K NVRAM space on-board, as well as timers, interrupts, and 33 I/O lines.

Lite Programming

Programming is accomplished by downloading suitable words from the PC software provided with the boards. The TDS2020 starter pack includes lots of utility routines to make life easier for the designer. Included are serial input/output, timer, LCD/keyboard driver, memory test, and many other routines. Also available are string-handling routines, trig functions, graphics LCD display, interrupt-driven serial I/O, and round-robin multitasking.

Embed the TDS2020 in a product, talk to it from a PC-compatible down an RS-232 serial line, debugging each segment as you go, and the final code can be stored in NVRAM, with no need for PROM burning. You have very fast development time with no need for in-circuit emulators or test stubs for developing fault-free code. The application also runs at full speed, and the full resources of the development environment are available for use in debugging the application. In the Forth environment, any portion of the code can be exercised at full speed, and breakpoints can be introduced for snapshots, or single stepping.

"Advantage TDS"

When you have developed your product using the TDS2020 or TDS9092 and are now manufacturing it, that is not the end of the story for Forth. It can be used for repair and maintenance because the language is on-board. A connector can be built into the product which gives serial access to the TDS board in your instrument. With a PC or hand-held terminal, you can now gain access to the system. The command ctrl-C allows you to break out of your program and individually exercise all the procedures that make up the software. For instance, you can drive the printer, LCD, keyboard, or A/D routine to determine fault conditions. On-board Forth is very useful during design and debugging, but the ability to access individual software procedures in a finished product is invaluable. This also saves writing lots of "service routines," often requested by servicing departments, and frequently some options get forgotten, requiring new routines to be written. With on-board Forth, it's all there anyway.

Thanks for the Memory...

The TDS2020CM is a useful module which sandwiches on top of the TDS2020 and allows storage of up to 8 Mbytes of non-volatile data on industry-standard JEIDA/PCMCIA card memory, including Flash types. In an application, this removable card can be brought back to base from field data collections and read in another TDS2020 or by a PC with a card memory drive. Meanwhile, the datalogger is storing information on a new card. Datalogging for over a year on a single 9-volt battery is possible, since the TDS2020 only draws 300 μa in standby mode. A complete datalogging program is included with TDS2020 starter pack. In addition to standard fig-Forth, 200 words are supplied with the TDS2020 for simplifying tasks such as data-logging, key-pad and LCD control, stepper-motor driving, interrupt control, etc. The TDS2020 starter pack is $499 and the TDS9092 starter pack is $249, in stock from The Saelig Company (716-425-3753; fax 716-425-3835).

Carol Goldsmith is the Sales Manager for The Saelig Company.

JForth
A 32-bit, Subroutine-Threaded Forth for the Amiga

Phil Burk
San Rafael, California

JForth falls into the category of "big Forths." We at Delta Research believe that Forth development systems should offer the same facilities that C programmers enjoy. While minimal Forths are perfect for small embedded systems, they are inappropriate on larger computer systems. We feel that one of the reasons Forth has not sold as well on large systems is because many Forths adhere to a minimalist philosophy. We feel that Forths for large systems should have all of the file I/O routines, memory allocation, floating point, complex data structures, and other tools that are standard in competing languages. We applaud the ANS standardization efforts that include these facilities.

One of the areas that Forth does not usually compare well with C is in the generation of *small executable images.* We, therefore, added `Clone` which can generate standalone images as small as 3K. `Clone` starts at the top word in an application and disassembles its 68000 machine code, then disassembles all the words called by that word, and so on. It then reconstructs an image without headers and with only the words and data needed by the application.

We wanted JForth programmers to be able to call Amiga system libraries as easily as C programmers.

It also performs some optimizations made possible by the smaller image, such as converting absolute subroutine calls to PC relative. An executable image is then written to disk with an icon. Clone-able programs have a few restrictions related to storing addresses in variables at compile time. These are easily handled, however, by using run-time initialization, or by using `DEFER` for vectored execution.

We wanted JForth programmers to be able to *call Amiga system libraries* as easily as C programmers. To call Amiga system routines, JForth uses a simple `CALL` by name syntax that automatically builds code to move parameters from the data stack to the appropriate 68000 registers.

Since the Amiga relies heavily on passing structures, we implemented a *C-like structure* facility that automatically handles variously sized structure members. Thus, one can fetch a signed byte member or a 32-bit-long member using the same `S@` word. Signed versus unsigned members and address relocation is also handled. Here is an example structure definition plus some code to access it:

```
\ Define structure template
:STRUCT FOO
   LONG  FOO_SIZE
   APTR  FOO_BUFFER
   LONG  FOO_INDEX
   SHORT FOO_SCRATCH
;STRUCT

\ create a FOO structure
FOO MY-FOO
: TEST.FOO  ( -- index scratch )
   MY-FOO S@ FOO_INDEX
   MY-FOO S@ FOO_SCRATCH
;
```

If we use the JForth disassembler to examine `TEST.FOO` we will see that it built the following code:

```
BSR.L   MY-FOO
MOVE.L  $8(A4,D7.L),D7
BSR.L   MY-FOO
MOVE.W  $C(A4,D7.L),D7
EXT.L   D7      \ sign extend
RTS
```

Notice that it used `MOVE.L` for the *long* member, and `MOVE.W` and a sign extension for the *short* member. The top of the Forth data stack is cached in D7, so the results of the fetches are left there. A4 is a register that points to the base of the Forth dictionary and allows us to build relocatable code.

JForth provides other tools, including a *Source-Level Debugger* with single step and multiple breakpoints. The debugger also works with cloned images. A *code perfor-*

Phil Burk is a co-author of JForth and HMSL. His current interests include electronic music, animation, and 56000-based digital signal processing.

mance analyzer in JForth will periodically interrupt an executing program and gather statistics on where it is spending its time. JForth also provides *local variables* that use the following style:

```
: TYPE/2  { addr cnt -- }
      CNT 2/ -> CNT
      ADDR CNT TYPE
;
```

A new feature of JForth is support for IFF ANIM and ANIMBrush files. This utility lets you load animation images from other programs to create animated displays. The output of the Amiga can be plugged directly into a VCR for simple home video.

These, and other features, combine to create a powerful Forth-based application development environment that offers a real alternative for commercial developers.

HMSL
Hierarchical Music Specification Language

HMSL is an extension to Forth that provides MIDI support, and object-oriented compositional tools. The object classes include *Shapes* which are a general purpose array of N-dimensional points. The data can represent a melody, a tuning, a trajectory, or any user-defined parameter. Another class, called *Players*, schedules the conversion of Shape data into musical or other forms of output. *Jobs* schedule user-written functions for repeated execution. *Collections* can contain Players, Jobs, or other Collections, and allow you to create a complex hierarchy of music objects.

HMSL supports *standard MIDI files*. Thus, you can use HMSL to algorithmically create sequences for use with other commercial music programs. An *event buffer* provides low-level scheduling of MIDI events and supports a text-based Score Entry System. Here is an example of a simple score:

```
1/4 C3 F#  1/8 20 /\  A A A A
1/2 _mf CHORD{ E4 G B }CHORD
```

HMSL provides a toolbox for building interactive screens out of control grid objects like check boxes and faders.

The Amiga version of HMSL uses JForth. The Macintosh version has its own built-in Forth. HMSL pieces are generally portable between the Amiga and Macintosh versions.

A number of the other features of JForth and HMSL are mentioned in the accompanying advertisement, so I won't list them here. If you are interested in JForth or HMSL, give us a call and we can direct you to a discount retailer.

Object-Oriented Forth

Markus Dahm
Aachen, Germany

At the Institute for Measurement Techniques at the University of Technology RWTH Aachen, we have used Forth since 1987. Our interdisciplinary workgroup has developed medical image workstations. We have written a lot of software including memory management, image-processing algorithms, fibre optics network coupling, and a graphical user interface in our proprietary 32-bit Forth. The psychologists in our workgroup conduct experiments concerning the software- and hardware-ergonomical aspects of the design and functionality of the workstations using the prototype image workstation.

Some of the student laboratory work in image processing is done in Forth, which is picked up by the students usually within half an hour. Within this short amount of time, they learn enough to program image-processing algorithms.

So, for various reasons, the ease of understanding and getting access to a complex system is of high priority for us. For this purpose, our existing 32-bit Forth did not provide enough programming support and transparency, so we conceived a new and object-oriented Forth.

Within one-half hour, students learn enough Forth to program image-processing algorithms.

The work was funded by the German Ministry for Research and Technology, grant no. BMFT/AuT-01HK577-03, as part of the DIBA-project. Thanks to Maria Irene dos Reis Lourenço-Kaierle for her work on the implementation.

An Object-Oriented Forth

The paradigm of object orientation has been around for quite a while but has recently received a lot of attention. Apart from the hype—it was even called the "silver bullet" to shoot all programming troubles—it is a real advance for programmers in terms of structure, clarity, readability and, thus, useability of both the programming approach and the program code.

Forth's advantages are the interactivity of the interpreted language and the extensibility which allows the language to be fitted to a special application, which make it suited for non-expert users. Moreover, it enables you to test everything easily and directly via the keyboard, which makes debugging easy. Forth supports—almost forces—the method of factoring, which greatly enhances the clarity of programs and thus the programmer's productivity and content.

Our main interest is to work with a programming language that supports fast and easy understanding and debugging, and thus allows rapid prototyping of user interfaces by both engineers and, on a higher level, by psychologists.

OOF strives to achieve this by combining the best of both worlds by extending Forth following the paradigm of object orientation in a strict sense. It provides all its amenities, such as security, inheritance, and late binding. This is achieved by strictly adhering to the concepts of data encapsulation, strong typing, and message passing rather than direct procedure calls. The system still has a small kernel that performs everything from interpreting to compiling the source code in a simple but smart fashion.

The principles of OOF and their consequences are best explained by examples. The use of OOF is therefore described step by step, from simple definitions of objects to the creation and extension of classes and methods, explaining the nomenclature and buzzwords of object-oriented languages en passant.

Here's How

Everything in OOF is an *object*. Every object is an object of some *class* (e.g., integer or character); it consists of a data field and a set of methods to manipulate the data. For example, when you want to create an integer object `start` or two character objects `c1` and `c2`, you write:

```
integer : start ;
character : c1 , c2 ;
```

This shows one of the basic syntax elements, the *colon declaration*, which in Forth only declares words. According to one of my favorite guidelines, simplification by generalization, the colon is used in OOF as the general method of declaration. It can be applied to any class that is known in the system in order to create objects (or *instances*) of this class. If you want to declare more than one object of the same class, the names of the objects separated by commas form a list of objects to be declared, terminated by a semicolon.

Figure One. Defining an instance method.

```
im : size
(( image : i ; -- integer : s ; || integer : pixels ; ))
      i -> xdim   i -> ydim  *  pixels  !
     i -> bits/pixel  pixels  *  s !;
```

Figure Two. Using individual instance methods.

```
object subclass : state ;                      \ define class state
state iim : keypressed (( state : s ; -- ))
        " A key was pressed" print ;    \ define default-reaction
state : idle , input ;          \ define states for state-machine
idle  iim : keypressed (( state : s ; -- ))
        " Idle state: key" print ;   \ define individual reaction
```

Actually, the comma is exactly the same method as the colon. In some cases, the colon method needs some more parameters; e.g., when defining a string, you want to give the maximum number of charactes in the string:

```
30 string : text1 ;
```

If, as a more elaborate example, you want to handle images in your system, you define the new class `image`. You do not want to invent the methods anew for creation, deletion, or debugging methods of objects every time you define a new class. So you let image *inherit* all these properties by declaring `image` a subclass of `object`, the most basic class of all classes, which already provides these properties:

```
object subclass : image ;
```

`image` is now defined as an object of the class `subclass` and, at this moment, has exactly the same properties as the class `object`. The subclass `image` is now going to be extended in order to fulfill the purpose we defined it for. For every image, you need to know, for example, its dimension in x and y and how many bits are in a pixel. These data are part of every object of the class `image`, i.e., that is an *instance* of `image`. Thus, we have to define *instance variables* (i.e., instance objects, but "instance variables" in the typical nomenclature of object-oriented languages; in OOF it is abbreviated as "IV") of `image`:

```
image IV integer : bits/pixel ;
image IV integer : xdim , ydim ;
```

When you want to declare two new images `im1` and `im2`, you write:

```
image : im1 , im2 ;
```

using the general colon declaration. Now you have two `image` objects, each containing one set of the above-defined instance variables. In order to achieve the desired security and consistency, the instance variables of any object may only be modified by the methods that have been declared for its class, the *instance methods* (abbreviated as "`im`"). No method defined for any other class may alter, or even read, these instance variables. One method for the class `image` might, for example, compute the size of an image in bits. You can define this method as in Figure One.

So `size` is defined as an object of class `im`. What is known as the stack comment `( -- )` in Forth, has evolved to a full declaration of input and output parameters as well as local variables in OOF: `(( -- || ))`.

The parameters are defined in the same way as any object: by the colon declaration. The method `size` can refer to the object that was passed to it on TOS as `i`, the object that is to be passed as the result can be referred to as `s`, and `pixels` is a local object. It goes without saying that you can define as many of these temporary objects (here: `s`, `i`, and `pixels`) as you like. Their scope is only within the definition of this method, they cannot be accessed from outside the method. They make possible clear and readable programming without stack juggling, and they ensure that only the values declared are popped off the stack and only the values declared are pushed onto the stack as results. This is performed automatically when entering and exiting the method according to these declarations, thus enhancing security.

The instance variables `xdim`, `ydim`, and `bits/pixel` of the `image` object `i` are accessed by the method `->`, which may only be called inside an instance method for that particular class (here: `image`). This is called *data encapsulation* and ensures that these operations can only be performed on object data that you have explicitly allowed and defined to do so, again enhancing program security.

Note that, in order to push the value of, for example, `pixels` onto the stack, there is no method `@` involved (and thus cannot be forgotten any more). Every object lays itself onto the stack when invoked. The low-level difference between the object's value and its location is no longer visible—there are only objects.

Making Passes

The above-defined method `size` for the class `images` can now be applied to the previously declared `image`-object `im1` by:

```
im1 size
```

Figure Three. Using shallow objects.

```
state ^ : active ; \ creates the shallow state-object "active"
idle  active <-    \ makes the shallow state-object "active" act
                   \ as if it were the state-object "idle"
```

Figure Four. Using shallow objects in individual instance methods.

```
idle iim : keypressed (( state : s ; -- ))
      " Idle state: key, entering input state " print
      input  active  <- ;    \ now make "input" the active state
```

Figure Five. Using the Do ... Loop.

```
im : looptest
(( integer : from , to ; -- || integer : end , run ; ))
      from run !   to 1 + end !    \ set limit and index variables
      end run do
                   run print       \ print the index value
              loop    ;
```

Non-object-oriented languages, such as Forth, call methods directly: they compile the address of the code to be executed. OOF instead sends a message to the object on top of the stack (TOS). In its essence, a message is the name of a method to be called. When a message is passed to the object O on TOS, a method of that name is searched at run time in the list of all methods available for objects of O's class. When it is found, the method is executed; if not, an error method is called. This ensures that only valid code meant for objects of the given class is run.

This mechanism also makes it possible to have the same message sent to objects of different classes, where different methods of the same name are called. This property is called *polypmorphism*. It saves inventing new names (e.g., print_integer, print_string, etc.) for the same function (print an object) applied to objects of different classes. In particular, it enables you to send the same messages that can be sent to objects of class C, to objects of all subclasses of C that inherited the methods from C.

Methods for Individuals

Instance methods of a class have the same effect on every object of that class, which is very desirable for consistency. But sometimes you want to have different objects of the same class to react differently to the same message. This is very useful when you want to program, for example, a finite state machine. There are a number of states the machine can enter and a number of possible events that can occur. This can be implemented by modelling each state as an object of class `state`, where each object is supposed to react specifically to a message, such as "a key was pressed." This behaviour could be achieved by means of a reaction table, but there is a more elegant way which is an evolution of the concept of message passing. OOF provides you with individual *instance methods* (`iim`). A default `iim` for all instances (objects) of the class is defined when the class is declared. For every individual object, a specific `iim` can now be declared which will be the individual response to still the same message. OOF code for this example might look like in Figure Two.

Shallow Objects

In almost every language, there is the notion of pointers. A pointer is not an object itself but keeps only a reference to an object. In Forth, every address can be interpreted as a pointer to a data field. The syntax for dealing with pointers can become very confusing (just think of C pointer puzzles) and error prone. I abandoned the idea of a class pointer for these reasons. Instead, Smalltalk inspired me to define *shallow objects*. They are disguised as normal objects of a class, but only bear a reference to another object. *They behave exactly as if they were the objects they keep the reference to*; you do not have to worry about their shallow nature. An example is a shallow object of class `state` (see above) that represents the active state of the state machine. It is created and handled as shown in Figure Three.

Now, in order to send the message `keypressed` to the momentarily active state, you write:

```
active keypressed
```

which, at this point, sends the message `keypressed` to the state object `idle`. Note that you need not perform some sort of pointer-indirection-operation, the shallow object `active` automatically pushes the referenced state-object `idle` onto the stack. The `iim keypressed` of `idle` now can be extended, as shown in Figure Four.

Looping

The control structures are quite the same as in Forth. The `do ... leave ... loop`, however, was modified: it still takes limit and index as parameters, but they must be given in the form of local integer variables. This offers you the opportunity to name the "functions" that access the index and limit of the loop (the former `i` and `j`) the way you like (and spares the implementation to clobber the return stack with looping

Figure Six. Header of an object.

name	name of object, terminated by 0, followed by a count-byte
link	object number of next object in list, e.g., vocabulary, etc.
time	object number of next object in list of all objects
owner	object number of owner-object, e.g., owner of instance object
module	object number of module-object = vocabulary + source file
module-offset	offset in source file
flags	e.g., shallow/deep object, private/public, allocated/deleted
size	body size of object
class	object number of the class the object belongs to
self	object number of the object itself
body	object number of the object that holds the object's body
offset	offset within the body

information). The example in Figure Five shows how to use this feature.

Implementation

OOF is not implemented by extending an existing Forth. It is not based on clever use of vocabularies and `create ... does>` constructs. I did start defining it that way, since it is the first and obvious way to any Forth programmer. But it soon turned out that, if I used a standard Forth as a basis to program OOF, the underlying Forth would either not be in use any more when running OOF or it would induce intolerable speed penalties. So the variety of new concepts forced an entirely new kernel for OOF. In the process, some wrinkles in Forth were ironed out by strictly adhering to the paradigm of object orientation. As with any Forth, the kernel consists of some assembler primitives for arithmetic, I/O, and special kernel functions—comprising the virtual machine (VM) of OOF—and the lion's share of the kernel is written in OOF itself. This OOF source code is translated by a metacompiler into the kernel's threaded code.

To make porting as fast and easy as possible, C source versions of all VM functions exist. Existing programs (e.g., image-processing libraries) written in C or other languages can easily be linked to OOF. The metacompiler itself is written in C as well. So, in order to port OOF to another host, all you need for the beginning is a C compiler for that machine.

In the following sections, I will describe in detail the structure of objects, the concept of object storage, and the consequences concerning access by the virtual machine of objects, the stack, and the execution of secondaries.

Handle with Care

Each object consists of a header and a body. The structure of the header is shown in Figure Six. It contains various information about the context and nature of an object, as well as information for debugging and the source of the definition of the object. The body of the object that contains its data is located anywhere else in memory, in a contiguous memory block. The body of a compound object that is built of instance objects consists of the bodies of its instance objects. As an example, the body of an `image` object is shown in Figure Seven.

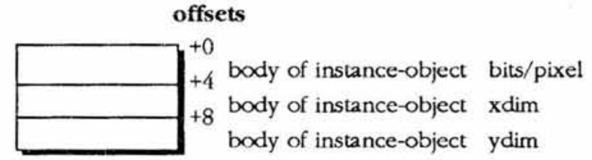
Objects may not be accessed directly by an address, only via (16-bit) object numbers (sometimes called *handles*). An object number can be converted into a memory address via the Object Table, an array that holds the memory address of every object that exists in the system. The address of an object is determined by using the object number as an index into the Object Table. Thus, it is possible to relocate an object without having to change every reference to this object; only the entry in the Object Table must be updated. This is very important in an object-oriented system, where objects are constantly created and deleted at run time, causing the need for garbage collection and relocation of objects.

Every object has a unique object number. The corresponding address from the Object Table points to the header of the object, and the nature and state of the object can now be deduced. However, since the body of the object can be stored anywhere else in memory, we need more information in the object's description: a further object number gives the address of the memory block where the body is located. An offset within the block must be added to this address to arrive at the complete body address of the object. This process is shown in Figure Eight. Thus, a complete description of an object consists of a reference to the header and a reference to the body of the object.

This concept has great impact on the format of stack entries. It makes sense to push onto the parameter stack not only an address, as in Forth, but a complete object descriptor. In order to speed up message passing, the object number of the class of the object is added. When primitive arithmetic or logical functions are executed, it would cause an enormous overhead when a new object would have to be created each time a result is returned. So, in the stack entry, there is a `value` field that can hold the result of operations on basic classes, such as integer, character, or even float. The complete format of a stack entry on the parameter stack is shown in Figure Nine.

Because the description of an object is divided into a reference to the header and a reference to the body, one

Figure Seven. Body of an image object.



header can be used to describe many bodies. This is the case for instance objects. E.g., every time an `image` object is created, its body contains the bodies of its three instance objects: `bits/pixel`, `xdim`, and `ydim` (Figure Seven). But a new header is created only for the new image object—there is no need to create a new header for each instance object. The one header that was built during the definition of the class `image` for each instance object, describes the nature of the instance objects completely. `OUT` and `LOCAL` objects of a secondary are treated similarily: every time a secondary is entered, only space for the bodies of these objects is allocated on the parameter stack, no new header is created. So, in most cases, no new header is constructed for new objects. This saves considerable amounts of both time and memory space.

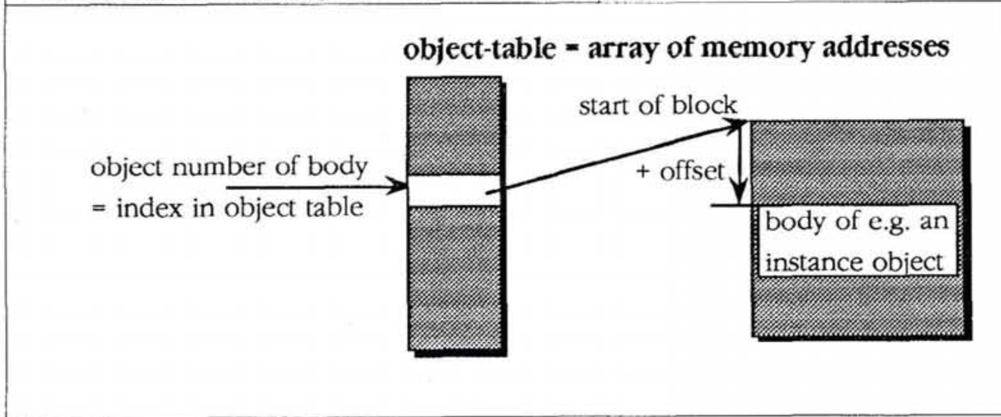


Figure Eight. Accessing the body of an object.

Additionally, it turns out, access to the body of an object is accelerated a great deal because there is only one mechanism to arrive at the body address. No testing of flags or considering of special circumstances at run time—which takes longer than it takes to actually access the Object Table—are necessary. The latter is especially important for the primitives, which should be as fast and efficient as possible.

Once again, the principle of simplification by generalization proves useful. Here, it saves space and time when creating new objects, and simplifies and, therefore, speeds up access to the body of an object. The next paragraph describes how the appropriate stack entries are composed according to the special nature of each object.

The Virtual Machine

The virtual machine (VM) of OOF consists of a number of Body Evaluation Codes (BECs), the inner interpreter, the primitives, internal registers, and the available RAM. It is the machine-dependent part of an OOF system.

A BEC is a piece of assembler code that performs a basic function, such as pushing an object descriptor onto the stack, sending a message to the object on TOS, and calling or returning from a secondary. They are the counterparts of Forth's CFA primitives.

The inner interpreter works similar to most Forths' inner interpreters: a register of the virtual machine called `OOFPC` points inside the body of a secondary, where the next entry shall be interpreted. Each entry in a secondary consists of the object number of a BEC followed by parameters for the BEC. In order to speed up the interpretation, OOF compiles references to the codes that evaluate the entries of the body, directly into the body rather than just storing a reference to the object. Everything that is known at compile time about an object to be compiled is stored as the appropriate BEC and the necessary parameters; so that, at run time, a BEC does not have to test flags or search for its parameters somewhere else. Since lots of kilobytes of memory are no big deal anymore, space is no problem in most development systems. OOF makes incremental development and compilation possible, so compilation speed is not critical. The emphasis in the design of OOF's kernel is on execution speed, which is always a critical issue in object-oriented systems. So, in OOF compilation takes a bit longer and the code size grows, but execution is sped up a great deal.

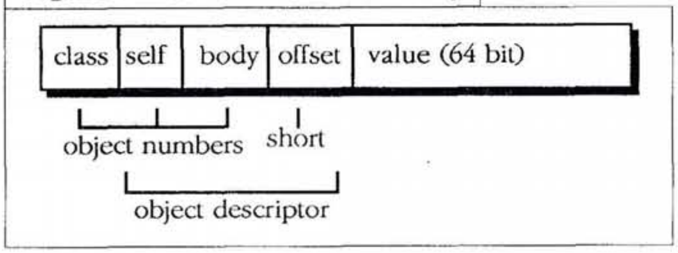


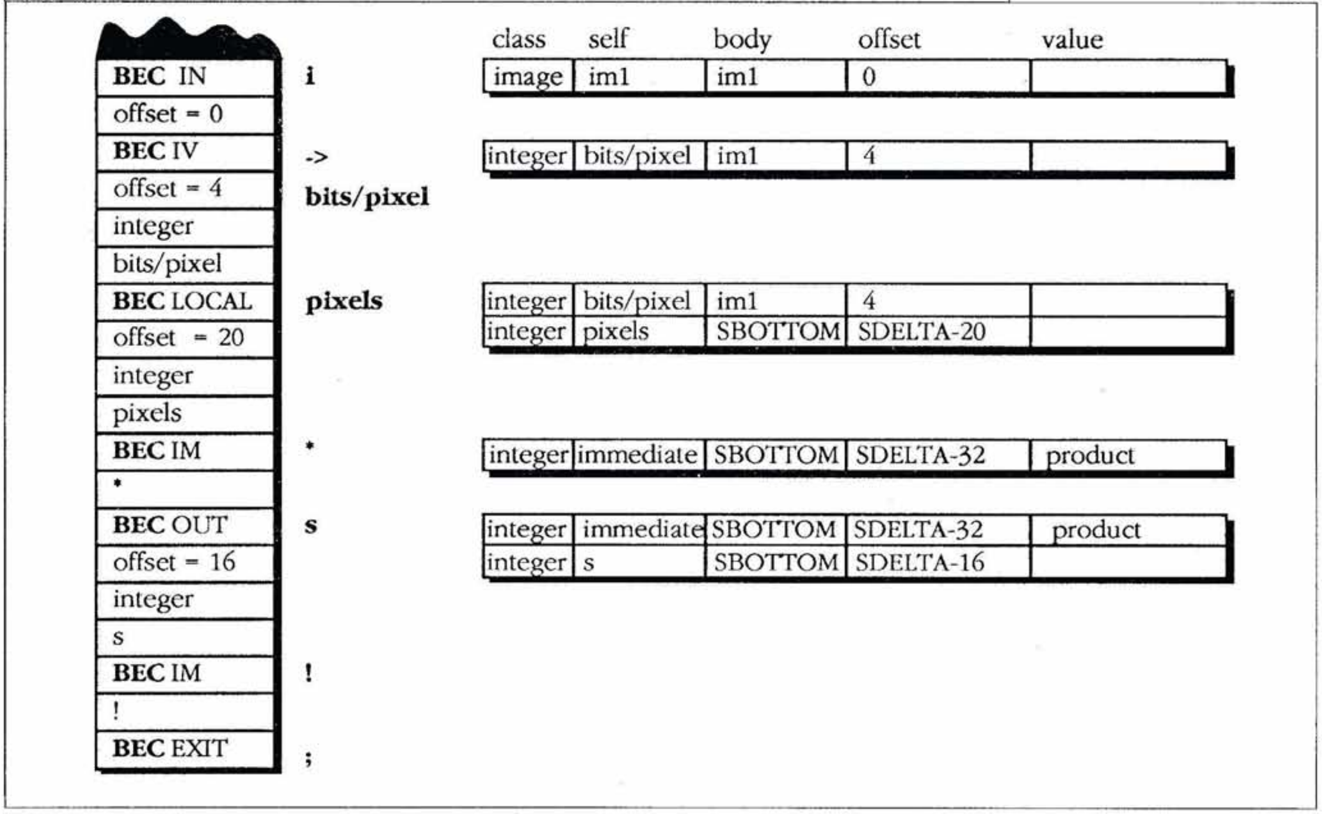
Figure Nine. Format of a stack entry.

One by one, each entry of a secondary is evaluated by calling the BEC that the `OOFPC` points to. At the end of every BEC, the `OOFPC` is set to the next entry, the processor jumps back to the inner interpreter, and the next entry in the body of the secondary is evaluated.

The execution of a secondary shall now be explained in detail. When a secondary is entered, the VM registers `SDELTA`, `SFRAME`, and `OOFPC`, and the object number of the secondary are saved on the return stack. Then `SFRAME` is set below the first input parameter within the parameter stack; now the stack frame starts with the stack entries of the input objects for the secondary. `SDELTA` is calculated as `SFRAME` – `SBOTTOM` (bottom of stack), and space for the bodies of the `OUT` and `LOCAL` objects is reserved on the stack by decrementing TOS. After `OOFPC` is set to the first entry in the body, the inner interpreter is ready to interpret the secondary. Figure Eleven shows the contents of the parameter stack at that time.

To explain the functions of the BECs, the evaluation of a sample part of the secondary that was called when the message `size` was sent to the global image object `im1`, shall now be traced. Figure Ten shows both a sample of the body of the method `size` and the appropriate contents of the parameter stack after the execution of each BEC.

Figure Ten. Body of a secondary and parameter stack during execution (extract from the method size sent to the global object im1).



BEC `IN` takes the offset parameter as an offset into the stack frame and copies the stack entry at that location to TOS, effectively pushing the current input parameter `im1` onto the parameter stack. This is an example of the behaviour of shallow objects: they do not appear on the stack themselves, only the object they refer to.

BEC `IV` manipulates the stack entry on TOS as follows: it adds the offset to the `offset` on TOS, replaces `class` with `integer`, and `self` with `bits/pixel`. The body object remains the same, as explained above. Now the instance object `bits/pixel` of the input object is on TOS.

In order to push `pixels` onto the stack, BEC `LOCAL` creates a new stack entry. Then it takes the parameter offset as an offset into the stack frame, subtracts it from the contents of the VM register `SDELTA` (equal to the stack frame – bottom of stack) and places the result as the `offset` into the new stack entry. The pseudo-object `SBOTTOM` (bottom of parameter stack) becomes the new `body`, `class` is set to `integer`, and `self` is set to `pixels`.

The parameter of BEC `IM` is a token that represents the message *. The message is sent to the object on TOS, which is now `pixels`. In the list of instance methods of the class of the object on TOS (here: `integer`), a method is searched that matches this token. In order to speed up the search, this list is 32-fold hashed. When the method is found, it is called; if not, an error message that is guaranteed to be understood by all objects—since it is defined in the kernel—is sent to the object.

The primitive for integer multiplication takes the two integer objects off the stack and pushes an integer object with the product in the value field of the stack entry. `Self` is set to the pseudo-object-number `immediate`, indicating that there is no valid header available for this object.

The `OUT` object `s` is pushed by BEC `OUT` the same way `pixels` was pushed.

The result of the multiplication is stored in `s` by the method that was found when BEC `IM` sent the message `!` to `s`.

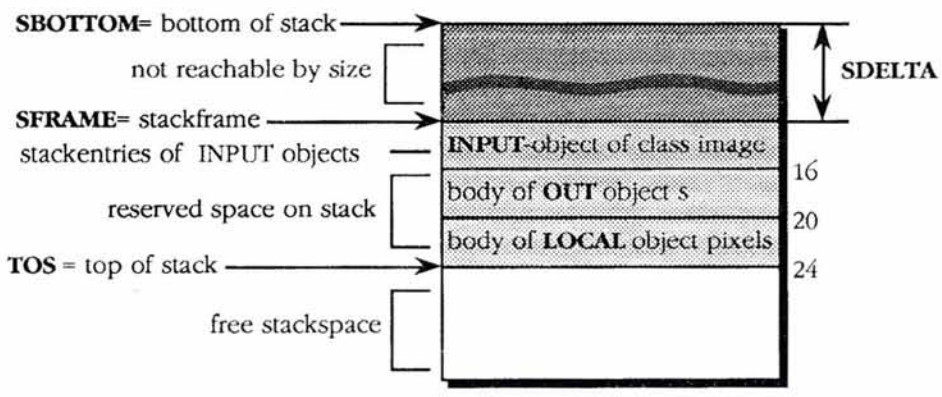
Finally, the secondary `size` is exited by BEC `EXIT`, which cleans the parameter stack by setting TOS to `SFRAME`, effectively freeing the space of `OUT` and `LOCAL` bodies in the stack frame and taking the `IN` objects off the stack. Then it pushes the `OUT` objects (here: `s`) onto the stack, restores the VM registers `SFRAME`, `SDELTA`, and `OOFPC` from the return stack, and resumes where `size` was called.

Experiences with OOF and Future Work

OOF was utilized to write a toolbox for graphical user interfaces for image workstations. Since the OOF kernel will be finished real soon now, we still have to gain experience about how the system behaves in terms of speed. However, early experiences about the impact on the style, clarity, and ease of writing programs in OOF are promising.

A very important lesson is that the way you tackle problems changes when using OOF. You no longer think about some data structure in the beginning and then write lots of code manipulating it independently. After having analysed your problem, you try to build a hierarchy of classes that can represent the solution. Here you use the well-known techniques of factoring and decomposition, which are

Figure Eleven. Parameter stack at entry of secondary size.



already recognised good style in Forth and elsewhere. Data structures and methods working on the data are tightly coupled now. When you write a function in OOF, first you have to consider to which object you will be sending a message to do the job.

Since OOF does a lot for you in terms of factoring or deciding what to do in special cases, which is done by inheritance, polymorphism, and message passing, code tends to be tighter and more to the point. You have more control over what is going on in your program and you don't get lost in an unordered heap of data and words. Last but not least, it is more fun!

Some words concerning standards: we have worked with more-or-less standard Forths and were less than happy with the environments and the support for non-trivial programs written by more than one engineer. OOF might not be suitable for tiny target applications or not fast enough for real-time applications (which is still to be decided, there is enough room for optimization). However, OOF points out how to tackle problems differently but still in the good ol' Forth style of writing and testing small chunks of code incrementally and interactively. It supports the programmer by providing the integral ordering mechnisms of object-oriented languages and by adding security—by means of strong type checking and local parameters—to the advantages of Forth. Thus, OOF could show a way towards a modernized, extended, supportive Forth living side by side with the current standard, minimalistic, compact Forth.

The future will see an OOF system running all described features, with optimized code for the kernel, more classes, ported versions on hosts such as PCs, Macs, and Unix machines, and a full debugging environment to make life even easier. We will run laboratory work for image processing and protoyping of medical image workstations using OOF. Any comments, annotations or additions are welcome. Stay tuned.

We hope our work will not be misused for military purposes. We will not take part in any military projects.

Markus Dahm received his Dipl. Ing. (electrical engineering) in 1987 at RWTH Aachen, University of Technology, Aachen, Germany; abd his M.Sc. (Computer Science) in 1988 at Imperial College, London, U.K. He has been a research assistant since 1989 at Lehrstuhl fuer Meßtechnik, RWTH Aachen, DIBA-project, working on user interfaces for medical imaging workstations. He can reached at the following:

Lehrstuhl fuer Messtechnik, RWTH Aachen, Templergraben 55, D-5100 Aachen, Germany. Phone: +241 - 80 78 64. Fax: +241 - 80 78 71.
E-mail: SE@DACTH51.BITNET

$ Contest Announcement $

Call for Papers!

Forth Dimensions is sponsoring a contest for articles about

"Forth on a Grand Scale"

Write about large-scale Forth applications, systems, or ...

"This theme applies equally to projects requiring multiple programmers, and to applications or systems consisting of large amounts of code and/or of significant complexity." ("Editorial," *FD* XIII/6) Papers will be refereed.

Mail a hard copy and a diskette (Macintosh 800K or PC preferred) to the:

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

Cash awards to authors:

1st place:	***$500***
2nd place:	***$250***
3rd place:	***$100***

Deadline for contest entries is August 3, 1992.

FIG
MAIL ORDER FORM

HOW TO USE THIS FORM: Please enter your order on the back page of this form and send with your payment to the Forth Interest Group.
Most items list three different price categories: USA, Canada, and Mexico / Other countries via surface mail / Other countries via air mail
Note: Where only two prices are listed, surface mail is not available.

"Were Sure You Wanted To Know..."

***Forth Dimensions*, Article Reference** 151 - $4/5
An index of Forth articles, by keyword, from *Forth Dimensions* Volumes 1–12 (1978–91).

FORML, Article Reference 152 - $4/5
An index of Forth articles by keyword, author, and date from the FORML Conference Proceedings (1980–89).

FORTH DIMENSIONS BACK VOLUMES

A volume consists of the six issues from the volume year (May–April)

Volume 1 Forth Dimensions (1979–80) 101 - $15/16/18
Introduction to FIG, threaded code, TO variables. fig-Forth.

Volume 2 Forth Dimensions (1980–81) 102 - $15/16/18
Recursion, file naming, Towers of Hanoi, CASE contest, input number wordset, 2nd FORML report, FORGET, VIEW.

Volume 3 Forth Dimensions (1981–82) 103 - $15/16/18
Forth-79 Standard, Stacks, HEX, database, music, memory management, high-level interrupts, string stack, BASIC compiler, recursion, 8080 assembler.

Volume 4 Forth Dimensions (1982–83) 104 - $15/16/18
Fixed-point trig., fixed-point square root, fractional arithmetic, CORDIC algorithm, interrupts, stepper-motor control, source-screen documentation tools, recursion, recursive decompiler, file systems, quick text formatter, ROMmable Forth, indexer, Forth-83 Standard, teaching Forth, algebraic expression evaluator.

Volume 5 Forth Dimensions (1983–84) 105 - $15/16/18
Computer graphics, 3D animation, double-precision math words, overlays, recursive sort, a simple multi-tasker, metacompilation, voice output, number utility, menu-driven software, vocabulary tutorial, vectorerd execution, data acquisition, fixed-point logarithms, Quicksort, fixed-point square root.

Volume 6 Forth Dimensions (1984–85) 106 - $15/16/18
Interactive editors, anonymous variables, list handling, integer solutions, control structures, debugging techniques, recursion, semiphores, simple I/O words, Quicksort, high-level packet communications, China FORML.

Volume 7 Forth Dimensions (1985–86) 107 - $20/22/25
Generic sort, Forth spreadsheet, control structures, psuedo-interrupts, number editing, Atari Forth, pretty printing, code modules, universal stack word, polynomial evaluation, F83 strings.

Volume 8 Forth Dimensions (1986–87) 108 - $20/22/25
Interrupt-driven serial input, data-base functions, TI 99/A, XMODEM, on-line documentation, dual-CFAs, random numbers, arrays, file query, Batcher's sort, screenless Forth, classes in Forth, Bresenham line-drawing algorithm, unsigned division, DOS file I/O.

Volume 9 Forth Dimensions (1987–88) 109 - $20/22/25
Fractal landscapes, stack error checking, perpetual date routines, headless compiler, execution security, ANS-Forth meeting, computer-aided instruction, local variables, transcendental functions, education, relocatable Forth for 68000.

Volume 10 Forth Dimensions (1988–89) 110 - $20/22/25
dBase file access, string handling, local variables, data structures, object-oriented Forth, linear automata, standalone applications, 8250 drivers, serial data compression.

Volume 11 Forth Dimensions (1989–90) 111 - $20/22/25
Local variables, graphic filling algorithms, 80286 extended memory, expert systems, quaternion rotation calculation, multiprocessor Forth, double-entry bookkeeping, binary table search, phase-angle differential analyzer, sort contest.

Volume 12 Forth Dimensions (1990–91) 112 - $20/22/25
Floored division, stack variables, embedded control, Atari Forth, optimizing compiler, dynamic memory allocation, smart RAM, extended-precision math, interrupt handling, neural nets, Soviet Forth, arrays, metacompilation.

FORML CONFERENCE PROCEEDINGS

FORML (Forth Modification Laboratory) is an educational forum for sharing and discussing new or unproven proposals intended to benefit Forth, and is an educational forum for discussion of the technical aspects of applications in Forth. Proceedings are a compilation of the papers and abstracts presented at the annual conference. FORML is part of the Forth Interest Group.

1980 FORML PROCEEDINGS 310 - $30/31/40
Address binding, dynamic memory allocation, local variables, concurrency, binary absolute & relocatable loader, LISP, how to manage Forth projects, n-level file system, documenting Forth, Forth structures, Forth strings.

1981 FORML PROCEEDINGS 311 - $45/48/55
CODE-less Forth machine, quadruple-precision arithmetic, overlays, executable vocabulary stack, data typing in Forth, vectored data structures, using Forth in a classroom, pyramid files, BASIC, LOGO, automatic cueing language for multimedia, NEXOS—a ROM-based multitasking operating system.

1982 FORML PROCEEDINGS 312 - $30/31/40
Rockwell Forth processor, virtual execution, 32-bit Forth, ONLY for vocabularies, non-IMMEDIATE looping words, number-input wordset, I/O vectoring, recursive data structures, programmable-logic compiler.

1983 FORML PROCEEDINGS 313 - $30/32/40
Non-Von Neuman machines, Forth instruction set, Chinese Forth, F83, compiler & interpreter co-routines, log & exponential function, rational arithmetic, transcendental functions in variable-precision Forth, portable file-system interface, Forth coding conventions, expert systems.

1984 FORML PROCEEDINGS 314 - $30/33/40
Forth expert systems, consequent-reasoning inference engine, Zen floating point, portable graphics wordset, 32-bit Forth, HP71B Forth, NEON—object-oriented programming, decompiler design, arrays and stack variables.

1985 FORML PROCEEDINGS 315 - $30/32/40
Threaded binary trees, natural language parsing, small learning expert system, LISP, LOGO in Forth, Prolog interpreter, BNF parser in Forth, formal rules for phrasing, Forth coding conventions, fast high-level floating point, Forth component library, Forth & artificial intelligence, electrical network analysis, event-driven multitasking.

1986 FORML PROCEEDINGS 316 - $30/32/40
Threading techniques, Prolog, VLSI Forth microprocessor, natural-language interface, expert system shell, inference engine, multiple-inheritance system, automatic programming environment.

1987 FORML PROCEEDINGS 317 - $40/43/50
Includes papers from '87 euroFORML Conference. 32-bit Forth, neural networks, control structures, AI, optimizing compilers, hypertext, field and record structures, CAD command language, object-oriented lists, trainable neural nets, expert systems.

1988 FORML PROCEEDINGS 318 - $24/25/34
Human interfaces, simple robotics kernel, MODUL Forth, language topics, hardware, Wil's workings & Ting's philosophy, Forth hardware applications, ANS Forth session, future of Forth in AI applications.

1988 AUSTRALIAN PROCEEDINGS 380 - $24/25/34
Proceedings from the first Australian Forth Symposium, held May 1988 at the University of Technology in Sydney. Subjects include training, parallel processing, programmable controllers, Prolog, simulations, and applications.

1989 FORML PROCEEDINGS 319 - $40/43/50
Includes papers from '89 euroFORML. Pascal to Forth, extensible optimizer for compiling, 3D measurement with object-oriented Forth, CRC polynomials, F-PC, Harris C cross-compiler, modular approach to robotic control, RTX recompiler for on-line maintenance, modules, trainable neural nets.

1990 FORML PROCEEDINGS 320 - $40/43/50
Forth in industry, communications monitor, 6805 development. 3-key keyboard, documentation techniques, object-oriented programming, simplest Forth decompiler, error recovery, stack operations, process control event management, control structure analysis, systems design course, group theory using Forth.

BOOKS ABOUT FORTH

ALL ABOUT FORTH, 3rd ed., June 1990, Glen B. Haydon 201 - $90/92/105
Annotated glossary of most Forth words in common usage, including Forth-79, Forth-83, F-PC, MVP-Forth. Implementation examples in high-level Forth and/or 8086/88 assembler. Useful commentary given for each entry.

THE COMPLETE FORTH, Alan Winfield 210 - $14/15/19
A comprehensive introduction, including problems with answers (Forth-79).

eFORTH IMPLEMENTATION GUIDE, C.H. Ting 215 - $25/26/35
eForth is the name of a Forth model designed to be portable to a large number of the newer, more powerful processors available now and becoming available in the near future. (w/disk)

F83 SOURCE, Henry Laxen & Michael Perry 217 - $20/21/30
A complete listing of F83, including source and shadow screens. Includes introduction on getting started.

FORTH: A TEXT AND REFERENCE 219 - $31/32/41
Mahlon G. Kelly & Nicholas Spies
A textbook approach to Forth, with comprehensive references to MMS-FORTH and the '79 and '83 Forth standards.

THE FORTH COURSE, Richard E. Haskell 225 - $25/26/35
This set of 11 lessons, called the Forth Course, is designed to make it easy for you to learn Forth. The material was developed over several years of teaching Forth as part of a senior/graduate course in design of embedded software computer systems at Oakland University in Rochester, Michigan. (w/disk)

FORTH ENCYCLOPEDIA, Mitch Derick & Linda Baker 220 - $30/32/40
A detailed look at each fig-Forth instruction.

FORTH NOTEBOOK, Dr. C.H. Ting 232 - $25/26/35
Good examples and applications. Great learning aid. poly-FORTH is the dialect used. Some conversion advice is included. Code is well documented.

FORTH NOTEBOOK II, Dr. C.H. Ting 232a - $25/26/35
Collection of research papers on various topics, such as image processing, parallel processing, and miscellaneous applications.

INSIDE F-83, Dr. C.H. Ting 235 - $25/26/35
Invaluable for those using F-83.

LIBRARY OF FORTH ROUTINES AND UTILITIES,
James D. Terry 237 - $23/25/35
Comprehensive collection of professional quality computer code for Forth; offers routines that can be put to use in almost any Forth application, including expert systems and natural-language interfaces.

OBJECT ORIENTED FORTH, Dick Pountain 242 - $28/29/34
Implementation of data structures. First book to make object-oriented programming available to users of even very small home computers.

STACK COMPUTERS, THE NEW WAVE 244 - $62/65/72
Philip J. Koopman, Jr. (hardcover only)
Presents an alternative to Complex Instruction Set Computers (CISC) and Reduced Instruction Set Computers (RISC) by showing the strengths and weaknesses of stack machines (hard-cover only).

STARTING FORTH (2nd ed.), Leo Brodie 245 - $29/30/38
In this edition of *Starting Forth*—the most popular and complete introduction to Forth—syntax has been expanded to include the Forth-83 Standard.

TOOLBOOK OF FORTH, V1 267 - $23/25/35
(Dr.Dobb's) Edited by Marlin Ouverson
Expanded and revised versions of the best Forth articles collected in the pages of Dr.Dobb's Journal.
TOOLBOOK, V1 with DISK (MS-DOS) 267a - $40/42/50

TOOLBOOK OF FORTH, V2, (Dr. Dobb's) 268 - $30/32/40
Complete anthology of FORTH programminmg techniques and developments, picks up where V.1 left off. Topics include programming windows, extended control structures, design of a Forth target compiler, and more.
TOOLBOOK, V2 with DISK (MS-DOS) 268a - $46/48/56

WRITE YOUR OWN PROGRAMMING LANGUAGE USING C++,
Norman Smith 270 - $15/16/18
This book is about an application language. More specifically, it is about how to write your own custom application language. The book contains the tools necessary to begin the process and a complete sample language implementation. [Guess what language!] Includes disk with complete source. **NEW**

REFERENCE

FORTH-83 STANDARD 305 - $15/16/18
Authoritative description of Forth-83 Standard. For reference, not instruction.

SYSTEMS GUIDE TO fig-FORTH 308 - $25/28/30
C. H. Ting (2nd ed., 1989)
How's and why's of the fig-Forth Model by Bill Ragsdale, internal structure of fig-Forth system.

BIBLIOGRAPHY OF FORTH REFERENCES 340 - $18/19/25
(3rd ed., January 1987)
Over 1900 references to Forth articles throughout computer literature.

F-PC USERS MANUAL (2nd ed., V3.5) 350 - $20/21/27
Users manual to the public-domain Forth system optimized for IBM PC/XT/AT computers. A fat, fast system with many tools.

F-PC TECHNICAL REFERENCE MANUAL 351 - $30/32/40
A must if you need to know the inner workings of F-PC.

MORE ON FORTH ENGINES

Volume 10 January 1989 810 - $15/16/18
RTX reprints from 1988 Rochester Forth Conference, object-oriented cmForth, lesser Forth engines.
Volume 11 July 1989 811 - $15/16/18
RTX supplement to *Footsteps in an Empty Valley*, SC32, 32-bit Forth engine, RTX interrupts utility.
Volume 12 April 1990 812 - $15/16/18
ShBoom Chip architecture and instructions, Neural Computing Module NCM3232, pigForth, binary radix sort on 80286, 68010, and RTX2000.
Volume 13 October 1990 813 - $15/16/18
PALs of the RTX2000 Mini-BEE, EBForth, AZForth, RTX-2101, 8086 eForth, 8051 eForth.
Volume 14 814 - $15/16/18
RTX Pocket-Scope, eForth for muP20, ShBoom, eForth for CP/M & Z80, XMODEM for eForth.
Volume 15 815 - $15/16/18
Moore: New CAD System for Chip Design, A portrait of the P20; Rible: QS1 Forth Processor, QS2, RISCing it all; P20 eForth Software Simulator/Debugger.

MISCELLANEOUS

T-SHIRT "May the Forth Be With You" 601 - $12/13/15
(Specify size: Small, Medium, Large, Extra-Large on order form)
White design on a dark blue shirt.
POSTER (*BYTE* cover) 602 - $5/6/7
FORTH-83 HANDY REFERENCE CARD 683 - free

DISKS: Contributions from the Forth Community

The "Contributions from the Forth Community" disk library contains author-submitted donations, generally including source, for a variety of computers & disk formats. Each file is determined by the author as public domain, shareware, or use with some restrictions. This library does not contain "For Sale" applications. *To submit your own contributions, send them to the FIG Publications Committee.*

Prices: Each item below comes on one or more disks, indicated in parentheses after the item number. The price of your order is $6/9 per disk, or $25/37 for any five disks.

FLOAT4th.BLK V1.4 Robert L. Smith — C001 - (1)
Software floating-point for fig-, poly-, 79-Std., 83-Std. Forths. IEEE short 32-bit, four standard functions, square root and log. **IBM.**

Games in Forth — C002 - (1)
Misc. games, Go, TETRA, Life... Source. **IBM**

A Forth Spreadsheet V2, Craig Lindley — C003 - (1)
This model spreadsheet first appeared in *Forth Dimensions* VII, 1-2. Those issues contain docs & source. **IBM**

Automatic Structure Charts V3, Kim Harris — C004 - (1)
Tools for analysis of large Forth programs, first presented at FORML conference. Full source; docs incl. in 1985 FORML Proceedings. **IBM**

A Simple Inference Engine V4, Martin Tracy — C005 - (1)
Based on inf. engine in Winston & Horn's book on LISP, takes you from pattern variables to complete unification algorithm, with running commentary on Forth philosophy & style. Incl. source. **IBM**

The Math Box V6, Nathaniel Grossman — C006 - (1)
Routines by foremost math author in Forth. Extended double-precision arithmetic, complete 32-bit fixed-point math, & auto-ranging text. Incl. graphics. Utilities for rapid polynomial evaluation, continued fractions & Monte Carlo factorization. Incl. source & docs. **IBM**

AstroForth & AstroOKO Demos, I.R. Agumirsian — C007 - (1)
AstroForth is the 83-Std. Russian version of Forth. Incl. window interface, full-screen editor, dynamic assembler & a great demo. AstroOKO, an astronavigation system in AstroForth, calculates sky position of several objects from different earth positions. Demos only. **IBM**

Forth List Handler V1, Martin Tracy — C008 - (1)
List primitives extend Forth to provide a flexible, high-speed environment for AI. Incl. ELISA and Winston & Horn's micro-LISP as examples. Incl. source & docs. **IBM**

8051 Embedded Forth, William Payne — C050 - (4)
8051 ROMmable Forth operating system. 8086-to-8051 target compiler. Incl. source. Docs are in the book *Embedded Controller Forth for the 8051 Family.* **IBM**

F83 V2.01, Mike Perry & Henry Laxen — C100 - (1)
The newest version, ported to a variety of machines. Editor, assembler, decompiler, metacompiler. Source and shadow screens. Manual available separately (items 217 & 235). Base for other F83 applications. **IBM, 83.**

F-PC V3.53, Tom Zimmer — C200 - (5)
A full Forth system with pull-down menus, sequential files, editor, forward assembler, metacompiler, floating point. Complete source and help files. Manual for V3.5 available separately (items 350 & 351). Base for other F-PC applications. Req. hard disk. **IBM, 83.**

F-PC TEACH V3.5, Lessons 0–7 Jack Brown — C201a - (2)
Forth classroom on disk. First seven lessons on learning Forth, from Jack Brown of B.C. Institute of Technology. **IBM, F-PC.**

VP-Planner Float for F-PC, V1.01 Jack Brown — C202 - (1)
Software floating-point engine behind the VP-Planner spreadsheet. 80-bit (temporary-real) routines with transcendental functions, number I/O support, vectors to support numeric co-processor overlay & user NAN checking. **IBM, F-PC.**

F-PC Graphics V4.2f, Mark Smiley — C203a - (3)
The latest versions of new graphics routines, including CGA, EGA, and VGA suppport, with numerous improvements over earlier versions created or supported by Mark Smiley. **IBM, F-PC.**

PocketForth V1.4, Chris Heilman — C300 - (1)
Smallest complete Forth for the Mac. Access to all Mac functions, files, graphics, floating point, macros, create standalone applications and DAs. Based on fig & *Starting Forth.* Incl. source and manual. **MAC**

Yerkes Forth V3.6 — C350 - (2)
Complete object-oriented Forth for the Mac. Object access to all Mac functions, files, graphics, floating point, macros, create standalone applications. Incl. source, tutorial, assembler & manual. **MAC,** System 7.01 Compatable.

JLISP V1.0, Nick Didkovsky — C401 - (1)
LISP interpreter invoked from Amiga JForth. The nucleus of the interpreter is the result of Martin Tracy's work. Extended to allow the LISP interpreter to link to and execute JForth words. It can communicate with JForth's ODE (Object-Development Environment). **AMIGA,** 83.

Pygmy V1.3, Frank Sergeant — C500 - (1)
A lean, fast Forth with full source code. Incl. full-screen editor, assembler and metacompiler. Up to 15 files open at a time. **IBM.**

KForth, Guy Kelly — C600 - (3)
A full Forth system with windows, mouse, drawing and modem packages. Incl. source & docs. **IBM, 83.**

ForST, John Redmond — C700 - (1)
Forth for the Atari ST. Incl. source & docs. **Atari ST.**

Mops V2.0, Michael Hore — C710 - (1)
Close cousin to Yerkes and Neon. Very fast, compiles subroutine-threaded & native code. Object oriented. Uses F-P co-processor if present. Full access to Mac toolbox & system. Supports System 7 (e.g., AppleEvents). Incl. assembler, docs & source. **MAC**

BBL & Abundance, Roedy Green — C800 - (4)
BBL public-domain, 32-bit Forth with extensive support of DOS, meticulously optimized for execution speed. Abundance is a public-domain database language written in BBL. Req. hard disk. Incl. source & docs. **IBM HD hard disk reequired**

ACM - SIGFORTH

The ACM SIGForth Newsletter is published quarterly by the Association of Computing Machinery, Inc. SIGForth's focus is on the development and refinement of concepts, methods, and techniques needed by Forth professionals.

Volume 1 Spring 1989 — 900 - $6/7/9
F-PC, glossary utility, Euroforth, SIGForth '89 Workshop summary (real-time software engineering), Intel 80x8x.

Volume 1 Summer 1989 — 901 - $6/7/9
Metacompiler in cmForth, Forth exception handler, string case statement for UF/Forth.

Volume 1, #3 Fall 1989 — 902 - $6/7/9
1802 simulator, tutorial on multiple threaded vocabularies.

Volume 1, #4 Winter 1989 — 903 - $6/7/9
Stack frames, duals: an alternative to variables, PocketForth.

Volume 2, #2 December 1990 — 904 - $6/7/9
BNF Parser, abstracts 1990 Rochester conf., F-PC Teach.

Volume 2, #3 — 905 - $6/7/9
Tethered Forth model, abstracts 1990 SIGForth conf.

Volume 2, #4 — 906 - $6/7/9
Target-meta-cross-: an engineer's viewpoint, single-instruction computer.

Volume 3, #1 Summer '91 — 907 - $6/7/9
Co-routines and recursion for tree balancing, convenient number handling.

Volume 3, #2 Fall '91 — 908 - $6/7/9
Postscript Issue, What is Postscript?, Forth in Postscript, Review: PS-Tutor.

1989 SIGForth Workshop Proceedings — 931 - $20/21/26
Software engineering, multitasking, interrupt-driven systems, object-oriented Forth, error recovery and control, virtual memory support, signal processing.

MEMBERSHIP IN THE FORTH INTEREST GROUP

The Forth Interest Group (FIG) is a world-wide, non-profit, member-supported organization with over 1,500 members and 40 chapters. Your membership includes a subscription to the bi-monthly magazine *Forth Dimensions*. FIG also offers its members an on-line data base, a large selection of Forth literature and other services.

Cost is $40.00 per year for U.S.A. & Canada surface mail; $46.00 Canada air mail; all other countries $52.00 per year. No sales tax, handling fee, or discount on membership.

When you join, your first issue will arrive in four to six weeks; subsequent issues will be mailed to you every other month as they are published—six issues in all. You will also receive a membership card and number which entitles you to a 10% discount on publications from FIG. Your member number will be required to receive the discount, so keep it handy.

Dues are not deductible as a charitable contribution for U.S. federal income tax purposes, but may be deductible as a business expense where applicable.

FORTH INTEREST GROUP

P.O. BOX 8231 SAN JOSE, CALIFORNIA 95155 408-277-0668 408-286-8988 (FAX)

Name ____________________

Company ____________________

Street ____________________

City ____________________

State/Prov. ____________ Zip ________

Country ____________________

Daytime phone ____________________

OFFICE USE ONLY

By______ Date__________ Type________

Shipped by__________ Date__________

UPS USPS XRDS

Wt.__________ Amt.__________

BO By__________ Date__________

Wt.__________ Amt.__________

Item #	Title	Qty.	Unit Price	Total
	*MEMBERSHIP →			**SEE BELOW**

☐ CHECK ENCLOSED (Payable to: Forth Interest Group)

☐ VISA ☐ MasterCard

Card Number ____________________

Expiration Date ____________________

Signature ____________________

($15.00 minimum on all VISA/MasterCard orders)

Sub-Total	
10% Member Discount Member # ________	
Sub-Total	
**Sales Tax (CA only)	
Mail Order Handling Fee	$3.00
*Membership in the Forth Interest Group ☐ New ☐ Renewal $40/46/52	
* Enclosed is $40/46/52 for 1 full year's dues. This includes $34/40/46 for *Forth Dimensions*.	

PAYMENT MUST ACCOMPANY ALL ORDERS

MAIL ORDERS
Send to:
Forth Interest Group
P.O. Box 8231
San Jose, CA 95155
U.S.A.

PHONE ORDERS
Call 408-277-0668 to place credit card orders or for customer service.
Hours: Mon.–Fri., 9 a.m.–5 p.m. PST.

PRICES
All orders must be prepaid. Prices are subject to change without notice. Credit card orders will be sent and billed at current prices. $15 minimum on charge orders. Checks must be in U.S. dollars, drawn on a U.S. bank. A $10 charge will be added for returned checks.

POSTAGE & HANDLING
Prices include shipping. The $3.00 handling fee is required with all orders.

SHIPPING TIME
Books in stock are shipped within seven days of receipt of the order. Please allow 4–6 weeks for out-of-stock books (deliveries in most cases will be much sooner).

**** CALIFORNIA SALES TAX BY COUNTY**
7.5%: Sonoma; **7.75%:** Fresno, Imperial, Inyo, Madera, Monterey, Orange, Riverside, Sacramento, San Benito, Santa Barbara, San Bernardino, and San Joaquin; **8.25%:** Alameda, Contra Costa, Los Angeles, San Diego, San Mateo, San Francisco, Santa Clara, and Santa Cruz; **7.25%:** other counties.

The Curly Control Structure Set

Kourtis Giorgio

Genoa, Italy

[Continued from the previous issue...]

Create a very large table. The available memory could consist of a non-integer part of cells. Nevertheless, the definitions work correctly and do not corrupt unavailable memory (this depends on the choice #times:=diff/step).

```
MaxAvailableChunk (addr size )
constant LogosSize  constant Logos
: LogosInit
  Logos  LogosSize Cell SIZE
  LOOP{ I off LOOP} ;

: LogoBackSearch ( logo -- false | addr true)
  Logos LogosSize cell SIZE BACK
  LOOP{
    dup I @ =
    WHEN{
      drop I true
    WHEN}
  }COMPLETED{ drop false
  LOOP}  ;
```

Simple examples

`10 19 3 End LOOP{ I . LOOP}`
types (19-10)/+3 = three numbers. These are: 10 13 16.

`19 10 -3 End LOOP{ I . LOOP}`
types (10-19)/-3 = three numbers. These are: 19 16 13.

`10 20 3 LOOP{ I . LOOP}`
types (20-10)/+3 = three numbers. These are: 10 13 16.

`20 10 -3 LOOP{ I . LOOP}`
types (10-20)/-3 = three numbers. These are: 20 17 14.

Subtle exercise

What gets typed by this phrase:
`10 20 3 END BACK LOOP{ I . LOOP}`

(Solution—the same numbers, but in reverse order, as `10 20 3 END LOOP{ I . LOOP}`. That is, 16 13 10.)

Implementation Preliminaries

It's possible to implement the above set of control-flow words in at least two different ways. First is by using the usual words like BRANCH, TBRANCH, FBRANCH, and similar techniques.

Additionally, an excellent idea appeared in the article, "LEAVEable DO LOOPs: a return stack approach" by George Lyons (*FORML Proceedings 1982*, page 132). It is a pity that such a good idea hasn't been considered much in subsequent works. Briefly, that idea consists of compiling after the beginning of the control structure a pointer to the end of the control structure (and to other relevant points, like }LEAVING{, etc.). Afterwards, at run time, when we enter a control structure we have to push the address of the beginning onto the return stack, along with other things like the index and step values, where applicable. Then any word, without need of pointers compiled later, can jump to the beginning of the control structure or, by using the pointer compiled at the beginning, can jump to the end or to other relevant points.

The above solution, a little more refined, is very powerful and provides really new possibilities, like the words AGAINS and LEAVES. 1 LEAVES is like LEAVE, 2 LEAVES leaves two levels of nested control structures, 3 LEAVES leaves three levels, etc. The concept of what is structured and what is not is cleaned up and some clarity is achieved.

Pascal, C, and standard Forth do not offer such possibilities. Besides (although the code isn't actually provided), it is possible to defined "named" control structures like LEAVE-TO and AGAIN-TO. So the sequence ABORT-CS AGAIN-TO would mean ABORT, while COLD-CS AGAIN-TO would mean COLD-START. Additionally, OUTMOST-CS LEAVE-TO would mean BYE.

For efficiency purposes when programming in machine language for typical processors like 68xxx and 80x86 and using assembler control structures, it may be necessary to use the usual xBRANCH words, providing a less powerful set of control structures (but probably more than enough for assembler needs).

Forth processors, on the other hand, are easily adaptable to variations of the more powerful solution, sometimes with gains in efficiency.

In this article, I provide two implementations. One is a 68xxx implementation of the more powerful solution for Forth control structures. Because the code presented must be readable by people using other processors, pseudo-assembler code is given where appropriate. So if you are interested in the precise implementation but do not know the 68xxx, please don't panic. (I found I am able to read 80x86 assembler easily without knowing the processor.)

Due to lack of time, I have been unable to present code for the less powerful solution. For a preview, please refer to the code provided in the article "Stack Variables (*FD* XII/1). Sorry!

Implementation Explanation

The generic control structure has the following format:

```
xxxx{  MainCode   }pppp1{  someCode1
                  }pppp2{  someCode2
                  ...
                  }ppppN{  someCodeN
xxxx}
```

`xxxx` may be any one of `CONTROL`, `REPEAT`, `CASE`, `FOR`, `TIMES`, `LOOP`, and `RECOVERABLY` while `pppp1`, `pppp2`, etc. can be any one of `LEAVING`, `COMPLETED`, `ONERROR`, etc. Not every ppppX is applicable to every `xxxx`.

So only some combinations are valid. Actually, the maximum number of pppp's is two (`LEAVING` and `COMPLETED` together or `LEAVING` and `ONERROR` together).

The position of reference points (pppp's) must be recorded at the beginning of the control structure, along with the position of the end of the structure.

The control structures `CONTROL`, `REPEAT`, `CASE`, and `FOR` can only use `LEAVING`, while `TIMES` and `LOOP` can also use `COMPLETED`. The `RECOVERABLY` structure can use `LEAVING` and `ONERROR`.

So with `CONTROL`, `REPEAT`, `CASE`, and `FOR` two pointers are necessary—one for the optional `LEAVING` point (if the `LEAVING` point doesn't exist, a -1 is stored in the pointer) and another for the end of the control structure. See Figure Two to understand the compilation effects of `CONTROL`, `REPEAT`, `CASE`, and `FOR`.

In `TIMES` and `LOOP`, three pointers are necessary—one for the *leaving* point, one to the *end* of the control structure, and one to the *completed* point (see Figure Three).

Upon entering a control structure, a "control-structure return-stack frame" is generated. A processor register CSF (control structure framer) is reserved to point to the actual frame, allowing access to index values from secondaries called within the control structure or, more generally, while using the return stack. The control-structure return-stack frame is composed of both necessary and optional items, the latter depending on the control structure. See Figure Four for a description of the return stack frame in various cases.

The exact actions we have to take upon entering a control structure are:

1. Depending on the number of extra values needed (index, step, and backcounter; or index only; or nothing), we have to adjust the return stack pointer to reserve space for them.
2. We have to push onto the return stack the address (absolute or relative) of the routine that will deallocate the return stack and/or other resources related to the control structure.
3. We have to advance the instruction pointer to skip over the pointers compiled after the beginning of the control structure and let it point to the first word after the beginning.
4. We have to push the old contents of the CSF register onto the return stack.
5. We have to push the contents of the adjusted IP to mark the address of the beginning of the control structure.
6. Having completed the control structure stack frame, we have to store into CSF the contents of the RP register, letting CSF point to the new control structure frame.

On the other side, leaving the control structure involves the following actions:

1. Using the CSF register, recall the saved value of the IP pointing to the beginning of the control structure.
2. Fetch the pointer compiled at the beginning of the control structure that points to the end.
3. Set the IP register to point to the first word after the end of the control structure.
4. Use the address pushed onto the return stack that points to the deallocating routine, and jump to that routine. Standard cases are handled by three very similar routines that deallocate the space used into the return stack, along with the space occupied for extra data like the index, step, and backcounter. Other control structures may have much more complicated un-framing actions. (The flexibility provided by pushing an address of an un-framing routine, rather than a number of cells to deallocate, is absolutely necessary for other control structures like `RECOVERABLY`, `TRACK`, and `LOCALS`.)

Rationale for Name and Notation Choices

The need to extend the set of control structures has been described in many previous articles. What I'll describe here are the choices peculiar to the set of control structures proposed in this article.

The idea of using the same name at the beginning and end of a control structure simplifies both the choice of names when inventing control structures and their memorization by the user when learning new ones.

I could have chosen, as in other languages, to write `REPEAT { ... }` or `REPEAT begin ... end` without using a `REPEAT` before the closing bracket. While this is possible with very slight modifications to the presented code, I found that readability and compile-time error checking are greatly enhanced by specifying what is beginning and what is ending, instead of asking the programmer to stack this in his brain. If you try to program in C or Pascal, you'll soon realize what I mean.

There isn't any real reason for selecting `{` and `}` for opening and closing symbols instead of `(` and `)`, or `[` and `]`, or < and >. The motivations are mainly aesthetical or practical ones depending on the keyboard used: American, French, Italian, Swedish, etc,

The choice to write `xxxx{ ... xxxx}` instead of `xxxx{ ... }xxxx`, or `{xxxx ... xxxx}`, or `{xxxx ... }xxxx` depends on the fact that the word `xxxx{` could be written as `xxxx CSbegin` and that `xxxx }` could be written as `xxxx CSend`.

In fact, I am uncertain of which to select:

```
TIMES{ ... TIMES}
TIMES { ... TIMES }
TIMES{ ... }
TIMES { ... }
```

Actually, the provided code allows `TIMES{ ... TIMES}` but also, while discouraged, `TIMES{ ... }` and, similarly, `WHEN{ ... WHEN}` as well as `WHEN{ ... }` (useable for very short WHENs).

So the { and } signs are read as "begin" and "end," while the spelling used is postfix.

When you indent vertically, in my opinion

```
TIMES{
TIMES}
```

reads better than

```
TIMES{
}TIMES
```

Furthermore, locating the braces at the end of each word helps indicate that the beginning of the control structure is really outside the structure, so that in a loop it is only executed once. Likewise, it helps indicate that the word compiled at the end of the control structure is inside the control structure, so it is executed repeatedly.

About the name choices, I haven't found anything better for `CONTROL` (any ideas?). `REPEAT` and `CASE` are needed to maintain historical continuity. `FOR` has been borrowed from the C language, where it allows for the test of any condition and the execution of any operation. The name `FOR` conflicts with the established use of `FOR NEXT`, but I don't think *that* is clear, either. (Wouldn't it be more appropriate as `COUNT BACK` or `FOR PREVIOUS`, etc.?) Nevertheless, if someone has a better name to propose, it is welcome.

`TIMES` is obvious, and reads well. `WHILE` has the same meaning as before (and gains more flexibility). `WHEN` is short and could be renamed as `?LEAVE` for clarity—but once used, it is a good name.

In particular, I think `WHEN{ ... WHEN}` read very well. `}COMPLETED{` is long and would be clearer if named `}ONCOMPLETION{`, but I don't like typing so much. `}LEAVING{` means "while leaving..." and sometimes, but not always, could be named `}ELSE{`.

Name choices depend mainly on personal taste, and the discussion could go on forever without being really constructive—so here it ends.

IF ELSE THEN

Forth's main control structure hasn't been changed for several reasons:

Efficiency.

Code simplification (if the `IF THEN` structure generated a return stack frame, a `LEAVE` embedded in it would have the effect of leaving `IF THEN` instead of the outer control structure).

Presumed psychological resistance from individuals (myself included) to so radical a change.

The main reason for leaving `IF THEN` unchanged is the efficiency preserved. Nevertheless, many complaints have been raised about its counter-intuitive syntax. While any syntax becomes intuitive once it is learned, the time needed to memorize a syntax depends on its relationship to previous use (usually spoken language). New names for `IF THEN` that follow the presented syntax guidelines are `THEN{` and `THEN}`, with the word `IF` acting like an optional comment word that doesn't compile anything. So we could write:

```
IF 3 X @ >  THEN{ ... THEN}
```

which is equivalent to

```
   3 X @ >  THEN{ ... THEN}
```

equivalent to the classic

```
   3 X @ >  IF ... THEN}
```

The syntax shown is probably more teachable than the old one. But I resisted the temptation to rename it, because my goal wasn't to offer new names for old words but to offer new possibilities in a coherent, unitary frame.

Future Directions

I already have some ideas of how to expand the presented control structure set, but I am still experimenting with these extensions. When they become more stable, I will present them. Meanwhile, here are some ideas to think about.

RECOVERABLY

```
RECOVERABLY{
code.to.execute
  }ONERROR{ error.handler
RECOVERABLY}
```

(See provided code for more elucidations.)

MULTILOOP

```
(( start0 step0  start1 step1 ... startN stepN ))
#times
MULTILOOP{ ... I0 I1 I2 etc.
MULTILOOP}
```

Iterate a loop that takes a variable number of starts and steps and, at any iteration, moves all the indices together, each with its own step. The loop must be executed `#times`.

TRACK

Every word that allocates a resource (e.g., files, memory, windows, hardware, etc.) must place into a stack variable or onto the stack an identifier for the allocated object and for the deallocating routine. Leaving the `TRACK` structure for any reason must have the effect of deallocating, in addition to the return stack, all the resources allotted within the above structure.

The `LEAVE` action may be executed as the result of a `LEAVE`, `WHEN`, `WHILE`, or similar word, or due to an error that happened inside a word called directly or indirectly within the `TRACK` structure. (Pay attention to the implementation of `LEAVES` and `ERROR`.)

Articles on Control Structures

Forth Dimensions

Vol. 1 No. 3 "D-Charts," Kim Harris.
Vol. 1 No. 5 Case statement contest.
Vol. 1 No. 5 "Forth-85 Case Statement," Richard B. Main.
Vol. 2 No. 2 "A Generalized LOOP Construct for Forth," Bruce Komusin (multiple WHILEs).
Vol. 2 No. 3 Case contest
Vol. 2 No. 3 "The Kitt Peak GODO Construct," David Kilbridge.
Vol. 2 No. 4 "Case Statement," Bob Giles (letter).
Vol. 2 No. 4 "The CASE, SEL, and COND Structures," Peter H. Helmers.
Vol. 3 No. 1 "Compiler Security," George W. Shaw.
Vol. 3 No. 3 "Multiple 'WHILE' Solution," Julian Hayden (letter).
Vol. 3 No. 6 "Cases Continued," John J. Cassady.
Vol. 3 No. 6 "Eaker's CASE for 8080," John J. Cassady.
Vol. 3 No. 6 "Generalized CASE Structure in Forth," Edgar H. Jr. Fey.
Vol. 3 No. 6 "CASE as a Defining Word," Dan Lerner.
Vol. 3 No. 6 "Eaker's CASE Augmented," Alfred J. Monroe.
Vol. 3 No. 6 "Transportable Control Structures with Compiler Security," Marc Perkel (LEAVE discussion).
Vol. 4 No. 3 "Forth-83 DO LOOP," Robert L. Smith.
Vol. 4 No. 3 "Forth-79-compatible LEAVE for Forth-83 DO LOOPs," Klaxon Suralis.
Vol. 5 No. 3 "Yet Another Case Statement," Marc Perkel (letter).
Vol. 5 No. 3 "RPN Blues—Revisited," Horst G. Kroker.
Vol. 5 No. 3 "Forth-83 Standard," Robert L. Smith.
Vol. 5 No. 3 "Forth-83: a Minority View."
Vol. 5 No. 4 "Forth-83 Loop Structure," Bill Stoddard.
Vol. 5 No. 5 "Within WITHIN," Gary Nemeth.
Vol. 5 No. 5 "A More General CASE," Martin Schaaf (letter).
Vol. 5 No. 5 "Just One Exit in Case," Ed Schmauch (letter).
Vol. 5 No. 6 "Do...When...Loop Construct," R.W. Gray.
Vol. 5 No. 6 "DO...LOOP 83 Caution," Nicholas Pappas.
Vol. 6 No. 1 "Parnas' it...ti Structure," Kurt W. Luoto (subcases COR CAND).
Vol. 6 No. 1 "More on WITHIN," Rich Leggit (letter).
Vol. 6 No. 2 "Forth Control Structures," David W. Harralson.
Vol. 6 No. 4 "ANDIF and ANDWHILE," Wendall C. Gates.
Vol. 6 No. 6 "Enhanced DO LOOP," Michael Hore (fallthrough).
Vol. 6 No. 6 "Techniques Tutorial: YACS," Henry Laxen.
Vol. 7 No. 1 "YACS, part two," Henry Laxen.
Vol. 7 No. 1 "Another Forth-83 LEAVE," John Hayes.
Vol. 7 No. 3 "Improved Forth-83 DO LOOP," Dennis Feucht.
Vol. 8 No. 4 "Second Take: Multiple Leaves by Relay," Richard Miller (letter).
Vol. 8 No. 5 "Ultimate Case Statement," Wil Baden.
Vol. 12 No. 2 "Interactive Control Structures," John R. Hayes.

FORML Proceedings

1981 "Unravel and Abort. Improved Error Handling for Forth," David Boulton.
1981 "A Generalized Forth Looping Structure," Robert Berkey (COUNTS RANGE).
1981 "Comprehensible Control Structures," Howard Jr. Goodell (new syntax).
1982 "Non-Immediate Looping Words."
1982 "LEAVEable DO LOOPs: a Return Stack Approach," George Lyons.
1983 "Modern Control Logic," Wil Baden.
1983 "Error Trapping, a Mechanism for Resuming Execution at a Higher Level," Klaus Schleisiek.
1982 "Proposed Extensions to Standard Loop Structures," Kim Harris and Michael McNeil.
1983 "User-Specified Error Recovery in Forth," Don Colburn.
1984 "Doubling the Speed of Indefinite Loops," Michael McNeil.
1984 "An Improvement Proposal for DO +LOOP Structure," John Bowling.
1984 "Yet Another CASE," John Rible.
1984 "Error Trapping and Local Variables," Klaus Schleisiek.
1985 "Interpretive Logic," Wil Baden.
1985 "Improvements in Error Handling," Loring Cramer.
1985 "Error Handling Using Standard Compiler Directives," Frans Cornelis (definition of QUIT).
1985 "Extending Forth's Control Structures into the Language Requirements of the 90's," David W. Harralson.
1986 "Charting, Escaping, Hacking, Leaping Forth," Wil Baden.
1986 "Extended Forth Control Structures for the Languages Requirements of the 1990's," David W. Harralson.
1987 "Loops and Conditionals in LaForth," Robert L. Smith.
1987 "Interpreting Control Structures the Right Way," Mitch Bradley.
1987 "Forth Control Structures for the Language Requirements of the 1990's," David Harralson.
1988 "GOTO: A Proposal," C.H. Ting.
1989 "Have Dot-if Dot-else Dot-then," Klaus Schleisiek-Kern.
1989 Control-Flow Words from Basis 9," Wil Baden.
1989 "Pattern-Matching in Forth," Brad Rodriguez (interaction between control structures and pattern matching).

Rochester Forth Conference Proceedings

1981 "Transportable Control Structures," Kim Harris.
1982 "The Importance of the Routine QUIT," Hans Nieuwenhuijzen.
1982 "Techniques Working Group," Rieks Joosten.
1984 "Hello, a Reptil I AM," Israel Urieli.
1985 "REvised REcursive AND? 'REPTIL :IS" Israel Urieli.
1985 "Exception Handling in Forth," Clifton Guy and Terry Rayburn.
1986 "Do-Loop Exit Address in Return Stack and ?leave."
1988–89 not available to author
1990 "Non-Local Exits and Stacks Implemented as Trees," R.J. Brown (abstract).
1990 "Cryptic Constructs," Rob Spruit.

Dr. Dobb's Journal

9/83 "Non-deterministic Control Words in Forth," Louis L. Odette.
1/84 "Non-determinism Revisited," Kurt W. Luoto.
11/86 "Extended Control Structures," Wil Baden (letter).

Miscellaneous Sources

"Adding GOSUB to Forth," Michael Ham. *Computer Language* 4/86.
"A Fast and Versatile Control System Using High-Level Programming," I Ohel. Motorcon 81 Conf.
"Extensibility with Forth," Kim R. Harris. *Proceedings of the West Coast Computer Faire* (date n/a).
"Data Structures Issue," James Basile. *Journal of Forth Application and Research* Vol. 2 No. 1.

Figure Seven.

```
xxxx{   SomeCode1      WHEN
        SomeCode2      WHILE
        SomeCode3      WHEN{   SomeCode5   WHEN}
        SomeCode4      WHILE{  SomeCode6   WHILE}
        SomeCode7      }LEAVING{ CodeToInsert xxxx}
```

Figure Eight.

```
xxxx{   SomeCode1      WHEN{     CodeToInsert    WHEN}
        SomeCode2      WHILE{    CodeToInsert    WHILE}
        SomeCode3      WHEN{     SomeCode5       WHEN}
        SomeCode4      WHILE{    SomeCode6       WHILE}
        SomeCode7      xxxx}
```

Such an error, besides resuming execution at the level of the first error handler above the TRACK structure, will also have the automatic effect of deallocating the resources allotted within that structure without leaving open files, unused memory, etc. In addition, if the TRACK} word is reached, resources still left intact will be deallocated automatically.

LOCALS

The locals solutions can be viewed at the internal of the control structure frame. The syntax could be:

```
( x l o )
L{ A B C -- ... code ... L}
```

Conclusions

I hope to have shown that the presented control structure set is easy to use and learn, powerful, expandable, uniform, and unifying. More work has to be done on the RECOVERABLY and TRACK structures, and on the pattern-matching problem that is related to control structures. Is anybody willing to implement the above structure set for the 8086 processor on another system (F-PC, for example) and to present the developed code? Does anybody have any new control structure?

Has anybody encountered inconsistencies in the above set of words? I would be very glad to discuss the positive and negative issues of this wordset and any problems that remain unresolved.

Speculating on the structure of Forth engines, I believe I have found ways to render these control structures "pipelineable" and as efficient (or more so, due to pipelines) as normal branch words. If fact, variations of the above scheme are easily adaptable on some Forth engines to run as fast as their branch equivalents. For structures like BEGIN WHILE REPEAT in particular, pushing the address of the beginning of the structure onto the return stack means that, without compiling offsets, the code is relocatable automatically while given the efficiency of subroutine return (or better program counter load from the top of the return stack); and we are able with "slight" processor modifications to execute an AGAIN concurrently with some other data stack manipulation.

Does anyone have the ability to do benchmarks of various solutions? Are modifications needed to achieve maximum performance?

How does this wordset compare to other solutions in Forth or, more generally, to the control structures of other languages? Are there ideas to borrow from other languages?

If, as is the case, flexibility and freedom are the best characteristics of Forth, let's use them to our best advantage.

And to conclude our story:

AUTHOR
(tired, observing the reader)
Do you like all this?

READER
(thinking)
Hmm! Have you got the code for this?

AUTHOR
(serious)
Sure, on the following pages!. (Becoming impatient) But tell me, do you like it?

READER
(smiling)
Let me try, my friend. I'll try the code and tell you.

AUTHOR
(thinking silently)
...Forthers are never satisfied... very, very strange people...

```
\ When the size is unspecified default is long.
\ B> W> L> mean respectivelly "byte move," "word move," "long move,"
\ The assembler chooses always best form for instruction.
\ That means that add, may compile
\ addg ( add general ) addq ( add quick ) addi ( add immediate )
\ and move may compile moveg ( move general ) moveq ( move quick to data reg )
\ or movei ( move immediate ). For address registers , size is ininfluent
\ so a long immediate move may be compiled as word immediate move
\ and a long adda ( add to address register ) may be compiled as word
\ immediate adda ( if the immediate value is small enough ).

: CodeAddrOf  ( "name"InputStream  --  Pfa )  '  cfa  [compile] literal ;
                                                                 immediate
macro: CSbegining        CSF ()   macro; \ Observe figure 5 to understand to
macro: OldCSF         04 CSF I)   macro; \ what make reference the above words.
macro: releaser       08 CSF I)   macro; \ CSF I) means in forth pseudocode:
macro: Index          0A CSF I)   macro; \ "CSF @ +"

macro: Step           0E CSF I)   macro; \ Step and counter are used by the
macro: Counter        12 CSF I)   macro; \ loop structure.

macro: OldErrorCSF    0E CSF I)   macro; \ OldErrorCSF and OldSP are used
macro: OldSP          12 CSF I)   macro; \ by the recoverably structure.

variable lastErrorCSF  lastErrorCSF off \ Contains the value of the CSF of the
                                        \ last RECOVERABLY structure.
macro: IPtoBegining,        \ Set the instruction pointer to point to the
       CSbegining IP l>     \ begining of the control structure while
              CSF RP l>     \ reseting the return stack to be as when the
macro;                      \ control structure was entered. This is necessary
                            \ to allow an AGAIN beeing executed by a
                            \ secondary called within the control structure.
                            \ without filling the RetStack with unnecessary addresses.
macro: IPbeg>end,      -2 IP I)  IP  w. add,   macro;
\ Assuming the Instruction Pointer ( IP ) points to the begining of the control
\ structure make it point after the end of the control structure ( figure 3 )

macro: unframe, ( #of_extra_cells_onRS -- )
       OldCSF CSF l>         \ Restore the old contents of the CSF register
       cells 0A + ## RP add, \ while freeing from the return stack the space
macro;                       \ used for the control structure.

create releasers
  code 0unframe         0 unframe,  rts,  end-code \ Unframe a return stack frame
  code 1unframe         1 unframe,  rts,  end-code \ where the space occupied by
  code 3unframe         3 unframe,  rts,  end-code \ extra values ( index step ecc )
                                                   \ is of 0 , 1 or 3 cells.
  code RECOVERABLYrelease                          \ More elaborate behaviour to
         LastErrorCSF ApcI) a0 lea,                \ unframe a recoverably control
         oldErrorCSF  a0 () l>                     \ structure frame.
         3 unframe,
         rts,                     end-code
\ code otherReleasers  ...        end-code

macro: frame, ( releaserAddr #extra_cells_onRS #of_compiled_pointers )
 >r ( #extra_cells_onRS ) cells ## RP w. sub, \ Reserve space on RetStack for Extra cells
    ( releaserAddr       ) releasers  -  ## w. rpush, \ push offset of unframing routine.
 r> ( #of_compiled_pointers ) 2* ## IP w. add,       \ Make the IP point to the first word
                                                     \ after the control structure start.
         CSF rpush,     IP rpush,          RP CSF l>
macro;  \ push old CSF  push start IP addr  set new CSF
```

```
macro:  Resources&RSrelease,         \ execute the unframing routine
        releasers  ApcI) A0 lea,     \ Load A0 with the base addr of unframing routines.
        releaser A0 w. add,          \ Add the unframing routine offset to the base addr.
        A0 () jsr,                   \ Jump subroutine to the routine.
macro;

code LEAVE   IPtoBegining, IPbeg>end, Resources&RSrelease, next,   end-code
\ move IP to begining. Move it to end . Execute unframing routine.

code fromBeginingLeave     IPbeg>end, Resources&RSrelease, next,   end-code
\ special case more efficient LEAVE

\ Beeing at the begining of the control structure we want to jump to
\ the LEAVING COMPLETED or similar points if they exist.
\ Other wise LEAVE the control structure directly.
macro: fromBegNEXTtoReferencePoint,   ( offset_of_pointer_to_ref.point -- )
       ( offset = -2,-4,-6 )  IP I) d0 w> \ offset IP @ + w@ d0 w!
       0<,  CodeAddrOf fromBeginingLeave  CCabranch, \ No code provided for LEAVING
                                                     \ or COMPLETED. LEAVE out directly.
       d0 IP w. add,  \ move to the reference point: d0 w@ IP +!
       next,          \ and continue execution.
macro;

macro: fromBegNEXTtoLeaving,      -4 fromBegNEXTtoReferencePoint,  macro;
macro: fromBegNEXTtoCompleted,    -6 fromBegNEXTtoReferencePoint,  macro;
macro: fromBegNEXTToOnError,      -6 fromBegNEXTtoReferencePoint,  macro;
\ Beeing at the begining of a control structure go to a specific reference point.

code   NegError           error", #times is negative"  end-code
\ from assembly issue an error message.

macro: NoNegativeTimes,  0<, CodeAddrOf NegError CCabranch,    macro;
\ if the condition code flags signal a negative value issue an error message.

macro: OnlyPositive,      NoNegativeTimes, 0=, if, next, then,  macro;
\ If the Cond Code flags signal a negative number issue an error ,
\ If a they signal a 0 number stop here without doing nothing else.

code LEAVES ( #timesToLeave -- )
     d1 pop,  OnlyPositive, \ continue only if the #times is positive
     d1 wtimes<, IPtoBegining, IPbeg>end, Resources&RSrelease, wtimes>,
     \ Unframe the return stack for #timesToLeave times.
     next,
end-code

code AGAIN  ( -- ) IPtoBegining, next, end-code \ Continue from the begining
                                                \ of the control structure.
code AGAINS ( #timesToAgain -- )                \ Resume execution from the begining of the
     d1 pop,  OnlyPositive,  1 ## d1 sub,       \ n-th outer control structure.
     d1 wtimes<,  Resources&RSrelease, wtimes>, \ So n-1 control structure frames
     IPtoBegining, next,                        \ must be unframed.
end-code



 code (SIMPLE()  ( -- ) \ It gets compiled by CONTROL{ or REPEAT{ .
   CodeAddrOf 0unframe ( releaser )  0 ( extra values )  2 ( #pointers )  frame,
   next, \ when we enter CONTROL or REPEAT we have only to make a control structure frame.
 end-code
```

```
code (INDEXED{() ( value -- ) \ It gets compiled by FOR{ or CASE{ .
  CodeAddrOf 1unframe ( releaser )  1 ( extra value  )  2 ( #pointers )  frame,
  index pop, \ when we enter FOR and CASE besides making a control structure frame
  next,      \ reserving space for the index we have to set the initial index value.
end-code

macro: ?Completed, ( -- ) 0<, if, fromBegNEXTtoCompleted, then,  next,  macro;
\ if the backcounter is negative the loop must go to the COMPLETED clause or
\ if COMPLETED doesn't exist it must LEAVE the control structure.
code (TIMES{()
  CodeAddrof 1unframe ( releaser )  1 ( extra value  )  3 ( #pointers )  frame,
  index pop, NonegativeTimes, \ issue error if negative #times.
  1 ## index sub, ?Completed, \ predecrement the backCounter and if it is
end-code                      \ 0 go to COMPLETED ( of leave if COMPLETED is absent )

code (TIMES})
  IPtoBegining,               \ set the IP to the begining of the control structure.
  1 ## index sub, ?Completed, \ decrement the backCouunter and if exausted go
end-code                      \ to COMPLETED ( or leave if COMLETED doesn't exist )

code (LOOP{()  ( begining #times step -- )
  CodeAddrOf 3unframe ( releaser )  3 ( extra values )  3 ( #pointers )  frame,
  step pop,    counter pop,  NoNegativeTimes,   index pop,
  1 ## counter sub, ?Completed,
end-code
\ make the control structure return stack frame reserving space for 3 extra
\ values. Set the step value set the backCounter value ( checking that it isn't negative )
\ set the index starting value, predecrement the backcounter value
code (LOOP})
  IPtoBegining,   step d0 L>   d0 index add, \ go to the control structure start.
  1 ## counter sub,   ?Completed,            \ Add the step to the index , decrement
end-code                                     \ the backCounter check it ecc.


code (RECOVERABLY{() ( -- )
  CodeAddrOf RECOVERABLYRelease ( releaser )  3 ( extra values  )  3 ( pointers ) frame,
  lastErrorCSF ApcI) A0 lea, \ Save the old value of the variable oldErrorCSF
  A0 () oldErrorCSF 1>       \ on the return stack as an extra value.
  SP oldSP 1>                \ Save the Stack pointer position on the RetStack.
  CSF A0 () 1>               \ Set the new value of the oldErrorCSF point to the actual
  index clr,   next,         \ return stack frame. Set the initial value of the index
end-code                     \ to 0 . The index counts the #times an error occured until
                             \ now.
code ERROR
  repeat<,   lastErrorCSF ApcI) CSF cmp,  <>, while,          \ Unwind the return stack to
           Resources&RSrelease,                    repeat>, \ reach the more recently
  1 ## index add,  IPtoBegining,  fromBegNEXTtoOnerror,       \ set error handler and
end-code                                                      \ start executing the
                                                              \ ONERROR clause.
code ?ERROR ( flag -- )                              \ Do Error if the flag is
  d0 pop,  0<>,  CodeAddrOf ERROR CCabranch, next, \ true.
end-code

code ErrorPropagate                                  \ BackPropagate the error to
  Resources&RSrelease,  always, CodeAddrOf ERROR  CCaBranch,\ the previous error handler.
end-code

code StackMark    ( -- )  SP oldSP L>  next,  end-code
code StackRestore ( -- )  oldSP SP L>  next,  end-code \ reset the stack to the level
                                                       \ it had when the error handler
                                                       \ had been set.
```

```
code WHEN  ( flag -- )
     d0 pop,  0<>, if,  IPtoBegining, fromBegNEXTtoLeaving,  then, next,
end-code \ if the flag is true go to the LEAVING clause or if it doesn't exist LEAVE the CS

code WHILE ( flag -- )
     d0 pop,  0=,  if,  IPtoBegining, fromBegNEXTtoLeaving,  then, next,
end-code \ same as  "0= WHEN"

macro: ?enter,       ( condition -- )
       if,   2 ##  IP   add, next,   then, \ " 2 ## IP add, " compiles addq,
             IP )+ IP w. add, next,
macro; \ a word pair beginner ( like WHEN( ) has to decide if the code between
       \ the word pair has to be executed or skipped. If the condition is true
       \ we execute the code between the word pair.
code   (WHEN()   ( flag -- )  d0 pop, 0<>, ?enter,  end-code
code   (WHILE()  ( flag -- )  d0 pop, 0=,  ?enter,  end-code

code   (OF()      ( number_to_compare_against_index )
       d0 pop,  index d0 cmp,  =,  ?enter, \ execute the pair code if the index
end-code                                  \ equals the stack argument.

\ Forth definition of WITHIN is:
\ : WITHIN ( value lower upper )   over - >R - R> U<  ;
\ That means:  result = (Up-low) U< (value-low)
\ If you design numbers on a circle in a counterclockwise manner
\ value is WITHIN lower and upper IF AND ONLY IF starting from lower
\ and moving on the circle in a counterclockwise manner you find Value
\ strictly before then Upper.
\ ( the starting position must be checked first ).
\ So lower=0  value=10  upper=23  is okay
\    lower=23 value=23  upper=30  is okay
\    lower=10 value=30  upper=30  isn't okay
\    lower=30 value=-10 upper=-1  is okay
\    lower=34 value=-30 upper=30  is okay
\    lower=0  value=-4  upper=0   isn't okay

code (WITHIN()   ( lower upper )  ( d0:=lower, d1:=index, d2:=upper )
   index d1 l>  d2 pop, ( upper )  d0 pop, ( lower )
   d0 d1 sub,  d0 d2 sub,  \ subtract lower from both index and upper.
   d1 d2 U<, compare, ?enter,
   \ above line is equivalent to: d1 d2 cmp, CC, ?enter,
   \ That means in forth pseudocode : d1 @ d2 @ - U< ?enter,
end-code
\ The rawIN is used as subroutine (the code is unneficient but doesn't matter).
code rawIN  ( num1 num2 ... numN N -- ) \ subject on d0 result on d1.
  a0 pop, \ keep in a0 the return address.
  0 ## d1 l>   d2 pop, ( d2 contains the counter )
\ Loop on register d2. If at start d2 is 0 the Loop isn't done.
  d2 wtimes<,                          \ d2 @ times<
              SP )+ d0 cmp,            \ SP @ @ d0 @ -  4 SP +!
              0=, if,  -1 ## d1 L> then, \ 0= if -1 d1 ! then
     wtimes>,                          \ times>
  a0 () jmp, \ return from subroutine.
end-code

code backIN ( num1 num2 ... numN N subject -- flag \\  group subject -- flag )
  d0 pop,  CodeAddrOf rawIN  Absr,  d1 push,  next,
end-code
```

```
: IN        ( subject  num1 num2 ... numN N -- flag ) ( subject group -- flag )
  dup 1+ pick backIN nip ; \ Doing ROLL would have been unefficient.
code (IN{) ( num1 num2 ... numN N -- )
  index d0 1>   CodeAddrOf rawIN  Absr,    d1 tst,  0<>, ?enter,
end-code

\ To use IN{ IN} ecc consider to define (( and )).
\ They may be defined as
\ Svariable OldDepth
\ : ((   ( -- )   depth OldDepth push   ;
\ : ))   ( -- )   depth OldDepth pop -  ;
\ or if you aren't familiar with Stack Variables as described in FD XII number 1
\ you may use this alternative definition ( that allows for nested (( and )) :
\ VARIABLE OLDDEPTH
\ : ((   ( -- x ) OldDepth @  depth oldDepth ! ;
\ : ))   ( x n1 n2 ... nN -- n1 n2 ... nN  N )
\       depth olddepth @ - dup 1+ roll OldDepth ! ; \ "1 ROLL" means SWAP

code I     ( -- IndexValue )  index push,  next,  end-code
code TO-I  ( newvalue   -- )  index pop,   next,  end-code
code STEP  ( valueToAdd -- )  \  "STEP" or "+TO-I"
                  d0 pop,  d0 index add, next,  end-code \ add to the index a value
code J     ( -- IndexValue )
           oldCSF a0 1>      \ reference the old Control structure frame
           0A a0 I) push,    \ Attention no information localisation.
           next,             \ Value 0A is that of the "index," macro.
                             \ Better but unefficient definition is:
                             \ CSF a0 1>   OldCSF CSF 1>  index, push,  a0 CSF 1> next,
end-code

\ Full compile time error checking is provided.
\ An easy syntax is provided to construct new control structures.
structure{ BegStructure
     cell: >BegToken                                  \ the words >begToken >BegStarter
     cell: >BegStarter    cell: >BegEnder             \ >BegEnder are equivalent to:
     cell: >Beg#pointers  cell: >BegApplicableMids \ 0 CELLS +   1 CELLS +   2 CELLS +  ecc
structure}
\ the above structure is tied to structure beginer words like CASE{ TIMES{  ecc.
\ the field >BegToken contains the token of CASE{ or TIMES{ or what is the case.
\ the field >BegStarter contains the token of the word to compile at the structure
\ begining (   (SIMPLE{)   (INDEXED}  (TIMES{)   ). See figures 3 and 4.
\ The field >BegEnder contains the token to compile at the control structure end
\ (  words like LEAVE AGAIN  (TIMES})  (LOOP})  )
\ The field >Beg#Pointers contains the # of pointers to reference points to
\ compile at the control structure begining.
\ The field >BegApplicableMids is a bit Array that specifies wich clauses
\ like LEAVING COMPLETED ONERROR ecc are applicable to the considered control structure.

structure{ mid}{Structure
  cell: >midMask  cell: >midPointerOffset  cell: >midEnder
structure}
\ The above structure is related to the clause words ( like }LEAVING{  }COMPLETED{ ecc )
\ the >midMask field contains a bit array with the bit associated to the clause
\ word on. The field  midPointerOffset  specifies the offset ( -4 for LEAVING
\ and -6 for COMPLETED ) of the pointer at the begining of the control structure.
\ See figure 3 and 4

variable Beg  variable CSbegining    variable ender
\ Tree variables to hold the token of the start word of the last
\ control structure under construction, the address of the begining of the
\ control structure  and the token of the word to compile at the end of the
\ control structure ( like LEAVE AGAIN (TIMES}) ecc ) .
```

```
: BegAddr ( -- addr ) Beg @ >body ; \ Give the address of the Begstructure associated
                                    \ with the last control structure.

: Keep&! ( NewValue addr -- OldValue ) dup @ >r ! r> ; \ Store a new value into a variable
                                                       \ holding the old one on the stack.
: ofspoints  ( offset_of_pointer  addr_to_point ) \ Set the pointer compiled at the start
  CSbegining @ -  swap  CSbegining @ + w!    ;    \ of the control structure to point to
                                                  \ the specified address.

: ofspoints? ( offset_of_pointer -- flag   ) \ Does the specified pointer point
  CSbegining @ +  w@ -1 <>             ;     \ already somewhere?

: CSbegin  ( BegToken -- ) \ given the token of the control structure beginner
  >body >r
    r@ >BegToken     @  Beg KEEP&!             \ set the Beg variable accordingly
    r@ >BegStarter   @  token,                 \ compile the associated starting word
    r@ >Beg#pointers @  Ø DOold -1 w, LOOPold  \ set to -1 the initial pointers
    here CSbegining KEEP&!                     \ set the CSbegining var to point here.
    r@ >BegEnder @ ender KEEP&!                \ set the ender variable.
  r> drop ;

: CSend ( -- )  ender @ token,             \ Compile the ender token
  -2 ( end_of_structure ) here ofsPoints   \ make the pointer to end point point to the end.
  ender ! CSbegining ! Beg ! ;             \ Restor the old values of the 3 variables.

: cells, ( values .. values  #cellsToCompile -- ) \ compile a certain number of cells.
  here swap  cells allot  here cell-   DOold Iold !  -cell +LOOPold ;


\ The use of the subsequent word is like:
\ create CONTROL{  ' CONTROL{  ' (SIMPLE{)   ' LEAVE  Mids{  ' }LEAVING{  Mids} BegIs
: BegIs ( dataToFillBegStructure )
  5 cells, immediate does> ?comp  >BegToken @ CSbegin   ;
\ compile the 5 structure parameters declare immediate the structure beginer
\ previously created, and declare it to DO the code after does>

: enderIs    ( correspondingBeg --  //  "name" -IS- )
  create , immediate does> ?comp @ Beg @ <> ?abort" Ender doesn't maches Beg" CSend ;
\ declare a control structure ender word associated to the beginner.

: }  ?comp CSend  ;  immediate \ Generic ending word to be used with any
                               \ control structure start or leaving pair.

Clause words ( as }LEAVING{ and }COMPLETED{ )  have a certain bit number associated.
When we define a CLAUSE word we must "allot" the next free bit number for the clause.
When executed a clause during compilation it must check that we are into compile
state, check that the CLAUSE is applicable to the actual control structure,
check that it hasn't been already used , it must compile the ender token set
by the control structure beginner, it must set the associated pointer compiled at
the control structure begining point to HERE and finally it must set it's own ender.

variable midFreeMask  1 midFreeMask !

: mid}{Is  ( midPointerOffset midender //IS   "name" )
  create midFreeMask @  dup ,  2* midFreeMask ! 2 cells, immediate does> ( addr )
  ?comp \ check compilation state.
   dup  >midmask @  begaddr >BegApplicableMids @  and \ is it applicable to this CS ?
        Ø= ?abort" midEnderBeginer isn't applicable to that control structure"
   dup  >midPointerOffset @  ofsPoints? ?abort" mid}{ already applied."
        ender @  token,
   dup  >midpointerOffset @  here ofsPoints \ set the pointer point here
   dup  >midender @  ender ! \ set new ender
  drop ;
```

```
-4 ( midpointerOffset ) ' LEAVE            ( midender )   mid}{Is }LEAVING{
-6 ( midpointerOffset ) ' LEAVE            ( midender )   mid}{Is }COMPLETED{
-6 ( midpointerOffset ) ' ERRORPROPAGATE ( midender )   mid}{Is }ONERROR{

: Mids{ ( -- 0 ) 0 ;
: Mids} ( 0 n1 n2 ... nN -- ) 1 0 ( #pointers(at_least_one) applicableMask )
  BEGINold rot dup WHILEold >body >midMask @ or swap 1+ swap REPEATold drop ;
\ Mids{ ... Mids} is used to construct the mask of the control structure applicable
\ Clause words.




create CONTROL{ ' CONTROL{ ' (SIMPLE{) ' LEAVE Mids{ ' }LEAVING{ Mids} BegIs
' CONTROL{ EnderIs CONTROL}

create REPEAT{ ' REPEAT{ ' (SIMPLE{) ' AGAIN Mids{ ' }LEAVING{ Mids} BegIs
' REPEAT{ EnderIs REPEAT}

create CASE{ ' CASE{ ' (INDEXED{) ' LEAVE Mids{ ' }LEAVING{ Mids} BegIs
' CASE{ EnderIs CASE}

create FOR{ ' FOR{ ' (INDEXED{) ' AGAIN Mids{ ' }LEAVING{ Mids} BegIs
' FOR{ EnderIs FOR}

create TIMES{ ' TIMES{ ' (TIMES{) ' (TIMES})
Mids{ ' }LEAVING{ ' }COMPLETED{ Mids} BegIs ' TIMES{ EnderIs TIMES}

create LOOP{ ' LOOP{ ' (LOOP{) ' (LOOP})
Mids{ ' }LEAVING{ ' }COMPLETED{ Mids} BegIs ' LOOP{ EnderIs LOOP}

create RECOVERABLY{ ' RECOVERABLY{ ' (RECOVERABLY{) ' LEAVE
Mids{ ' }LEAVING{ ' }ONERROR{ Mids} BegIs ' RECOVERABLY} EnderIs RECOVERABLY}


create WHEN{ ' WHEN{ ' (WHEN{) ' LEAVE Mids{ Mids} BegIs
' WHEN{ EnderIs WHEN}

create WHILE{ ' WHILE{ '' (WHILE{) ' LEAVE Mids{ Mids} BegIs
' WHILE{ EnderIs WHILE}

create ON{ ' ON{ ' (ON{) ' LEAVE Mids{ Mids} BegIs
' ON{ EnderIs ON}

create IN{ ' IN{ ' (IN{) ' LEAVE Mids{ Mids} BegIs
' IN{ EnderIs IN}

create WITHIN{ ' WITHIN{ ' (WITHIN{) ' LEAVE Mids{ Mids} BegIs
' WITHIN{ EnderIs WITHIN}
```

Working with Create ... Does>

Leonard Morgenstern
Moraga, California

It has been well said that programs are not written in Forth. Rather, Forth is extended to make a new language specifically designed for the application at hand. An important part of this process is the *defining word,* by which one can combine a data structure with an action, and create multiple instances that differ only in detail. One thinks of a cookie-cutter: all the cookies are the same shape but have different-colored icing.

The Basics

Defining words are based on the Forth construct CREATE ... DOES>. Beginners quickly learn to apply the method mechanically, using two familiar steps: 1) Start a colon definition, write CREATE, and follow by the actions that lay down data or allot RAM. 2) Write DOES> and follow by the action to be performed on the body of the word, the address of which has been put on the stack by DOES>. (Experienced programmers will please forgive certain oversimplifications.) Although the CREATE ... DOES> pair is easy to use at this basic level, understanding the details is hard because there are no fewer than three phases of action. Words compiled in one are executed in another.

A simple example is 3CONSTANT, which creates the six-byte analog of CONSTANT. (Screen One) It has two stack diagrams; the first for creating an instance, and the second for executing it. The first phase is in effect when 3CONSTANT is defined (Line One). It is a colon definition and works in the usual way; that is, : sets up a header, after which the CFA's of ordinary Forth words are compiled, and immediate words such as DOES> are executed. The process is ended by the semicolon.

In the second phase (Line Three), 3CONSTANT creates an instance named 3FOO. The CFA's that were compiled in the first phase are now executed one at a time, as follows: CREATE picks up the next word in the input stream, which is 3FOO, and makes a header from it. The commas lay down the top three words from the stack; they become the body. DOES> stops the action and sets the CFA of 3FOO to execute the Forth words that follow it at Point A. These are not executed until phase three, in which 3FOO is executed (Line Four); the address of its body is put on the stack, and the Forth words at Point A are executed, moving three Forth words from the dictionary to the stack.

Using ;CODE

Just as it is possible to substitute assembler for high-level Forth by starting an ordinary definition with CODE instead of :, one can do the same for defining words by substituting ;CODE for DOES>. In the alternate definition on Screen One, 3CONSTANT is rewritten in this way. ;CODE is followed directly by the necessary assembler words, and the definition is terminated by NEXT and END-CODE with no semicolon (Line Five).

As another example (Screen Two), we construct number-machines. The real ones look like rubber stamps, but print sequence numbers. Their Forth equivalent simply puts the next number on the stack. Note that commands can precede CREATE. We can specify that the machines reside in a vocabulary named #MACHINES. We could make all of them immediate by writing IMMEDIATE just before DOES>.

What CREATE Does

In the Forth-83 Standard, CREATE will "define a word that returns the address of the next available user memory location." Hence, if we write CREATE FOO and then execute FOO, an address is returned. Most existing Forths (fig-Forth is the important exception) follow this rule, as does the ANSI draft standard. Differences derive from the fact that each implementation interprets "the next available memory location" in its own way. For example, in F83 the dictionary is confined to a 64K space, and the address returned by FOO immediately follows the CFA. In F-PC, header and body are in separate spaces called the head segment and the code segment respectively, and FOO returns an address in the latter. The ANSI draft standard adds specifications as to alignment. The casual user need not be concerned with these details because words that allot memory, such as , (comma), C, and ALLOT itself, automatically do so in the proper place, namely, at the first available memory location.

It is worthwhile to comment here that one should not use 2+ to go from the code field to the body of a word. It will work in F83, but may not in other versions. Porting from one Forth to another is never easy, and a shortcut of this kind merely aggravates the problem. The correct word is >BODY.

CREATE can stand alone, either inside or outside a colon definition, without an associated DOES>, and is so used when the word to be created merely returns the address of its body, for example, variables and non-indexed arrays. Thus, we can write CREATE FOO and follow it with 0 ,. When FOO is executed, the address of the zero will be returned, so the action is the same as a variable. Or, we can use the predefined VARIABLE which is defined as

```
: VARIABLE CREATE 0 , ;
```

and write VARIABLE FOO. The first method is preferred when only one instance is wanted, as it avoids the overhead entailed in writing a defining word, while the second is better when multiple instances are (or might be) needed.

What DOES> Does

DOES> is immediate, and is executed during phase one of a definition. It lays down the word (;CODE) and some assembler instructions. Therefore, if you decompile a Forth word that includes DOES>, you will see (;CODE), followed by the possibly undecompilable assembler instructions. These will be followed by the address tokens of the Forth words that are to be executed in phase three.

(;CODE) is actually executed in phase two. It sets the CFA of the most-recently created header to point to the assembler instructions. At this point, we can clarify the imprecise statements made in earlier paragraphs. As a kind of shorthand, it is convenient to attribute to DOES> actions that are actually executed by (;CODE). We also say that DOES> makes the CFA of the word being defined point to the Forth words that follow DOES>, when it actually points to certain assembler instructions that precede them.

Don't confuse ;CODE and (;CODE). The latter is a "primitive" laid down by both DOES> and ;CODE. It is conventional in Forth to name a primitive by enclosing in parentheses the name of the word that compiles it. Other examples include (LIT), ("), (."), etc.

Separating CREATE and DOES>

CREATE ... DOES> are nearly always seen together, but unlike the halves of a pair of scissors, they can be useful when separated. It is not well known that DOES> can stand alone although it cannot be employed outside a colon definition. When a word that contains DOES> is executed, regardless of whether it is part of a defining word or not, the CFA of the last-created header is set to execute the Forth words that follow DOES>. Screen Three shows how to define an indexed array with 125 eight-bit elements by using an "external" DOES>.

This trick is not often used because it is not often useful, although Laxen and Perry did employ it in F83. It makes it possible to define words in groups, for example, pairs that vary slightly in spelling, or words with the same name in different vocabularies. This can be done in Forths (for example F83 and F-PC) that factor CREATE into two parts, one to get a string from the input stream, and the other to create a new word from it. In F83, for example, CREATE is defined as follows:

```
: CREATE  BL WORD ?UPPERCASE "CREATE ;
```

BL WORD gets the string and places it at HERE, leaving its address on the stack. ?UPPERCASE converts it to capitals if the variable CAPS is set, and "CREATE (a --) uses the result to form a new word.

Suppose that we are writing an adventure game in which we want compass directions to have two different actions. In the GAME vocabulary, NORTH will move the adventurer, while in the FORTH vocabulary, the same word with an appended # will act as a constant and put a number on the stack. With conventional methods, each direction would need two defining words, one for NORTH and the other for NORTH#. Screen Four shows how a single defining word, DIRECTION, can create the two at the same time.

The first step is to factor out the DOES> action of all but one of the words to be created. This is necessary because the run-time action of (;CODE) which is laid down by DOES> is to exit from the word that it is in, after setting the CFA in the most recently laid-down header. In our example, the game-word action is factored out into MOVE, which fetches the direction number from the body, and moves the adventurer. The defining word DIRECTION gets a string from the input stream, converts it to upper case, and places it in the buffer DBUF (Line One). In Line Two, the resulting string is used to create NORTH in the GAME vocabulary. DUP, lays down its parameter field, and MOVE executes DOES> to set the action. Lines Four and Five append a # to the string in DBUF, and Line Six uses the modified string to create NORTH# and set its action with DOES>.

Nested Defining Words

Seasoned Forth programmers know that defining words can create defining words, which in turn can create other defining words. The nesting can, in theory, be continued indefinitely. Suppose that we want to define colors as a series of arbitrary constants, numbered 0, 1, 2, etc., and that we also need shapes and other attributes defined in a similar way. We proceed as on Screen Five. Here ATTRIBUTE defines a word that contains the CREATE ... DOES> sequence, and is therefore another defining word. This idea is not merely a clever trick; it is the basis of most object-oriented Forth systems.

RED, BLUE, and GREEN are effectively constants with the values 0, 1, and 2, and ROUND, SQUARE, and OVAL are constants with the same series of values. I leave it to the reader to work out the detailed actions of the various words.

Some Random Thoughts

Why is there a right angle-bracket in DOES>? It originated with certain early Forths, where CREATE laid down a header whose code-field contained a pointer to the next byte in memory instead of an execution token. To set up a defining-word, it was necessary to follow CREATE by the pair, <BUILDS ... DOES>. The Forth-83 Standard changed the action of CREATE, so that <BUILDS was no longer needed, but did not change the original action and spelling of DOES>.

The action of defining words ranges from simple to complex. Simplest are those that lack DOES>. At the opposite pole are highly specialized words, for example, 1MI and 1AMI, used by the F-PC assembler to generate 80x86 commands. Beginners, carried away by a sense of power and freedom, often create too many defining words. Although there is little cost in memory or execution speed, doing this can result in hard-to-read source files. Most programs need only the built-in set of defining words and a few novelties.

Conclusion

The easy formation of defining words is one of the features that makes Forth powerful and enjoyable. At the basic level, the technique is easy to learn and apply, but programs are always better-written when a programmer is aware of what is going on. A deeper understanding is also required to create specialized extensions, which, though not often needed, can be very useful.

```
SCREEN 1
: 3CONSTANT ( n3 n2 n1 -- ) ( -- n3 n2 n1)                    ( Line 1)
      CREATE , , ,
      DOES> ( Point A ) DUP 4 + @ SWAP 2@ ;                    ( Line 2)
1 2 3 3CONSTANT 3FOO                                           ( Line 3)
3FOO .S            ( Forth will display 1 2 3)                 ( Line 4)

\ Alternate definition of 3CONSTANT using ;CODE
: 3CONSTANT ( n3 n2 n1 -- ) ( -- n3 n2 n1)
      CREATE , , ,
      ;CODE ( Point A ) POP BX  PUSH 4 [BX]  PUSH 2 [BX]
            PUSH 0 [BX]  NEXT  END-CODE                        ( Line 5)

SCREEN 2
: NUMBER-MACHINE ( -- ) ( -- n)
      CREATE 0 ,
      DOES> DUP @ 1 ROT +! ;

\ First alternate definition uses ;CODE
: NUMBER-MACHINE ( -- ) ( -- n)
      CREATE 0 ,
      ;CODE  POP BX  MOV AX, 0 [BX] INC  0 [BX] 1PUSH  END-CODE

\ Second alternate definition puts all number machines in
\ a special vocabulary
VOCABULARY #MACHINES
: NUMBER-MACHINE ( -- ) ( -- n)
      ALSO #MACHINES DEFINITIONS CREATE 0 , PREVIOUS DEFINITIONS
      DOES> DUP @ 1 ROT +! ;

SCREEN 3
: MAKE-8 ( i -- a) swap 8 * + ;

CREATE INDEX1 1000 ALLOT MAKE-8      \ 125 8-bit elements

SCREEN 4
VOCABULARY GAME                         \ Player's vocabulary
CREATE DBUF 33 ALLOT                    \ A buffer to hold the name
: MOVE DOES> @ ( Write game action here ) ;
: DIRECTION ( n -- )
   BL WORD ?UPPERCASE  COUNT DBUF PLACE                        ( Line 1)
   GAME  DEFINITIONS  DBUF "CREATE  DUP , MOVE                 ( Line 2)
   FORTH DEFINITIONS                                           ( Line 3)
   ASCII > DBUF COUNT + C!                                     ( Line 4)
   DBUF C@ 1+ DBUF C!                                          ( Line 5)
   DBUF "CREATE , DOES> @ ;                                    ( Line 6)

0 DIRECTION NORTH \ Create game word NORTH and constant NORTH#
3 DIRECTION EAST
: test [ forth ] north . [ newstuff ] north [ forth ] ;

SCREEN 5
\ Nested defining words
: ATTRIBUTE CREATE 0 ,
                DOES>  CREATE DUP @ 1 ROT +!  ,
                       DOES> @ ;

      ATTRIBUTE COLOR   ATTRIBUTE SHAPE
      COLOR RED      COLOR BLUE      COLOR GREEN
      SHAPE ROUND    SHAPE SQUARE    SHAPE OVAL
```

Leonard Morgenstern is a retired pathologist and computer hobbyist. His interest in Forth goes back over ten years. Currently, he is a sysop of the Forth RoundTable on GEnie. His son, David Morgenstern, is also an author on computer-related subjects.

A Forum for Exploring Forth Issues and Promoting Forth

Fast FORTHward

In volume 13 of *Forth Dimensions*, many FIG members requested more promotion of Forth. Here and elsewhere, we should tout the advantages of Forth. Every cause has benefitted from promotion at times. I think you'll agree that the promotion of Forth and FIG should extend to several areas.

One area is the promotion of trade or commerce. For-profit activity is ultimately what has kept us fed, clothed, and sheltered. At some point in the development of an industry, commerce also spawns "trade magazines" directed at fostering better-informed trade amongst the producers and consumers in a particular industry. Often user groups are born because of the widespread sale of one product.

Unfortunately for Forth, the trade magazines do not serve Forth adequately (although they seek an occasional Forth article). Worse, the number of people who are buying and selling Forth-based goods and services is probably too few to fund a Forth-dedicated trade magazine. Nevertheless, FIG can help promote trade by making sure vendor names and product information somehow appear in the pages of *Forth Dimensions*. I hope we will be hearing from Forth vendors in "Fast FORTHward," not as a means for them to provide product-specific information, but as a way for them to help promote Forth generally. Beyond that, the Board of Directors of the Forth Interest Group wants to try to maximize the advertising space sold (up to postal limits for this type of journal). FIG intends to play its part to promote trade.

***FIG can help promote trade by making sure vendor names and product information appear in the pages of* Forth Dimensions.**

(I hope that other parts of the magazine may soon feature articles about various Forth products in hardware, firmware, or software. We have taken steps to help ensure that this takes place appropriately. Such articles should serve the higher purpose of educating our readers about important programming techniques, about practical ways to develop successful applications, and so forth.)

Professional societies and standards efforts can promote Forth in ways that would be difficult for individual vendors. They can help ensure that consumers of a product or service are getting the best that can be made available. The ACM and ANSI organizations are well known for their service in such areas. Thanks to the dedicated efforts of Forth vendors and enterprising Forth activists, Forth contingents have been installed in each of those organizations, ACM SigFORTH and ANSI X3J14. I expect "Fast FORTHward" to offer essays describing standards and "open systems," and how they should be able to benefit everyone in our industry, consumers as well as producers of Forth products.

(One related activity that FIG has supported is the China Forth Examination project. It helped China determine the level of competency of Forth programmers and it brought guaranteed employment to the top performers on the test. Dr. C.H. Ting will be translating portions of this test into English so that we are better able to appreciate it.)

Publicity is another area of promotion that can help further a cause. It also takes many forms. For FIG purposes, publicity should help create visibility for Forth in as much of the trade and general media as possible. Another way FIG can help publicize Forth is to make sure educational materials are readily available to anyone who is curious about Forth. I promise to use "Fast FORTHward" as a forum to publish analytical essays regarding the nature of Forth. Such explorations can help educate newcomers—and they can hold the interest of the Forth pros, too. I will quote liberally (or reprint where appropriate) the materials from vendors, standards committees, Forth books, articles, and just about any source that can help shed light on this thing we call Forth. If Forth is a philosophy besides a language, then words must be found to express it adequately.

A valuable marketing exercise is to consider a marketplace without regard to existing products. What does a market composed of software and hardware developers need? Once that is known, perhaps we can state how Forth uniquely meets those needs. A market study should show how one's own product has a place among existing products serving the same customer base. Along these lines, "Fast FORTHward" invites the diverse customer base for Forth, including laboratory researchers and mechanical engineers, to write about their ideal Forth system.

The type of short articles, letters, or essays that I expect to appear here should help foster communication among the Forth user, developer, and vendor communities. As your newly appointed FIG Publicity Director, I also need reviewers who can help me determine what Forth-promotional messages should be offered to promote Forth and FIG. If you have the interest and/or background to help develop and review such materials, please contact me, in care of the FIG office. If you wish to write material for this department, send your ideas or finished work to me by way of Marlin Ouverson. *[Forth Interest Group, P.O. Box 8231, San Jose, California 95155]*

Please do your part to help Forth and FIG by renewing your membership immediately and, if possible, help me support our worthwhile cause by considering how you might contribute to this department. (If you received this issue as a complimentary gift, I hope you will see that *Forth Dimensions* is becoming a more broadly informative magazine, with more potential benefit for everyone involved.)

—Mike Elola

Vendor Spotlight

Paladin Software, Inc.

Started in 1982, this software consulting firm wrote custom software for a wide variety of industries, providing systems and applications software in projects ranging from HVAC to real-time space telemetry and serial protocol implementations. Recently, the company released DataScope™ version 2.0, the latest in a family of PC-based software products that lets PCs replace much more expensive communication debuggers and serial-line monitors.

Version 1.0 of DataScope was brought out in 1991. Version 1.4 is available as shareware (on CompuServe, FIDONET, EXEC-PC, and other bulletin boards as well as from the company itself—see "Product Watch").

Version 2.0 of DataScope features an SAA CUA-compliant (Systems Application Architecture and Common User Access) user interface option. It includes user-alterable multitasking window displays and a "Windows-like" pull-down menu interface. It also provides search tools that can find data that is ordinarily an invisible part of a transmission.

Since 1982, Paladin Software, Inc. has written a number of software applications for a variety of clients, including General Motors (Saturn plant HVAC cluster-interlink protocol), Eastman Kodak (T88 Densitometer), McDonnell Douglas Electrophoresis Operation in Space Ground Data Systems, Lockheed, ITT (Power Systems SUPERVISOR), and Federal Express (X.PC protocol and Astra Label System for the Super Tracker).

The company's founder is James Dewey. He has implemented X.PC, SECS-II, DDCMP and a variety of other protocols, primarily for applications involving single-chip microcomputers. He has programmed in PL1, PLC7, ASYST, polyFORTH, and Fortran, as well as various assembly languages. Before founding Paladin Software, Inc. he worked as an Electrical Engineer and was a senior programmer with Forth, Inc. He holds degrees in Electrical Engineering and Psychology from Cornell University.

Product Watch

May 1991
Orion Instruments revealed a trade-up program for converting from the UniLab/UDL microprocessor emulator-analyzer to a more powerful UniLab 8620 microprocessor emulator-analyzer. (The discount offer ended September 30, 1991.)

July 1991
BDS Software announced CF83, a 1983 Standard Forth for the Radio Shack Color Computer running RS-DOS.

September 1991
Paladin Software announced DataScope™ Version 2.0, a serial-line monitor and protocol analyzer sporting a windowed GUI and requiring MS-DOS 2.1 or higher running on a PC.

September 1991
Forth, Inc. announced the chipFORTH 68332 Software Development System, which includes one-year telephone support, uses an MS-DOS PC host, and includes a 68332 board set known as the Motorola Evaluation Kit (EVK).

October 1991
Forth, Inc. announced a new release of EXPRESS Event Management and Control System™, a process-control software package.

Companies Mentioned

BDS Software
P.O. Box 485
Glenview, Illinois 60025-0485
Phone: 708-998-1656

Forth, Inc.
111 N. Sepulveda Blvd.
Manhattan Beach, California 90266-6847
Phone: 310-372-8493
Fax: 310-318-7310

Orion Instruments
180 Independence Dr.
Menlo Park, California 94025
Phone: 415-327-8800
Fax: 415-327-9881

Paladin Software, Inc.
3945 Kenosha Avenue
San Diego, California 92117
Phone: 619-490-0368
Fax: 619-490-0177

On-Line Resources

ForthNet

ForthNet is a virtual Forth network that links designated message bases of several bulletin boards and information services in an attempt to provide greater distribution of Forth-related info.

ForthNet is provided courtesy of the SysOps of its various links, who shunt appropriate messages in a manual or semi-manual manner. The current branches of ForthNet include UseNet's comp.lang.forth, BitNet's FIGI-L, the bulletin board systems RCFB, ACFB, LMI BBS, Grapevine, and FIG's RoundTable on GEnie. (Information on modem-accessible systems is included below.)

The various branches of ForthNet do not have the same rules of appropriate postings or etiquette. Many bulletin board posts are very chatty and contain some personal information, and some also contain blatant commercial advertising. Most comp.lang.forth posts are not like that. ForthNet messages that are ported into comp.lang.forth from the rest of the ForthNet all originate on GEnie, which is a kind of *de facto* ForthNet message hub. All such messages are ported to comp.lang.forth with a from-line of the form: From: ForthNet@willett.pgh.pa.us ...

Most messages ported to comp.lang.forth also contain some trailer information as to where they actually originated, if it was not on GEnie.

There is no e-mail link between the various branches of ForthNet. If you need to get a message through to someone on another branch, please either make your message general enough to be of interest to the whole net, or contact said person by phone, U.S. Mail, or some other means. Thoughtful message authors place a few lines at the end of their messages describing how to contact them (electronically or otherwise).

Phone information for the dial-in services mentioned above:

RCFB (Real-Time Control Forth Board) 303-278-0364
SysOp: Jack Woehr SprintNet node coden
Location: Denver, Colorado, USA

ACFB
(Australia Connection Forth Board) 03-809-1787 in Australia
SysOp: Lance Collins 61-3-809-1787 International
Location: Melbourne, Victoria, AUSTRALIA

LMI BBS (Laboratory Microsystems, Inc.) 213-306-3530
SysOp: Ray Duncan SprintNet node calan
Location: Marina del Ray, California, USA

Grapevine (Grapevine RIME hub) 501-753-8121 to register
SysOp: Jim Wenzel 501-753-6859 thereafter
Location: Little Rock, Arkansas, USA

GEnie (General Electric Network for
Information Service) 800-638-9636 for info.
SysOps: Dennis Ruffer (D.RUFFER)
Leonard Morgenstern (NMORGENSTERN)
Gary Smith (GARY-S)
Location: Forth RoundTable—type M710 or FORTH

Forth Libraries

There are several repositories of Forth programs, sources, executables, and so on. These various repositories are *not* identical copies of the same things. Material is available on an *as-is* basis due to the charity of the people involved in maintaining the libraries. There are several ways to access Forth libraries:

FTP

Note: You can only use FTP if you are on an Internet site which supports FTP (some sites may restrict certain classes of users). If you have any questions about this, contact your system administrator for information. Your system administrator should always be your first resort if you have any difficulties or questions about using FTP.

For MS-DOS-related files, there are currently two sites from which you can anonymously FTP Forth-related materials:
WSMR-SIMTEL20.ARMY.MIL (Simtel20 for short)
WUARCHIVE.WUSTL.EDU (Wuarchive for short)

Wuarchive maintains a "mirror" of the material available on Simtel20. Simtel20 has a limited amount of material, most of it binaries for MS-DOS computers. The Forth files on Simtel20 are in directory PD1:<MSDOS.FORTH>. The Forth files on Wuarchive are in directory /mirror/msdos/forth. For detailed information on how use FTP and the Simtel20 archive (it is too much to include here), see the text files in:
PD1:<MSDOS.STARTER>SIMTEL20.INF *or*
/mirrors/starter/simtel20.inf

An FTP site containing a mirror of the FIG library on GEnie is "under construction" and will be announced when it is ready.

FIGI-L Gateway

For those who have access to BITNET/CSNet but not Usenet, comp.lang.forth is echoed in FIGI-L. The maintainer of the Internet/BITNET gateway since first quarter 1992 is as follows:

Pedro Luis Prospero Sanchez internet: pl@lsi.usp.br (PREFERRED)
University of Sao Paulo uunet: uunet!vme131!pl
Dept. of Electronic Engineering hepnet: psanchez@uspif1.hepnet
phone: (055)(11)211-4574
home: (055)(11)914-9756
fax: (055)(11)815-4272

Modem

For those desiring to use (or stuck with) modems, the dial-in systems listed above also have Forth libraries.

Note: If you are unable to access SIMTEL20 via Internet FTP or through one of the BITNET/EARN file servers, most SIMTEL20 MS-DOS files, including the PC- network at 313-885-3956. DDC has multiple lines which support 300/1200/2400/9600/14400 bps (HST/V.32/V.42/V.42bis/MNP5). This is a subscription system with an average hourly cost of 17 cents. It is also accessible on Telenet via PC Pursuit, and on Tymnet via StarLink outdial. New files uploaded to SIMTEL20 are usually available on DDC within 24 hours.

Information provided by:
Keith Petersen Maintainer of SIMTEL20's MSDOS,
MISC & CP/M archives [IP address 26.2.0.74]
Internet: w8sdz@WSMR-SIMTEL20.Army.Mil or
w8sdz@vela.acs.oakland.edu
Uucp: uunet!wsmr-simtel20.army.mil!w8sdz
BITNET: w8sdz@OAKLAND

This list was compiled 20 February 1992. While every attempt was made to produce an accurate list, errors are always possible. Sites are also subject to mechanical problems or SysOp burnout. Please report any discrepancies, additions, or deletions to the following:

Gary Smith *uunet!ddi1!lrark!glsrk!gars*
P. O. Drawer 7680 *nuucp%ddi1@uunet.UU.NET*
Little Rock, AR 72217 *GEnie Forth RT & Unix RT SysOp*
U.S.A. *ph: 501-227-7817*
fax: 501-228-9374
8–5 Central, M–F

E-Mail

For those with e-mail-only access, there is not much. For now, posts from ForthNet ported into comp.lang.forth sometimes advertise files being available on GEnie. Those messages also contain information on how to get UU encoded e-mail copies of the same files. There is an automated e-mail service. The entire FIG library on GEnie is available via e-mail, but no master index or catalog is yet available. The file FILES.ARC contains a fairly recent list of the files on GEnie, and files added since then are only documented for comp.lang.forth readers by way of the "Files On-line" messages ported through ForthNet.

If you have any questions about ForthNet/comp.lang.forth or any information to add/delete or correct in this message, or any suggestions on formatting or presentation, please contact either Doug Philips or Gary Smith (preferably both, but one is okay) via the following addresses:

- Internet: dwp@willett.pgh.pa.us
  or ddi1!lrark!gars@uunet.uu.net
- Usenet: ...!uunet!ddi1!lrark!gars
  or ...!uunet!willett.pgh.pa.us!dwp
- GEnie: GARY-S or D.PHILIPS3
- ForthNet: Grapevine, Gary Smith
  leave mail in Main Conference (0)

To communicate with the following, set your modem and communication software to 300/1200/2400 baud with eight bits, no parity, and one stop bit, unless noted otherwise. GEnie requires local echo (half duplex).

GEnie*

For information, call 800-638-9636

- Forth RoundTable (ForthNet*)
  Call GEnie local node, then
  type M710 or FORTH
  SysOps:
  Dennis Ruffer (D.RUFFER)
  Leonard Morgenstern (NMORGENSTERN)
  Gary Smith (GARY-S)
  Elliott Chapin (ELLIOTT.C)

BIX (Byte Information eXchange)

For Information, call 800-227-2983

- Forth Conference
  Access BIX via TymNet, then type j forth
  Type FORTH at the : prompt
  SysOp: Phil Wasson

CompuServe

For Information, call 800-848-8990

- Creative Solutions Conf.
  Type ! Go FORTH
  SysOps: Don Colburn, Zach Zacharia, Ward McFarland, Greg Guerin, John Baxter, John Jeppson
- Computer Language Magazine
  Type ! Go CLM
  SysOps: Jim Kyle, Jeff Brenton, Chip Rabinowitz, Regina Star Ridley

The WELL (Unix BBS with PicoSpan frontend)

- Forth conference
  Access WELL via CPN (CompuServe Packet Net)
  or via SprintNet node: casfa
  or 415-332-6106
  Forth Fairwitness: Jack Woehr (jax)
  Type ! j forth

Citadel Network - two sites

- Undermind (UseNet/Citadel bridge)
  Atlanta, GA
  404-521-0445

**GEnie is the repository of the Forth Interest Group's official Forth Library.*

- Interface (formerly Nite Owl)
  SysOp: Bob Lee
  Napa, CA
  707-823-3052

Non-Forth-specific BBS's with extensive Forth libraries:

- DataBit
  Alexandria, VA
  703-719-9648
  SprintNet node dcwas
- Programmer's Corner
  Baltimore/Columbia, MD
  301-596-1180 or
  301-995-3744
  SprintNet node dcwas
- PDS*SIG
  San Jose, CA
  408-270-0250
  SprintNet node casjo

International Forth BBS's

See Melbourne Australia in ForthNet node list above

- Serveur Forth
  Paris, France
  From Germany add prefix 0033
  From other countries add 33
  (1) 41 08 11 75
  300 baud (8N1) *or*
  1200/75 E71 *or*
  (1) 41 08 11 11
  1200 to 9600 baud (8N1)
  For details about high-speed,
  Minitel, or alternate carrier
  contact: SysOp Marc Petremann
  17 rue de la Lancette
  Paris, France F-75012
- SweFIG
  Per Alm Sweden
  46-8-71-35751
- NEXUS Servicios de Informacion, S.L.
  Travesera de Dalt, 104-106
  Entlo. 4-5
  08024 Barcelona, Spain
  + 34 3 2103355 (voice)
  + 34 3 2147262 (data)
- Max BBS (ForthNet*)
  United Kingdom
  0905 754157
  SysOp: Jon Brooks
- Sky Port (ForthNet*)
  United Kingdom
  44-1-294-1006
  SysOp: Andy Brimson
- Art of Programming
  Mission, British Columbia, Canada
  604-826-9663
  SysOp: Kenneth O'Heskin
- The Forth Board
  Vancouver, British Columbia, Canada
  604-681 3257
  Forth-BC Computer Society

U'NI-net/US

- The Monument Board (U'NI-net/RIME ForthNet bridge)
  Monument, CO
  Jerry Shifrin (ForthNet charter founder)
  719-488-9470

A Space Application for the SC32 Forth Chip

by Silicon Composers, Inc.

Overview

Applications requiring real-time control and high-speed data acquisition can take advantage of systems solutions that combine these features into one small package. Fast and easy software development is especially important to generate control programs that can be easily tested with application hardware to shorten development schedules. What follows is an example of a space application involving solar astronomy that meets this profile.

Sun spots, flares, and granularity are solar phenomena of interest to scientists since a good theory of solar dynamics must take them into account. The granularity of the sun is caused by convection cells, which appear over the entire surface of the sun. To some extent, the sun's surface is similar to a pot of boiling oatmeal with the bubbles of oatmeal paralleling the convection cells on the sun. Although convection cells are about the size of the state of Texas, high resolution visual imaging of individual cells from earth-based solar telescopes is difficult to achieve because of the distortion due to the earth's atmosphere.

Solar telescopes operating from suborbital flights have the advantage of being above the atmosphere, which allows them to acquire high resolution images that show more detail of convection-cell dynamics. For this type of mission, using a single on-board computer to control subsystems and data services can reduce system design complexity and development time. This single embedded computer can perform tasks such as telescope pointing, optics filter control and experiment sequencing as well as image acquisition, data storage, and down-link communications.

Hardware

A good embedded system for this type of application is the SBC32 single board computer (using the SC32 Forth RISC chip) and the DRAMIO32 board. Together, these provide a large solid-state memory space, high-performance I/O, and a microprocessor with plenty of horsepower and flexibility suitable for a wide variety of tasks, ranging from real time control to high speed data compression. The system software resides in 128KB of on-board shadow EPROM, which is loaded on power up into on board zero wait state SRAM (maximum of 512KB).

The DRAMIO32 Board is designed for use in applications requiring high-speed data acquisition or control capabilities. The DRAMIO32 has up to 16 MB of DRAM, a 16-bit bidirectional parallel port, 4 serial ports, SCSI port, 2 timer/counters, wristwatch chip and CMOS RAM.

For the solar telescope application, the DRAMIO32's four serial ports are used to acquire control data from and send servo-control commands to the telescope pointing, optics filtering and control, and mirror adjustment subsystems. The observation light beam is reflected to the telescope's CCD camera via servo control using parallel handshake bits and a counter/timer on the DRAMIO32 board. Solar-image snapshots are initiated at pre-programmed times. Solar-image data is read from the 16-bit parallel port and written into 16 MB of on-board battery-backed DRAM. Once the rocket telescope payload is recovered, mission data can be transferred from the DRAM to a second SBC32/DRAMIO32 system or other system by way of the DRAMIO32's ports. Alternatively, image data can be down-linked from the DRAM to a GSE (Ground Support Equipment) station.

Up to 48 MB additional DRAM can be added with the DRAMEXP plug-on board. A 64MB system can hold 1,024 gray-scale (uncompressed) 64KB images formatted as 256x256 8-bit pixels. An application specific image compression routine can be used to increase storage capacity.

SC32 technology can also be used in the GSE station. Data from the rocket telescope can be down-linked to a PC based GSE system using the PCS32 (Parallel Co-processor System32), a PC plug-in coprocessor board which uses the SC32 chip and supports the DRAMIO32. Data from the down-link is routed through the DRAMIO32 parallel port and sent out the on-board SCSI port to high-speed SCSI devices, such as optical disk, tape, or hard drive, without going through the PC. Once on the SCSI drive, data can be accessed by any SCSI based system for analysis.

Software

During project development, the SBC32/DRAMIO32 flight hardware can serve as a development system by connecting it to a host terminal or PC for I/O services. When developing applications such as instrument control, programming in Forth on 32-bit Forth hardware with high-speed I/O is a major advantage over other development methods.

Creating software in high-level interactive Forth significantly speeds up development, while running the application on a 32-bit Forth chip provides high resolution and performance. High-speed I/O permits real-time signal filtering, data compression and encryption as data is acquired or transferred. The code is tested and then placed in on-board EPROM for the space mission.

Sample Program

The following code fragment shows how straightforward it is to use this board set. ?UBW returns a flag showing that the next CCD data is available on the parallel port. Direct manipulation of the hardware is possible, such as %PARRD @ to read memory mapped parallel port data. CCD data is collected 16 bits at a time and placed in 32-bit wide 0-ws SRAM, where it is processed at high speed before being stored in slower DRAM. Access to drivers is shown in the call to SCSIWR which takes {block number, address, number of blocks} to write large chunks of data to a SCSI device. COMPRESS-IMAGE compresses and copies completed images to DRAM, and updates pointers. A list of snapshot times in an experiment sequence is loaded into EPROM or RAM before the rocket is launched. After each image is collected, RELOAD-TIMER sets the time until the next picture. After the solar imaging phase is complete, additional data is collected until memory is full. This data is then unloaded to a SCSI device after payload recovery.

Code example -- SBC32 ROCKET data collection

```
CREATE PIC 32768 ALLOT          ( allocate pic SRAM buffer)
HERE CONSTANT ENDPIC            (   and mark end)
VARIABLE NEXTIMG                ( pointer to image time array)
: COLLECT-IMAGE ( -- )          ( CCD parallel -> SRAM)
  ENDPIC PIC DO                 ( FOR size of picture DO)
   BEGIN ?UBW UNTIL             (   wait for CCD word ready)
  %PARRD @  I ! LOOP ;          ( copy parallel to SRAM)
: RELOAD-TIMER  ( -- )
  NEXTIMG @ @ 256 /MOD          ( get time till next image)
  %CTUR1 ! %CTLR1 !             ( set hardware timer reg)
  1 NEXTIMG +! ;                ( advance picture pointer)
: ROCKET ( -- )                 ( Turnkey task for launch)
  INIT-COLLECT                  ( Set pointers,timer)
  BEGIN ?MORE WHILE             ( Outer Space loop ... )
   POSITION-CAMERA              ( adjust camera if needed)
   ?TIME4PIC IF                 ( time for nth picture?)
     COLLECT-IMAGE              ( CCD image into SRAM)
     COMPRESS-IMAGE             ( compress, move to DRAM)
     RELOAD-TIMER               ( wakeup call for next pic)
     THEN  REPEAT               ( ... rest of images)
  COLLECT-REENTRY ;             ( more until reach earth)
: EARTH ( block# -- )           ( Save data on SCSI drive)
  DRAM 16384 SCSIWR ;           ( 16K blocks = 16MB! )
```

On the Back Burner

Demonstrating Competency

Conducted by Russell L. Harris
Houston, Texas

Perhaps the most basic problem facing a Forth programmer is that of obtaining, from a client unfamiliar with Forth, authorization to use Forth on a particular contract. The situation has been exacerbated in recent years by the unquestioning and near-universal acceptance of C along with the methodology of object-oriented programming. A secondary problem is that of convincing the client that the programmer has the expertise to successfully complete the assignment. The following paragraphs present one approach to surmounting these barriers.

Better vs. Safe

Programming assignments and contracts are not always won by the most talented programmer or by the one having the best tools and expertise. The factors which typically weigh most heavily in the choice of a programmer are the language in which he programs and his previous performance. The factor typically of greatest import in the selection of a programming language is code maintainability. Predictability of completion date is a factor which influences selection of both language and programmer.

Clients tend to view code maintainability as a function of the language in which the program is written, and the measure of maintainability as the relative abundance of programmers claiming proficiency with the language in question. They appear to give little, if any, consideration to the relationship between programming technique and maintainability.

Clients frequently value predictability of completion date over minimization of programming time. A program may be only one element in a complex system involving many components and the services of many vendors. With the interdependency of schedules, a missed deadline may have consequences which greatly outweigh the expenditure for programming. Likewise, once a budget has been authorized and a schedule has been set, the programmer may receive little, if any, reward for early completion. From the standpoint of the client, the best insurance against a missed deadline is to select a programmer and a language, both of which he personally knows to have produced serviceable code within reasonable time on a project of complexity comparable to that of the project at hand.

Shock Therapy, or Back to Reality

Something more than a resume listing past projects is required if the Forth programmer is to overcome the contemporary mind-set of C and object-oriented programming and bring his client back into a state of objectivity regarding Forth. He must convincingly demonstrate the capabilities of Forth, the maintainability of programs written in Forth, and his mastery of the art of programming; and he must do so in a manner which will profoundly impress his client.

A demonstration may take any of several forms. One could, for example, quote statistics, studies, or respected authorities regarding the matter in question. However, one of the more effective means of demonstrating the efficacy of a product or a technique is through the use of apparatus. In the first place, apparatus—be it basically mechanical, electrical, or virtual (i.e., a screen image) in nature—almost always draws attention. In the second place, apparatus provides a concrete example of technique. Finally, functioning apparatus proves capability.

Computerized apparatus programmed in Forth can attract and hold the attention of a client, thereby affording the programmer opportunity to demonstrate his own effectiveness and the effectiveness of Forth. Source listings which exhibit orderly arrangement, functional grouping, and intuitive names can dispel qualms regarding code maintainability. The overall appearance of the demonstration is perhaps the best indication to a client of the programmer's ability to bring to completion on schedule the project under consideration. Attention to detail is vital. Confidence in the programmer's reliability can be severely eroded by poor workmanship, by program bugs or quirks (no matter how minor), and by source code which is abstruse.

The apparatus need not relate to the project under consideration. It should perform an obvious function of some complexity. Ideally, it should allow demonstration of the manner in which the interactiveness of Forth facilitates the development cycle.

Effective Yet Practical Mechanisms

A demonstration mechanism, for maximum effectiveness, should be elegant, functional, and attractive; yet practicality usually demands that it be both simple and economical to construct. Ideally, the complexity of the mechanism (including the electronics) should be no greater than necessary to support the programming demonstration, so that the mechanism spotlights the code rather than overshadowing it.

Although demonstration apparatus frequently has no intrinsic usefulness, it should be possible to devise a number of useful mechanisms which are simple enough to be practical in this role.

A Clearinghouse

This is the first appearance of what is intended to be a regular *Forth Dimensions* column serving as a clearinghouse for the exchange of ideas and technical assistance regarding computerized apparatus for demonstration or other purposes.

The continuation of this undertaking will depend largely upon reader response. What I, as editor of this column, hope to receive is a variety of submissions, ranging from verbal descriptions and conceptual sketches to dimensioned drawings, schematics, source code, and photos of working devices, together with suggestions, criticism, and feedback regarding specific devices and the column in general.

Material for publication may be sent directly to me at 8609 Cedardale Drive, Houston, Texas 77055. I can be contacted by phone at 713-461-1618 during normal business hours and on most evenings, or on GEnie (RUSSELL.H).

Russell L. Harris is a consulting engineer working with embedded systems in the fields of instrumentation and machine control. He programs in polyForth, types on a Dvorak keyboard, and keeps his wristwatch set to Greenwich time.

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

Second Class
Postage Paid at
San Jose, CA