$10 Volume XVII, Number 1 May 1995 June

FORTH DIMENSIONS

ANS Forth Debugger

VLSI Design

A CASE Study

Link to C Subroutines

Distributed Shared Memory

Yet Another Interpreter Organization

Contents

Features

Departments

Editorial

Forth Dimensions
Volume XVII, Number 1
May 1995 June

Published by the
Forth Interest Group

Editor
Marlin Ouverson

Circulation/Order Desk
Frank Hall

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 2154, Oakland, California 94621. Administrative offices: 510-89-FORTH. Fax: 510-535-1295. Advertising sales: 805-946-2272.

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, 4800 Allendale Ave., Oakland, CA 94619. Second-class postage paid at Oakland, CA. POSTMASTER: Send address changes to *Forth Dimensions*, P.O. Box 2154, Oakland, CA 94621-0054."

A Common Language

When the idea of developing an ANS Forth was discussed by a FORML Conference working group several years ago, it was not without controversy. The community had only recently recovered from efforts that resulted in the Forth-83 Standard, which had met a mixed reception; and developing a new standard was sure to be both exhausting and expensive. But the group generally agreed that an ANS Forth could achieve greater industry recognition than any standard not mediated by ANSI, that it might result in easier sales to government bodies and certain large corporations, and that it could form a common dialect for communication about Forth both to the public and among Forth users.

Now, of course, we have ANS Forth in the form of the official ANSI document. (For those who are reluctant to pay its price, an electronic version, with or without HTML formatting, of the somewhat-less-than-official dpANS document is circulating.) The walls have not come tumbling down—after all, adoption of a new standard is a process, not an event—but the Forth Vendors List that is resurrected in this issue shows that a number of companies already are supporting ANS Forth in some way.

FIG Board Election

The Forth Interest Group is a non-profit organization governed by a Board of Directors. The Board determines the overall direction of the group and oversees its operations.

All of the seats on the current Board are up for election, and more candidates have applied than the number of positions will accommodate. At press time, statements submitted by candidates are scheduled to be included with this issue, along with a voting ballot. This means that current members of FIG (i.e., readers of *Forth Dimensions*) will be able to vote to select the members of the new Board.

I encourage you to use your vote—read the candidates' statements, mark your choices on the ballot, and return it promptly.

Forth Conferences

As of this writing, we're informed that euroForth conference dates have not been finalized, but that the event will be held this autumn in Marienbad, Germany, or at Dagstuhl Castle. Typically, euroForth has been praised for both its atmosphere and its content, and has attracted leading theorists, developers, and users from many countries. Organizer Marina Kern has extended a special welcome to North Americans.

More information will be available at the euroForth office in Hamburg:
Phone: +49 40 28015210
Fax: +49 40 28015290
E-mail: deltat@hamburg.com

Also, the annual Rochester Forth Conference has been scheduled for June 21–24, 1995. Those who might like to attend should contact The Forth Institute in Rochester, New York at 716-235-0168.

—Marlin Ouverson
FDeditor@aol.com

Letters

Market Appeal

Dear Mr. Ouverson:

I have enjoyed comments from other readers about the state of Forth and the future of the language. It is difficult to compete with various versions of C, C++, BASIC, and Visual BASIC, especially when those languages are taught in most colleges and most businesses use those tools for development of applications.

There are few books available which explain and teach the Forth language (been to a bookstore lately?). And there are no books available which provide an example business application in Forth. How do you develop an accounting, inventory, or other type of business application using Forth? Proponents of Forth indicate the language can support standard business applications, as well as embedded or special-purpose applications. Then let's have some Forth books on business projects in the bookstores! (Examples are helpful to most users.)

If you know of text, articles, or books which provide examples of meeting standard business requirements in Forth, then publish appropriate articles in industry journals or magazines. The Forth magazine needs to be on the magazine racks in most bookstores to assist in making the language known and to share its capabilities. With Windows-type environments becoming the standard on most desktops, how do you interface Forth to the windowing environment? Windows v3.1, Windows NT, or the Macintosh, how do you use these environments with Forth? How do you interface Forth to commercial database products? Or is a Forth db available?

In a word, *communication* will help Forth maintain a future.

Thank you,
Gary Weseman
Plano, Texas

Program Note

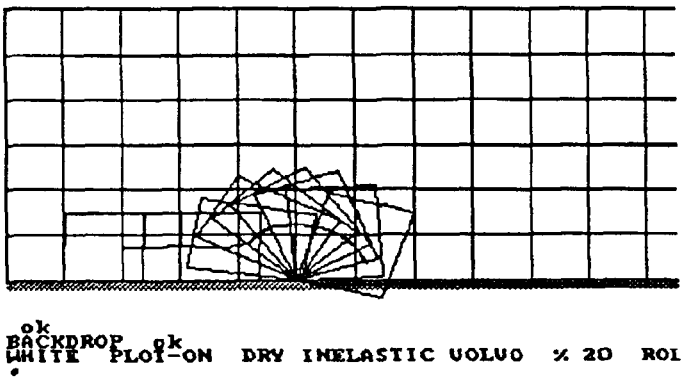
Correction to the program listing that accompanied "An Assembly Programmer's Approach to Object-Oriented Forth" in our previous issue: some TeX formatting commands appear twice in the listing and are deceptively similar to ANS Forth comments:

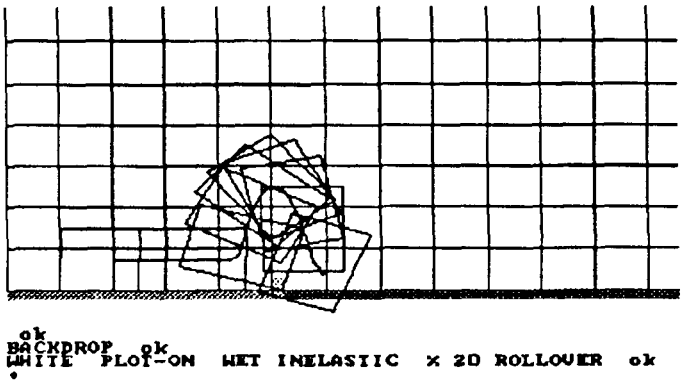
```
\end{verbatim}
\pagebreak
\begin{verbatim}
```

They should have been deleted from the listing, and we apologize for any inconvenience.

Reconstructing Accident Reconstruction

In our last issue, Julian Noble's "Vehicular Rollover in Accident Reconstruction" was missing an illustration. On page 24, the two illustrations immediately following the cross-sectional view of a change-of-grade curb should have preceded the paragraph beginning "If there is any truth..." and, in their place, the following two illustrations should have appeared:





Our aplogies to the author and to any readers whose Volvos did not roll as expected when they attempted to verify the author's data.

Debugging ANS Forth

Joerg Plewe
Muelheim an der Ruhr, Germany

Since the ANSI Forth Standard has been released, there are already several implementations of Forth systems compliant, or at least nearly compliant, with this document. In at least three (near) ANSI Forth implementations I know, one thing is missing: the debugger.

Of course, Forth does not need a debugger. Forth programs are always well factored and very highly structured. And Forth is interactive, so all components can be tested separately. But sometimes, when we do not have to convince someone of Forth's strength, having a debugger might be useful.

I found it challenging to design a debugger working only with the possibilities of an ANSI Forth system. Normally, a debugger works very closely to a compiler. It has to know everything the compiler does.

Well, a pure ANSI Forth debugger cannot be of this kind. ANSI Forth systems may be of different types concerning their methods of code generation and vary widely concerning creation style. There are systems built by assembler generating native code; others written in a high-level language, also generating native code; and high-level, highly portable implementations creating code somehow. The same diversity shows in the structure of the dictionaries: linear lists, hash tables, trees... The standard does not impose anything on a compliant system concerning generation of code or internal structure. In fact, it cannot, because it is meant to be a standard for the Forth language, not for a language on a limited circle of machines.

So how to write a debugger? When considering some clear facts, there seems to be a straight way to the "must be" design of a debugger.

You cannot access code!

Once code is generated by the compiler, there is no possibility to access it any more. There is not even a legal way to find out where it is in memory, nor whether it actually is in memory—it may be anywhere. Even if we knew where the code is, we cannot rely on anything about its structure.

So it is clear: an ANSI Forth debugger has to do at least part of its work before, or while, the code is generated. And because not too much can be done with code at run time, it has to have some kind of "self-debugging" ability.

You cannot patch the interpreter!

Some debuggers rely on patching the interpreter/compiler in order to generate the appropriate code for debugging. Sometimes `COMPILE,` or a word nearby is vectored and can be patched. Not so in an ANSI system. So if you want to manipulate the generation of code, you will have to write an outer interpreter. (There is another possibility: defining a vocabulary of "shadow words" for all words in the system. These can then compile debugging stuff. This solution is complicated and error prone. And if you want a shadow to be created with each new word, again you will have to write a new interpreter.)

You cannot access the dictionary!

There is no possibility to find a word's name from its execution token. In former times, when the execution token was called CFA, some systems provided words (like `>NAME`) which could do that. ANSI Forth does not! For this, it is not sufficient to compile the execution token somewhere and later recall the source representation of the code. You will have to reference the source in some way.

You cannot access source!

When compiling from blocks, it should be possible to compile also `BLK` and `>IN` so the source becomes available at run time. When compiling from a file, it is more complicated, because the file may be represented by a single, OS-dependent number. A standard `FILE` wordset does not have to provide the possibility to reaccess a file from a formerly assigned number. If you want to make this possible, you will have to write a second-level `FILE` wordset. I did not intend to do that.

When the source comes from TIB or a string, it is gone at run time anyway. So the source, as far you can get hold of it, will have to be compiled with the code.

Now it is clear, how a debugger has to be designed!

I do not want to describe how an outer interpreter can be built. The standard words `WORD` and `REFILL` provide

(Text continues on page 12.)

```
\ ******************************************************************
\ *                                                                *
\ *                Debugger for ANSI Forth Programs                *
\ *                                                                *
\ * Contributed to the community by                                *
\ *                    Joerg Plewe, 1dec94                         *
\ *                                                                *
\ * This code can be used and copied free of charge.               *
\ * All rights reserved.                                           *
\ *                                                                *
\ * Comments, hints and bug reports are welcome. Please email to   *
\ *                     jps@Forth-eV.de                            *
\ *                                                                *
\ * tested with: F68KANS (>jan94), pfe0.9.7, ThisForth             *
\ * Special thanks to Ulrich Hoffmann and Bernd Paysan             *
\ *  for testing and commenting.                                   *
\ *                                                                *
\ * V0.1: Added treatment of nesting levels                        *
\ * V0.2: Decompiler feature                                       *
\ * V0.3: worked in hints from the net                             *
\ ******************************************************************
\
\ The following code provides a simple debugging tool for ANSI Forth programs.
\ It may be used to debug colon- and DOES>- and :NONAME-code on source level.
\
\ The debugger expects your system to be a well-behaved Forth system.
\ (Like my F68KANS :-)
\ When you suspect that your problems arise from the compiler itself
\ (do you use an optimizer?), please use another tool.
\
\ Usage:
\ There are two pairs of words switching the debugger on and off.
\
\ +DEBUG, -DEBUG
\ These two control a global switch, which has effects both at compile time
\ and run time. When used at compile time, -DEBUG will completely switch
\ off the debugger, so no debugging code is generated. This allows you
\ to leave your code with all debugging statements in it and test it
\ without the debugger.
\ At run time, -DEBUG switches off the evaluation of debugging code,
\ so your code will behave as normal, just a bit slower.
\
\ [DBG, DBG]
\ You will have to use [DBG at compile time in front of a ':' or a DOES>
\ to tell the debugger to generate special debugging code. [DBG is
\ valid until switched off with DBG]. DBG] may appear anywhere in the source!
\ So it is possible to debug only the first part of a word and then to switch
\ off the debugger, causing 'original' code to be generated for the rest.
\ It is not possible to generate normal code at the beginning of a definition
\ and debugging code in the end!
\
\ E.g.
\ : FOO CREATE [DBG  0 ,   DOES> @ ;   DBG]
\ will only debug the DOES> part of the definition. The reason is that [DBG
\ only switches the behaviour of ':' and DOES>.
\ Think about the difference of +-DEBUG and [DBG]!
\
\ There some additional words to control the debugger a run time. These words
\ have short names to be typeable at debug time. But of course you may also
\ compile them into your code. This gives you the possibility of breakpoints, etc.
\
\ [+I], [-I]
\ Interactive. This switch controls whether you do single-stepping or a
\ kind of code animation. When single-stepping, you can type any number
\ of Forth statements between two steps. The next step is performed by
\ simply pressing <return>.
\
\ [+V], [-V]
\ Verbose. [+V] adds a stack dump to the output of each step.
```

```
\ [+S], [-S]
\ Silent. [+S] switches off all outputs and the program begins to run.
\ Pressing a key switches it back to interactive mode.
\
\ [>L] ( n -- )
\ Goto Level of nesting. This option receives a parameter (don't forget).
\ It lets the debugger run in '[+S] [-I] [-V]' mode until the given
\ level of nesting is reached the next time. Then the previous state of
\ the debugger is restored.
\ Note that the given level may be lower, higher or equal to the current level.
\ You can overwrite the settings invoked by [>L] with further debugger commands.
\ Suppose you are on level 1 -- then
\    1 [>L] [-S]
\ will give you an animation of your code until the next word on nesting level 1
\ is reached.
\
\ [Y]
\ Step over. This command will avoid nesting to deeper levels. It is
\ equivalent to a [>L] with the current level. So the example above can
\ be written as:
\    [Y] [-S]
\
\ [DEF]
\ Default: [+I] [-V] [-S], no nest level targeting
\
\ The debugger also supports a decompiler feature for words compiled with the
\ debugger on. The decompiler is invoked by
\ DSEE <name>
\ and decompiles the whole word at once. This decompiler works completely
\ differently from those you maybe know; it has, e.g., the possibility to
\ decompile even things which were in your source with the compiler off.
\ This means, sequences like '… [ 1 2 3 + + ] LITERAL …' will reappear
\ while decompiling.
\
\ 0!DBG
\ This is the debugger's reset. It sets back e.g. the level of nesting.
\ You should use this at the beginning of a file you compile, e.g.
\      0!DBG
\ in the first line.
\
\ \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\ WORKS WITH
\   F68KANS (>jan94)     portable 68K nativecode Forth by me
\   pfe0.9.7             by Dirk Zoller
\   ThisForth            by Wil Baden
\
\ Reported to work with:
\   gforth               by Bernd Paysan (paysan@informatik.tu-muenchen.de)
\   iForth               by Marcel Hendrix (mhx@bbs.forth-ev.de)
\
\ \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\ ENVIRONMENTAL DEPENDENCIES
\ When the decompiler option is used:
\   The Control Stack (CS) has to be the data stack.
\
\ \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\ RESTRICTIONS:
\ The generation of debugging code can only be invoked with the words
\ ':', DOES>, and :NONAME (or words which use them, after the debugger
\ has been compiled).
\
\ The debugger is steered by some string literals: debugging is switched
\ off when the debugger's outer interpreter finds the words DBG] or ';'.
\ The words are compiled as string literals into the debugger, so no
\ definitions including them will be able to do their jobs!
\ Further, the words ';' and '[' have a special meaning for the
\ debugger (they both switch off the Forth compiler).
\
\ In the current state, the debugger cannot handle floating-point
```

```
\ literals. This will be removed in one of the next releases.
\
\ \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\  REMARKS
\ V0.2 initially did not work with Wil Baden's ThisForth.
\ The reason seemed to be
\ that VALUES cannot be POSTPONEd in ThisForth. So I turned the VALUE
\ 'decompile' into the VARIABLE 'nodecomp'.
\ ThisForth had (has?) some problems with its REFILL. Wil Baden sent
\ me a valid definition:
\  : REFILL ( -- flag ) next-char eof <> ;
\
\ Don't wonder about what you see when debugging ThisForth programs!
\ The debugger also sees ThisForth's macro expansions!!


CR .(  ANSI Forth debugger V0.3     by Joerg Plewe, 1dec94 ) CR

MARKER *debugger*

\
\ customization
\
 \ Compile the decompiler feature?
 \ This will introduce an environmental dependency!
\ TRUE CONSTANT withDSEE

\ Try to find out whether the control stack is the data stack.
\ In this case, the system fulfills the environmental dependency
MARKER *check_for_controlstack*
FALSE VARIABLE CSisDS  CSisDS !
VARIABLE saveDEPTH

: checker
  [ DEPTH saveDEPTH ! ]
  IF              \ IF should change the controlstack
  [ DEPTH saveDEPTH @ > CSisDS ! ]    \ datastack changed?
  THEN ;

CSisDS @ *check_for_controlstack*  CONSTANT withDSEE


: is_defined ( <name> -- flag )
  BL WORD FIND NIP ;

\ prelude
\ is_defined ON  is_defined OFF  AND  0=
\ [IF]
: ON  ( addr -- )   TRUE SWAP ! ;
: OFF ( addr -- )  FALSE SWAP ! ;
\ [THEN]


\ switching debugger globally
\
VARIABLE use_debugger       use_debugger ON
   \ use the debugger at all?
VARIABLE nodecomp           nodecomp ON
   \ controls decompiling vs. debugging at runtime
VARIABLE creating_dbgcode   creating_dbgcode OFF   \ internal switch
VARIABLE nestlevel          0 nestlevel !          \ level of nesting


: +DEBUG ( -- )
  use_debugger ON ;

: -DEBUG ( -- )
  use_debugger OFF ;
```

```
\ we need some routines for service

\
\ is a string a number?
\
: ?negate ( n sign -- n' )      0< IF NEGATE THEN ;
: ?dnegate ( d sign -- d' )     0< IF DNEGATE THEN ;

: number? ( addr c -- FALSE | u 1 | ud -1 )
   \ Tries to find out whether the given string can be
   \ interpreted as a numeric literal.
   \ Returns a flag and the converted number, if possible.
        0 >R                                          \ push default sign
        OVER C@ [CHAR] - = IF R> DROP -1 >R  THEN     \ - sign?
        OVER C@ [CHAR] + = IF R> DROP  1 >R  THEN     \ + sign?
        R@ ABS /STRING
        0. 2SWAP >NUMBER  ( ud2 c-addr2 u2 )
        ?DUP 0= IF  DROP D>S R> ?negate  1 EXIT  THEN   ( exit: single )
        1 = SWAP C@ [CHAR] . = AND                    \ with a '.', it is double
        IF  R> ?dnegate  -1 EXIT  THEN                ( exit: double )
        R> DROP 2DROP FALSE
        ;


\ things to be done while debugging

CREATE debugTIB 80 CHARS ALLOT
: eval_debug_statements ( -- )
  \ A simple outer interpreter for interactive input at
  \ debug time.
        BEGIN
          CR ." > " debugTIB DUP 80 ACCEPT SPACE DUP
        WHILE
          ['] EVALUATE  CATCH IF ." Oops!?" CR THEN
        REPEAT
        2DROP ;

: .next_statement ( addr len -- )
  \ addr len shows the name of the following statement in the
  \ source code. .next_statement formats and prints it.
        nestlevel @ 2* SPACES
        nodecomp @ IF
          ." Nxt["   nestlevel @ S>D <# #S #> TYPE  ." ]: "
        THEN
        TYPE
        ;


\ steering the debugger

VARIABLE debugstate     0 debugstate !
                          \ Bit 0 = Interactive
                          \ Bit 1 = Silent
                          \ Bit 2 = Verbose

: +debugstate: ( state <name> -- )
    CREATE ,
    DOES> @ debugstate @ OR debugstate ! ;

: -debugstate: ( state <name> -- )
    CREATE INVERT ,
    DOES> @ debugstate @ AND debugstate ! ;

: ?debugstate: ( state <name> -- )
    CREATE ,
    DOES> @ debugstate @ AND 0<> ;

1  DUP +debugstate: (+I)  DUP -debugstate: [-I]  ?debugstate: [?I]
2  DUP +debugstate: [+S]  DUP -debugstate: [-S]  ?debugstate: [?S]
```

```
4  DUP +debugstate: [+V]  DUP -debugstate: [-V]  ?debugstate: [?V]


\ define some additional rules

: [+I] ( -- )          \ interactive can never be silent
       [-S] (+I) ;

VARIABLE target_nestlevel      -1 target_nestlevel !
VARIABLE savedebugstate       debugstate @ savedebugstate !

: check_nesting ( -- )
  \ Checks whether the execution has reached a defined level of nesting
  \ (target_nestlevel). In this case, it switches off targeting (-1!) and restore
  \ the previously saved state of the debugger.
       target_nestlevel @ nestlevel @ =
       IF
         -1 target_nestlevel !           \ switch targeting off
         savedebugstate @ debugstate !
       THEN ;

: [>L] ( n -- )                        \ goto level
       target_nestlevel !
       debugstate @ savedebugstate !
       [+S] [-I] [-V]
       ;

: [Y] ( -- )                       \ step over
       nestlevel @ [>L]
       ;

: [DEF] ( -- )      \ the default behaviour
       -1 target_nestlevel !
       [+I] [-V] [-S] ;

[DEF]


\ check: what has to be displayed?

: ?.next_statement ( addr len -- )
  \ When the debugger is not running silent, the following has to be displayed.
  \ When not being interactive, a CR has to be added.
       [?S] 0=
       IF
         .next_statement
         [?I] 0= IF CR THEN
       ELSE  2DROP  THEN
       ;

: ?eval_debug_statements ( -- )
  \ When the debugger is interactive but not silent, we want
  \ to evaluate statements.
    [?I] [?S] 0= AND
    IF  eval_debug_statements  THEN ;

: ?.s ( -- )
  \ Perhaps a stackdump is needed. This is indicated by the verbose mode.
    [?V] [?S] 0= AND
    IF  .S  CR THEN ;

: ?>[+I] ( -- )
  \ Oh-oh. Return to interactive mode when a key is pressed.
       KEY? IF  KEY DROP  [+I]  THEN ;

: dodebug ( addr len -- )
  \ This word is executed between two statements in the source.
  \ Note I had to do some stack juggling, for the stack has to
  \ be 'original' when showing the stackdump!
```

(Text continued from page 6.)

all things needed. Without RE-FILL, it would have been harder... I will focus on how to create self-debugging (and self-decompiling) code.

What Should the Debugger Do?

Some common needs for a debugger have to be implemented:

- single stepping
- breakpoints
- nesting control
- decompilation
- expression evaluation at debug time

All these points require one to create a specific kind of code. The idea is that running the code itself means to debug it.

The generation of code, *not* taking decompilation into account, proceeds according to the following scheme:

- Get a word from the input source (WORD) and store it somewhere.
- Compile the word as a string literal to the dictionary (SLITERAL).
- Compile a definition (COMPILE,), which provides the debugging functionality for debug time. This definition may print the string provided by SLITERAL (the source), accept command lines, and whatever you want.
- FIND the word in the dictionary.
- If found, compile or execute it (according to its IMMEDIATE status and the compiler's state held in STATE).
- If not found, try to convert the string to a number and, if successful, look at STATE and compile (LITERAL, 2LITERAL) or push the (double) number to the stack.
- If all this fails, THROW an error code.
- Proceed from the beginning.

To make these steps a little clearer, let us see what is generated from a single word in the source.

Suppose there is a word defined with the debugger called dodebug, which takes a string,

```
        use_debugger @ IF       \ wonna debug anyway?
          check_nesting
          ?>[+I]
          >R >R ?.s R> R>       ( >R's for addr len )
          ?.next_statement
          ?eval_debug_statements
        ELSE  2DROP  THEN
        ;


\ \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\
\ this section is to create debugging code
\
\ \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

\ THIS word is the main point:
\ It compiles code suitable for debugging.
\ Or better: it compiles self-debugging code

: .source, ( c-addr -- )
    STATE @ DUP >R 0= IF ] THEN     \ switch compiler on for
                                    \ SLITERAL
    COUNT
    POSTPONE SLITERAL ( POSTPONE) ALIGN
    POSTPONE dodebug
    R> 0= IF POSTPONE [ THEN        \ switch compiler off when
                                    \ it was off
    ;

CREATE wordbuf 64 CHARS ALLOT

: >wordbuf ( c-addr -- )
  DUP C@ CHAR+ wordbuf SWAP CHARS MOVE ;

: C$= ( c-addr addr u -- flag )
     ROT COUNT COMPARE 0= ;

: $;=     ( c-addr -- flag )   S" ;"     C$= ;
: $DBG]=  ( c-addr -- flag )   S" DBG]"  C$= ;
: $[=     ( c-addr -- flag )   S" ["     C$= ;

: apply_semantic ( xt +-1 -- ? )
       0< STATE @  AND
       IF  COMPILE,  ELSE  EXECUTE THEN ;

: compile_number ( u 1 | ud -1 -- )
  STATE @ 0<>
  IF
     0< IF  POSTPONE 2LITERAL  ELSE  POSTPONE LITERAL  THEN
  ELSE DROP THEN ;

: compiler_error ( c-addr -- )
  ." Not found in dictionary: " wordbuf COUNT TYPE
  -13 THROW ;


\ handling the nesting level

: +nest ( -- )
   1 nestlevel +! ;
: -nest ( -- )
   -1 nestlevel +! ;


: endof_dbgd_def? ( -- flag )     \ end of debugged definition?
       wordbuf $;=
       wordbuf $DBG]= OR
       ;
```

```
: compiler_off? ( -- flag )      \ a word, which switches
                                 \ the compiler off?
        wordbuf $;=
        wordbuf $[= OR
        ;


\ compile conditinal branches to skip 'real'
\ code for decompiling

withDSEE [IF]

CREATE CSbuffer 20 CELLS ALLOT
VARIABLE decompilerIF      decompilerIF OFF
VARIABLE saveDEPTH         0 saveDEPTH !
VARIABLE CSsaved           0 CSsaved !

: saveCS ( ? -- )
  \ Save control structure information from the data stack
  \ to a special buffer.
  \ The variable saveDEPTH has to be set!!
        0 CSsaved !
        BEGIN
          DEPTH saveDEPTH @ <>
        WHILE
          CSbuffer CSsaved @ CELLS + !
          1 CSsaved +!
        REPEAT ;

: restoreCS ( -- ? )
  \ restore control structure information from the
  \ buffer to stack
        BEGIN
          CSsaved @
        WHILE
          -1 CSsaved +!
          CSbuffer CSsaved @ CELLS + @
        REPEAT ;

: decompiler_jump ( -- )
  \ Under right conditions, compile a 'nodecomp @ IF'
  \ The possible change on data stack (IF) is cleared, so that
  \ words like LITERAL do not come into trouble.
  \ The Control Stack CS defined in ANSI document may consist
  \ of some entries on the common data stack (which, indeed, is
  \ implemented in most Forth systems). But the data stack has
  \ to be unchanged by the debugger when compiling a word:
  \ ' ... [ 1 2 3 + + ] LITERAL ...'
  \ In this example, 'LITERAL' wants to compile the number 6,
  \ and not some token left on the stack by the decompiler's IF.
  \ Because it is not known what IF will place in an arbitary
  \ Forth system, this complicated construction has to be made.
        STATE @  compiler_off? 0= AND
        IF
          DEPTH saveDEPTH !              \ DEPTH of stack 'before'
          POSTPONE nodecomp  POSTPONE @
          POSTPONE IF      \ now compile IF. It may change stack!
          saveCS           \ stack effect of IF removed
          decompilerIF ON                \ ok, there is an IF
        THEN
        ;

: decompiler_target ( -- )
  \ Resolve the decompiler IF compiled
        decompilerIF @
        IF
          restoreCS                \ prepare stack with IF-values
          POSTPONE THEN            \ and resolve the jump.
          decompilerIF OFF         \ done!
```

prints it, and lets you enter Forth lines:

```
dodebug ( addr u -- )
```

When you now compile a source where the word <word> is contained, the following code will be generated:

```
... [ S" <word>" ]
SLITERAL dodebug <word>
...
```

When this code runs, the source (<word>) is presented to the user (by dodebug) before it is executed.

Now, dodebug is a simple, high-level word which can do everything Forth can do. My dodebug uses flags to control its actions. Typing the source and accepting a command line can be switched on and off separately. Making dodebug completely silent lets the code run as normal, just a bit slower. Placing the switches in the source allows one to realize breakpoints.

Nesting Control

The debugger allows one to delve into deeper nesting levels, or to stay at the current nesting level, or even to go forth until a specified level of nesting is reached. This feature is implemented simply by using the switches mentioned above, together with a variable which holds the current level. The level is increased by words starting the definition of code, like : (colon), DOES>, and :NONAME. These words have to be redefined for the debugger anyway, so it was easy to make them increment the level of nesting. ; (semi-colon) decreases the level again.

Decompilation

Some ANSI systems already have a decompiler. For those that have not, the debugger also implements a decompiler feature. For that, the creation of debugging code is extended a little.

When you compile source in which the word <word> is contained, the following code will be

generated:

```
...
[ S" <word>" ] SLITERAL
dodebug  nodecomp @
IF  <word>  THEN
...
```

This construction executes, when nodecomp is FALSE, only the debugging part of a word, not the code itself. So you can let the code go, and it will decompile itself without being actually executed! (It's a funny thing, I know.)

Because this mechanism caused some difficulties when compiling control structures, LITERAL, or such things, there appeared to be an environmental dependency: the control stack has to be the data stack. (If the standard provided something like CS>R and R>CS, this dependency could have been avoided.) For this reason, the decompiler feature can be switched off by means of conditional compilation.

First Experience

I tested the debugger with three (near) ANSI Forth systems:

- F68KANS — my own creation for 68K
- pfe — by Dirk Zoller
- ThisForth — by Wil Baden

It was reported by kind people from the net (thank you) to run on three additional systems.

This debugger shows a property which I have never seen elsewhere: it also shows code executed at compile time! So sequences like

```
... [ 1 2 3 + + ] LITERAL
...
```

will uniquely be displayed while debugging! (Think about it. It is clear from principles of its work.)

A funny thing happens with Wil Baden's ThisForth, which does macro expansion into the input source. Because the debugger works directly on the source, you later will see the original code *and* its expansion:

```
        THEN
        ;

[ELSE]  ( withDSEE )
: decompiler_jump ;   IMMEDIATE
: decompiler_target ; IMMEDIATE
[THEN]  ( withDSEE )


\ now construct a complete outer interpreter

\ a special hack to allow F68KANS to handle files with tabs, etc.
is_defined F68kAns
  [IF] blankbits  [ELSE]  BL  [THEN]
  CONSTANT whitespace

: create_debugging_code ( -- )
  POSTPONE +nest
  creating_dbgcode @ >R creating_dbgcode ON
  BEGIN                                                  \ loop to EOF
    BEGIN                                                \ loop to EOL
      whitespace WORD DUP C@
    WHILE
      >wordbuf
      wordbuf .source,
      endof_dbgd_def? IF  POSTPONE -nest THEN
      decompiler_jump
      wordbuf FIND ( c-addr 0 | xt +1 | xt -1 )  ?DUP
      IF   apply_semantic
      ELSE                                          ( caddr )
         COUNT number? ?DUP
         IF  compile_number  ELSE  compiler_error  THEN
      THEN
      decompiler_target
      endof_dbgd_def? IF  R> creating_dbgcode !  EXIT ( **) THEN
    REPEAT  DROP
  REFILL 0= UNTIL
  R> creating_dbgcode !
;


\ Define the decompiler

withDSEE [IF]
: DSEE ( <name> -- )
  \ Show a decompiler listing of a word compiled with the
  \ debugger. A non-debugger word will be executed instead.
        CR
        nodecomp @  >R     FALSE nodecomp !
        debugstate @ >R    [-I] [-V] [-S]
        ' EXECUTE
        R> debugstate !
        R> nodecomp !
        ;
[ELSE]
: DSEE ( <name> -- )
        CR BL WORD DROP
        ." Debugger compiled without decompiler option! "
        ;
[THEN]

\ Now the replacements for the code-beginning words.

: debug: ( <name> -- )
  :   create_debugging_code ;

: debug:NONAME ( --  xt )
  :NONAME   create_debugging_code ;
```

```
: debugDOES>
  creating_dbgcode @ IF POSTPONE -nest THEN
      \ when the decompiler is invoked between ':' and 'DOES>',
      \ there has to be a '-nest' compiled before 'DOES>'.
  POSTPONE DOES>  create_debugging_code ;


\ switching the debugger on and off

VARIABLE debugging      debugging OFF

: [DBG
  debugging ON ;  IMMEDIATE

: DBG]
  debugging OFF ;  IMMEDIATE


: 0!DBG ( -- )
  \ reset the debugger
        0 nestlevel !
        POSTPONE [DBG
        +DEBUG
        [DEF]
        creating_dbgcode OFF
        ;


\ redefinition of the code generating defining words

: : ( <name> -- )
  use_debugger @ debugging @ AND
  IF  debug:  ELSE  :  THEN ;

: DOES> ( <name> -- )
  use_debugger @ debugging @ AND
  IF  debugDOES>  ELSE  POSTPONE DOES>  THEN ; IMMEDIATE

: :NONAME ( <name> -- )
  use_debugger @ debugging @ AND
  IF  debug:NONAME  ELSE  :NONAME  THEN ;

\ OK
\
CR
.( The words for you are: ) CR
.(   +DEBUG -DEBUG    to switch debugging on/off globally ) CR
.(   [DBG   DBG]      to invoke and terminate generation ) CR
.(                    of debugging code at compile time ) CR
.(   [+I]   [-I]      Interactive mode on/off ) CR
.(   [+S]   [-S]      Silent mode on/off ) CR
.(   [+V]   [-V]      Verbose mode on/off ) CR
.(   [>L]   [Y]       level-targeting control ) CR
.(   [DEF]            DEFault settings ) CR
withDSEE [IF]
.(   DSEE       Decompile words compiled with debugger ) CR
[THEN]
.(   0!DBG      Reset the debugger when something goes wrong ) CR
```

```
: TEST ( -- )
  S" Hello" ;

DSEE TEST
S"       \ original
c"       \ expansion of
count    \ S" is
type     \ c" count type
;
```

So the debugger gives you the possibility to analyze what ThisForth is doing with your code!

(It turned out that it is good advice to give the system enough space in its code area, because the code really blows up.)

Future Work

I already have some ideas for the future. I think some kind of profiling can be done in a similar way decomposing is done now. Additionally, a thing commonly called "watch" would be nice to let the code run until a condition—e.g., a variable reaching a certain value—is fulfilled, then the debugger enters its interactive state.

Thanks to all the people who reviewed and commented on the code.

The code that accompanies this article can be downloaded via anonymous ftp from taygeta.oc.nps.navy.mil, the same system that hosts the Forth Scientific Library, and can be found in /pub/Forth/ANS/debugger.ans.

The author is 30 years old, married, and the father of three children. He received his master of science in physics degree in 1991. After that, he worked for a Forth company and in a mainframe environment. For the last three years, he has worked for a scientific institute dealing with database development for researchers. In 1988, he started to develop Forth systems for 68000 machines. The ANSI-compliant F68KANS is the latest result of that development.

Readers who want to contact the author directly can use the JPS@Forth-eV.de e-mail address.

Distributed Shared Memory

Jeff Fox
Berkeley, California

Distributed Shared Memory, or DSM, in these days of increasingly networked computers has been widely recognized as a simple construct upon which to build inexpensive parallel processing systems. It is widely accepted today in supercomputing that workstation farms are more economical than supercomputers. Since these networks of workstations are often already available and interconnected, DSM software permits these machines to be used as supercomputers. This paper will discuss the use of DSM in parallel programming in the Forth programming language.

Multitasking in Forth

Forth does not provide a standard multitasking mechanism. Multitasking in Forth is usually either done in a simple, portable, cooperative scheme or with OS multitasking services. Forth is often used in embedded applications where Forth is the OS and on small computers that may have no hardware memory protection. Figure One shows a memory map for a typical cooperative multitasking Forth application. There is no physical memory protection to prevent one task from crashing the Forth OS. Instead, each task has its own stacks and user variables, but all memory is shared and available to all tasks. However, each task must have a certain amount of local memory for its stacks and local user variables. In a traditional Forth multitasker, the word PAUSE would switch control from one task to another, which simplifies task synchronization.

In a computer that provides memory protection in hardware, the OS used often provides services to limit the memory that a program or task may access. Any access to memory through protected memory hardware, or through memory access services from an OS, can provide error traps for attempts to access protected memory. Certain tasks and programs may be able to run with access to only one section of memory, but many others will need a certain amount of memory to be available to more than one program or task. Figure Two is an example of a protected-mode operating system running three tasks that are physically separate in memory except for a small area of shared memory used for data only.

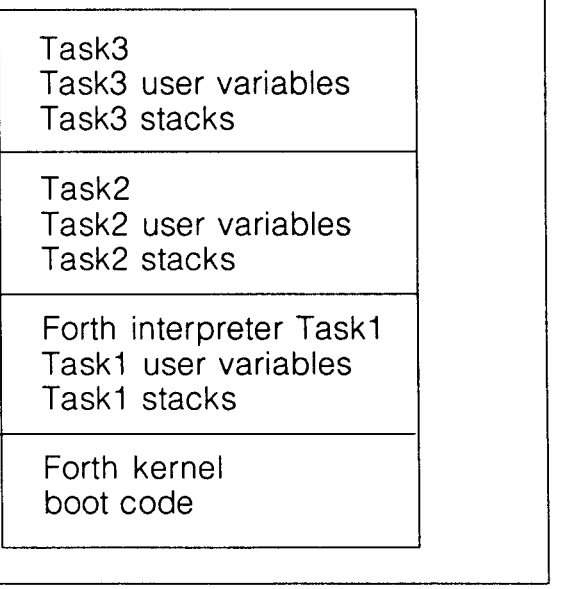
In the protected-mode OS example, the shared memory is logically separated using memory-protection hardware or OS services. Often, only a small amount of memory need be global or shared. This is a form of parallelism, as tasks may logically execute in parallel even though they are still physically time-sharing the CPU. In the case task2 and task3 are copies of the same task, just operating on different data, they could share the same memory for code but would still require a local (protected) memory for stacks and user variables.

Coupled Multiprocessors

Multiprocessors do not really have "central" processors; instead, they have multiple processors. The proces-

The number of operations to extend Forth to a parallel-programming language has been reduced from five to two.

Figure One. Cooperative multitasker memory map.



sors on these machines can be tightly coupled or loosely coupled. Tightly coupled machines use memory that is physically shared, so that part or all of the memory available to a processor is also available to other processors. Figure Three shows a multiprocessor with four processors and a shared memory.

The advantage of the tightly coupled design is that memory is physically shared, so access may be very fast and, in the example in Figure Three, all of the memory is available to each processor. In the case of the Cray II memory interface, there are four sections of memory that may be accessed simultaneously. So only when processors access a region that another processor is using will bus arbitration be needed. The disadvantages of the tightly couple design are the high cost and the physical constraints on the hardware interconnect at the memory-access level.

In Figure One, most memory is shared and each task has some local memory. In Figure Two, each task has a separate memory space, but some memory is shared for global variables. In Figure Three, each processor can have its own memory space, and shared memory is available to all processors. In Figure Four, a system with a networked ring of four computers is shown with the memory physically distributed across the four machines.

The arrow indicates a network interconnect between machines. In this case, a ring is depicted, but other topologies are possible. The memory in each of the machines is physically separate, but shared memory can easily be simulated in software using the network. The Distributed Shared Memory, or DSM, is just a portion of memory on each machine that is identical on all machines. Physically, the memory on these machines is separate, but logically it can be shared. To be DSM, it must be written to via an OS service. This service will actually update all of the memories on the computers on the network, so that all machines have their own copy of this global, shared memory. This portion of memory that is declared global is duplicated on each machine on the network. Since there is a local copy of this memory on each machine, there is no need to access the network to read from

Figure Two. Protected-memory OS multitasking memory map.

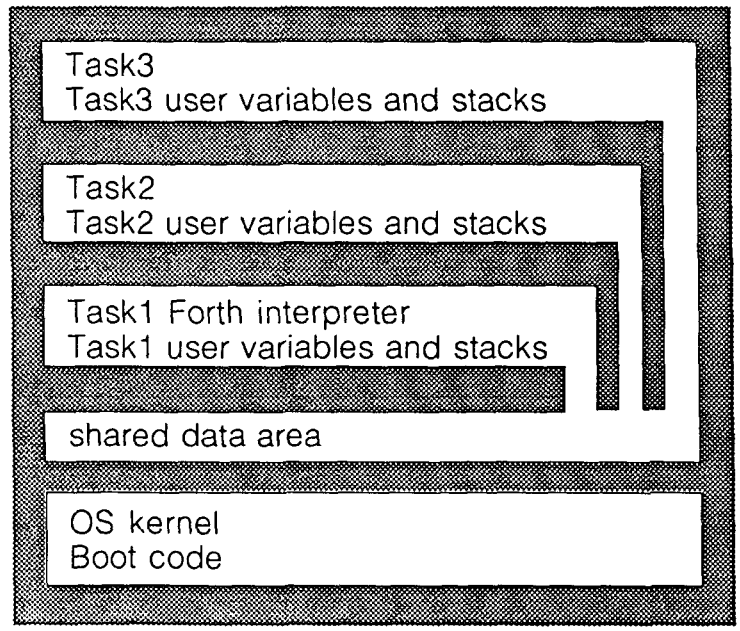


Figure Three. Four-processor, shared-memory multiprocessor.

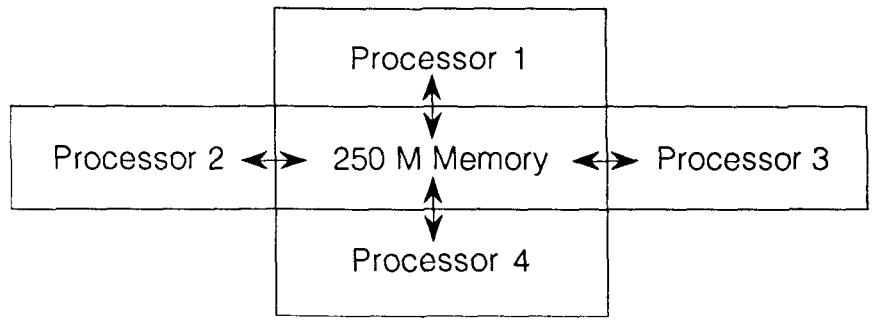
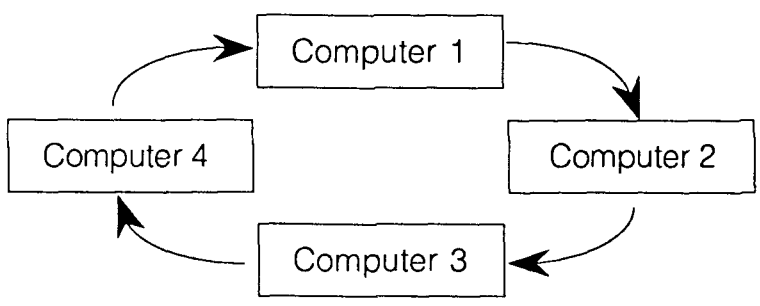


Figure Four. Four computers on a network ring.



this memory. However, writes to this memory must be performed via an OS service that guarantees atomic access to this memory.

Forth-Linda

Forth-Linda provided a "Linda" extension to the Forth language. Linda provides access to a "tuple" space, which is a form of DSM. In Linda, global data is written to and read from the DSM in passive tuples. These tuples have neither a name nor an address, but are accessed by a description of the tuple. In Linda, programs are executed remotely on other machines via active tuples.

Forth-Linda was much simpler than conventional Linda implementations for several reasons. Forth-Linda was intended for homogeneous environments, while Linda is actually designed for the more general-purpose, heterogeneous computing environments. Forth-Linda was also intended for a single network protocol, as well. Thus, Forth-Linda was really intended for the simpler case of symmetric-multiprocessing.

Forth-Linda was implemented on a simulated multiprocessor using multitasking, and experiments were done on a Novell network. Forth-Linda only needed five operations:

`EVAL( string )`	Remote execution of "string" somewhere
`RD( tuple_description )`	Read tuple matching description
`OUT( tuple_description )`	Write tuple or create tuple from description
`RM( tuple_description )`	Read and Remove tuple from description
`RQ()`	Request an active tuple to execute

Parallel Channels

Dr. Michael Montvelishsky published a parallel-programming extension to Forth based on a combination of Forth-Linda and OCCAM. This wordset takes advantage of the concept of synchronizing multiprocessing execution through data channels, as in OCCAM.

Dr. Montvelishsky kept the active tuple concept from Forth-Linda, but replaced the cumbersome passive tuples for data exchange with parallel channels. The parallel-channel wordset was published in *FD* in 1994 in a fairly portable form, and has been optimized for several Forth systems.

F*F

F*F is the name of the Forth parallel-programming extension that has evolved from Forth-Linda. F*F is even simpler than Forth-Linda, because passive tuples have been replaced with an atomic, network-global store. The implementation of F*F requires the extension of network services to provide a remote program execution queue similar to Forth-Linda, but simpler. This is `RX( string )`. Access to the DSM is provided by `G!`. `G!` acts like a normal Forth `!` in that it writes data to an address, but `G!` must write to the distributed memory of all the machines on the network. In fact, there will be provision for declaring a subset of machines on the network to be a group, and for broadcasting directly to this group of machines with a single transmission. The use of `G!` rather than passive tuples also simplifies the implementation. F*F is also a simpler and easier to use programming environment than Forth-Linda. The minimal number of operations needed to extend Forth to a parallel-programming language has been reduced from five to two. F*F will also include network tools and Dr. Montvelishsky's parallel-channel wordset.

Ultra Technology is considering an implementation of F*F under a portable Forth in C, and using TCP/IP across the Internet to deliver supercomputing power to certain applications.

In Forth-Linda or F*F, one of the machines on a network runs as a master and manages the queue of tasks and passive tuples or global-memory writes. Then, many machines on this network run identical copies of F*F. This is shown in Figure Five.

F21 DSM Hardware

Ultra Technology is developing an inexpensive, high-performance microprocessor called the F21. F21 will demonstrate that minimal hardware is needed to implement the network interface upon which DSM may be built. The network interface on F21 will provide two hardware services. A CPU interrupt or a DMA transfer may be sent to a machine or a group of machines on the F21 ring. The serial-network interface on F21 will only require a few transistors to implement, will add only a few cents to the cost of the chip, and should perform one to two orders of magnitude faster than Ethernet. F*F will be very simple to implement on F21, as most of the function of `RX( )` and `G!` will be performed by hardware. Figure Six is a function diagram of the F21 chip.

F21 Status

F21 is still being designed by Charles Moore, the inventor of Forth. Ultra Technology plans to prototype F21 in .8 micron CMOS VLSI technology at MOSIS around the date of this article's publication. Volume production should follow later in the year.

Jeff Fox programmed voice-recognition systems, digital video, legal transcription, office automation, and telecommunication applications before he learned Forth in 1978. Since then, he has done 3D interactive games, expert systems, compilers, assemblers, and other general-purpose tools in Forth. After a ten-year consulting contract with Pacific Bell's training department, he has focused on Chuck Moore's technology and the development of a custom VLSI chip for parallel processing and multimedia. For a couple of years, he has been working with Dr. Montvelishsky on optimizing and parallelizing compilers for Forth machines, and is also working on AI in Forth, using a combination of expert systems, neural networks, and a logic-reduction engine based on the Laws of Form. Jeff also has more than thirty years' experience in martial arts, and studies and teaches Aikido.

He says, "The parallel-processing community has a strong prejudice against Forth, and it seems there is little interest in the Forth community for parallel processing. I hope F21 and the chips that follow it will find a niche and some interest." Fox likes to talk and give presentations, and to spout off on the net about complexity in hardware and software.

The author can be contacted via his jfox@netcom.com address or via http://www.dnai.com/~jfox on the Web.

Figure Five. Forth DSM in Forth-Linda or F*F.

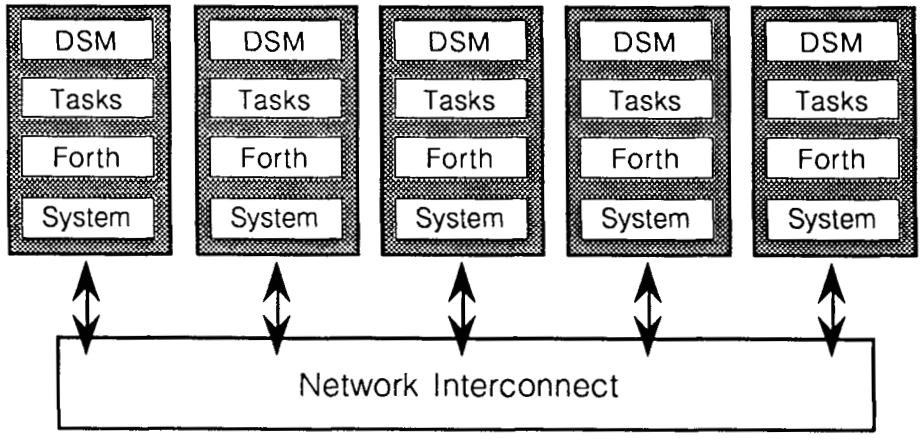
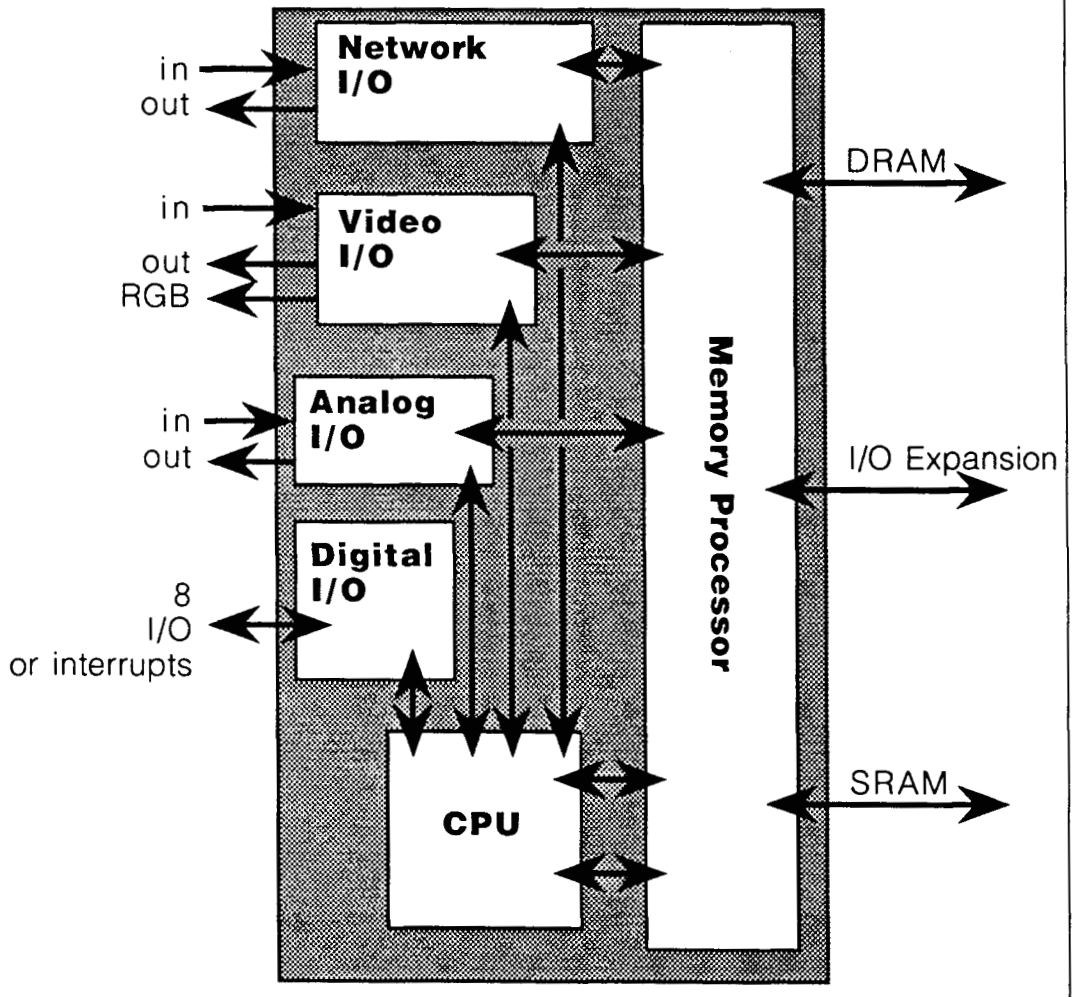


Figure Six. Functional diagram of F21.



Forth Link to C Subroutines

Michael Christopher
Dayton, Ohio

A "C" program in our beloved Forth publication? Rest easy, it is only enough C to allow us to use Forth in those cases where it would be impractical because we *must* link to existing C subroutines.

This article describes a technique for using C subroutine libraries with Forth. It was born out of the need to use—within a Forth program—existing C routines that came with a nine-track tape system.

This "linking" is done using software interrupts. This technique is not really new, IBM used it as a core technology when they created the PC! Software interrupts are familiar to most PC programmers; calls to DOS (interrupt 21H) and BIOS (int 10H) are made using them. The ability to "plug in" new behavior for a specific interrupt has made the PC adaptable as different I/O devices were added to the original PC realm: networks, mice, and so forth were all added using additional software interrupts.

For example, the mouse interrupt can easily be called from Forth, as Tom Zimmer's F-PC system so aptly demonstrates. The example I will show uses his Forth, but this technique can be adapted easily to any other Forth you might have that can call interrupts.

Most manufacturers of PC adapter cards provide a set of C-callable routines to access their cards. For example, a GPIB card would have routines to initialize the card, as well as read/write routines. Using the technique I will now describe, you could easily use these functions from Forth.

To use this method, two programs must be written. The first is a C program that is installed as a Terminate and Stay Resident (TSR) software interrupt handler. The second program is the Forth program with routines that invoke the software interrupt, passing parameters between the two programs as needed.

The C Part

I used Borland's 3.1 C/C++ compiler for this example. This C program will install an interrupt handler for Interrupt 68H. This interrupt is typically unused, although you might need to change it for your application.

Listing One shows all that is required to make a TSR program to "house" the C routines.

The Forth Part

The Forth program required to call the C routines will vary according to the number and type of C routines that are used. In this example, it is quite simple.

See Listing Two: that is what the Forth program must be to allow calling the C routines.

Using This Example

1. Compile the C program using Borland C:

```
c:\> BCC -ml ARTICLE.C    // Note the
argument is an 'el' not a 'one'
```

2. Load the TSR:

```
c:\> ARTICLE
```

3. Start F-PC:

```
c:\> f
```

4. Load the Forth sequence file:

```
fload article.seq
```

5. Test it:

```
c_sound          \ you'll hear some noise
c_quiet          \ it will go away
99 c_format_integer type
                 \ will display: 000099
bye              \ exit Forth
```

Minimizing TSR Memory Usage

You can use some public-domain utilities to remove the TSR after usage. I use MARK.COM and RELEASE.COM. This way, the TSR is only present when needed by the Forth program that uses it.

Conclusion

This technique has been very useful for me. I hope it finds a use in your toolbox. I have placed the source code shown in this article in the Forth section of the GEnie dial-up service under the name LINKTOC.ZIP.

Michael Christopher currently manages the Embedded Software Department for Ohio Electronic Engravers and lives with his wife Cindy and two very fat cats in Dayton, Ohio. He has loved and used Forth for 14 years. His largest Forth accomplishment was using it to create the page-description language for the world's fastest laser (-like) printer. It prints 300 two-sided pages per minute. He can be reached at savvyside@aol.com.

Listing One. TSR program in C.

```
This Borland C 3.0+ program will create a terminate and stay resident program.

   Michael Christopher January 8, 1995
   This program is released to the public domain.
*/

/*
This will install itself as Interrupt 68 HEX. Change it if needed.

   Use the ARTICLE.SEQ Forth program (F-PC 3.5 or later) to call this set of
   C subroutines.

   These routines use simple C library calls but they can easily be adapted
   to a commercial library.

   It can be tricky to test routines accessed via a TSR. I usually
   test the routines using a standard C program stub shell first. That way
   you can easily debug them using the normal debugger before adding them to
   this TSR shell.
*/

/*

   To build this software, use theses Borland Commands:
   (from DOS command line)

   BCC -ml ARTICLE.C  // NOTE: the argument is an 'el' not a 'one'

   Ignore the warnings about unused registers. Those warnings are NOT errors
   in this case.
*/


/* include some standard C header files. */
#include <stdio.h>
#include <string.h>
#include <conio.h>
#include <dos.h>


// this sets the interrupt vector we will hook into
#define HOOK_VECTOR 0x68

/* Global Variables */
int paragraphs_to_keep;       /* number of 16 byte paragraphs to reserve */
char test_string[300];        /* a temp string buffer */
unsigned char far *temp_ptr;  /* a pointer used to access a string */

/*
   The following function is what is called by Forth.
   This interrupt routine is installed for interrupt number 0x68 HEX.
   It serves to provide a software interrupt service for accessing some C
   functions in a way similar to accessing (INT 21H) DOS functions.
*/

void interrupt my_handler (
     unsigned bp,  unsigned di, unsigned si,  unsigned ds,
     unsigned es,  unsigned dx, unsigned cx,  unsigned bx,
     unsigned ax,  unsigned ip, unsigned cs,  unsigned flag
     )
```

```
    /*
    These parameters are pseudo-registers that allow access to the
    processor state that existed when the interrupt instruction was issued.
    The contents of these registers will be stuffed back into the real
    registers when this routine returns, so you can pass values back also.

    They are used to get and give parameters to the Forth program that
    invokes this SOFTWARE interrupt service routine.

    Borland C allows getting the CURRENT value of a register
    using the _DS (etc.) command. That can be useful for passing back
    the segment of data structures used in the TSR to the calling routine.

    The usage of the registers used to as in and out parameters are
    user defined. These simple examples all assume that the AX register has
    the function number you wanted.
*/

    {
    unsigned int i; // temp integer variable used as a loop index

    switch (ax)
      {

      // this function will make a sound
      case 0:
        ax = 0;           // pass back some status
        sound(300);       // call a C library routine
        break;

      // this function will stop the racket
      case 1:
        ax = 1;           // pass back some status
        nosound();        // call a C library routine
        break;

      case 2:
       // this will format a 16 integer into a 6 digit string with leading
       // zeros.

       // INCOMING PARAMETERS:

       //  BX - 16 bit integer to be formatted
       //  CX - Segment where to store the newly formatted string
       //  DX - offset of same

       // make sure the string is large enough to store the new string


       // create a pointer into the area where we were told to place the new
       // string.
       temp_ptr = (unsigned char far *) MK_FP( cx, dx);

       // format an integer argument in BX register into a string.
       sprintf(test_string, "%06d", bx);

       // store the result as a counted string for Forth

       // store the length of the return string in first byte
       *temp_ptr = (unsigned char) strlen(test_string) ;
```

```
      // now store the string contents after the count byte
      // repeat as many times as the count byte indicates
      for (i = 0; i < (int) *temp_ptr; i++) {
          *(temp_ptr+i+1) = test_string[i];
          }
      break;

     case 13:
      /* this function can be used to check if the tsr is installed */
      /* if installed, ax will return as -13. */
       ax = -13;  /* tsr installed flag*/
       break;

     // some unknown function was called
     default:
       ax = 0;
       break;
     }
}

void main(void)
   {
   // clear the screen
   clrscr();

   // set the interrupt vector to point to our routine
   setvect( (unsigned) HOOK_VECTOR, my_handler);

   printf("\nForth to C interface. V 1.00 ");

   // this sets the memory to be reserved in 16 bit paragraphs.
   paragraphs_to_keep = 4096;

   // install as a TSR
   _dos_keep(0,paragraphs_to_keep);
}
```

Listing Two. Forth program to call the C routines.

```
comment:
     Article.seq --- F-PC v 3.5 or later Forth source file
     Testing the Forth to C interface example.
     MDC   Jan 8, 1995
     This program is released to the public domain.
comment;

postfix

\ make a string for use for testing
create teststring 250 allot

\ this is the PC interrupt # that the C routine is accessed via
\ This may be changed to any UNUSED interrupt.
\ This one is safe according to my references.
$68 constant HOOK_INTERRUPT

\ The following word will load ax with a function to be performed.
```

```
\ It can be changed as needed. Just make sure that both sides of the
\ interface (the ISR and this program) expect arguments in specified
\ registers.
code simple_call ( func# - return_code)
    ax pop                  \ get the function the ISR will perform
    HOOK_INTERRUPT int      \ and call the ISR
    ax push                 \ return a status
    next                    \ do the next forth word....
    c;

\ This is an example of how you can perform a more complex call such as
\ subroutines that require that addresses be passed as parameters.

\ This subroutine takes an integer, and a segment:offset where we will
\ store the resulting Forth string.

\ input parameters to the C routine
\     BX integer to format
\     CX - Segment of a Forth style string at the address n
\     DX - offset              "

\ output parameters from the C routine is the modified string in given address
\ and a return code in ax

code complex_call ( segadr ofsadr integer func# - return_code)
    ax pop                      \ function number
    bx pop                      \ integer
    cx pop                      \ string segment
    dx pop                      \ string offset
    HOOK_INTERRUPT int          \ call the interrupt
    cx push                     \ the segment
    dx push                     \ the offset
    ax push                     \ the status
    next
    c;

\ This Forth word will test to see if the C TSR is already installed
: IsItInstalled? ( - )
    cr ." The interrupt is "
    13                  \ function to perform is # 13,"IS THE TSR INSTALLED?"
    simple_call         \ call ISR written in c.
    -13 <> if           \ if it doesn't return -13, it is not installed
        ." not "
    then
    ." installed." cr
    ;


\ make a beep sound by calling a C function in the C TSR
: C_beep ( -  )
    0                   \ function # 0, "BEEP!"
    simple_call
    drop                \ return status is meaningless, drop it from the stack
    ;

\ Turns off the darn racket!
: C_quiet ( -  )
    1                   \ function # 1, "QUIET"
    simple_call
    drop
    ;
```

```
\ This calls a function that is NOT available in the C TSR.
\ Its purpose it to show the behavior for an undefined routine. It should
\ return a 0
: testFunction99 ( - n )
    99 simple_call
    ;


\ Format a 16 bit integer according to a format specified in the
\ C program (6 digits with leading zeros).
\ For even more flexibility, you could pass a zero terminated string
\ (a C style string) that would specify the format to use in sprintf style.
\ I'll leave that as an exercise!
: C_format_integer ( n - a n )
    >r                          \ save integer to format on the rstack
    teststring                  \ offset of string to store formatted data
    ?CS:                        \ segment of string to store formatted data
    r>                          \ retrieve integer to format from rstack
    2                           \ function number, "Format an integer"
    complex_call
    drop                        \ the status returns are not used
    drop
    drop
    teststring count            \ return the formatted string
    ;
```

Yet Another Interpreter Organization

Mitch Bradley
Mountain View, California

Editor's note: This paper represents work the author did many years ago, in the context of the systems of that time; it does not represent his current thinking in all details.

There has been a mild controversy in the Forth community about how to implement the text interpreter. The particular problem is how the distinction between compiling and interpreting should be coded. At least three distinct solutions have been advocated over the years. I propose a fourth one, and claim that it is the best solution yet.

fig-Forth Solution

fig-Forth used a variable STATE whose value was zero when interpreting and (hex) C0 when compiling. The interpreter was coded as a single word INTERPRET which tested STATE to determine whether to compile or to interpret. Here is the code:

```
: INTERPRET ( -- )
  BEGIN  -FIND
    IF   STATE @  <
      IF  CFA , ELSE  CFA EXECUTE  THEN
    ELSE   HERE NUMBER  DPL @  1+
      IF   DROP [COMPILE] LITERAL
      ELSE      [COMPILE] DLITERAL
      THEN
    THEN  ?STACK
  AGAIN
;
```

The STATE @ < phrase is pretty clever (or disgusting, however you wish to look at it). Since the value stored in STATE is (hex) C0 when compiling, and since the length byte of a defined word (which is left on the stack by -FIND) is in the range (hex) 80–BF for a non-immediate word and in the range (hex) C0–FF for an immediate word, the STATE @ < test manages to return *true* only if the STATE is compiling and the word is not immediate. This fact is not salient to our discussion, but is included here to prevent confusion.

STATE is explicitly tested once inside this loop, but if you look at the code for the word LITERAL, it too tests STATE to decide whether to compile the number or not.

To switch between compiling and interpreting, fig-Forth uses the two words [and]. [is immediate and simply stores zero into STATE.] is not immediate and stores (hex) C0 into STATE. Compilation is typically started with : (colon), which is defined something like:

```
: :
  <some irrelevant stuff>
  ] ;CODE
  <some assembly language stuff>
END-CODE
```

The important point here is that when : executes to define a new word, the] just sets the STATE to compiling, then the ;CODE proceeds to execute. (The purpose of ;CODE is to patch the code field of the word defined by : so that it does the appropriate thing for a high-level Forth word.) The interpret word INTERPRET doesn't notice that STATE is now compiling until the ;CODE finishes.

So we see that [and] are pretty innocuous; they just change the value of a variable.

polyFORTH Solution

Forth, Inc. decided it would be better to have two separate loops for the two separate functions of compiling and interpreting. The compiling loop was called], so] actually executed the compile loop directly, rather than just setting a variable. This has two subtle side effects.

If you loop at the previous definition of : and now pretend that, instead of just setting a variable,] actually executes the compiler loop, you will see that the ;CODE following it doesn't actually get executed until *after* the compiling is finished. This, in itself, doesn't cause a problem for :, but the use of] inside programmer-defined words sometimes caused unexpected behavior because stuff after the] would get executed after a bunch of stuff had been compiled.

The other subtlety relates to how the loops are terminated. Note that the INTERPRET loop shown above never terminates! We all know that it really does terminate, and the mechanism is pretty kludgey. What happens is that there is a null character at the end of every line of text in the input stream, and at the end of every BLOCK of text from mass storage. The text interpreter picks up this null character just like a normal word. The dictionary contains

an entry which matches this "null word." The associated code is executed, and it plays around with the return stack in such a way that the INTERPRET loop is exited without ever knowing about it.

The problem with the dual-loop interpreter/compiler is that the end of each line of input from the input stream kicks our system out of whichever loop it was in. If the user is attempting to compile a multi-line colon definition from the input stream, he must start each line after the first with an explicit] because, once the compiler loop is exited at the end of the first line, the system doesn't remember that it was compiling.

One key thing to remember is that the compiler loop (which was named [) is executed from within the interpreter loop.

Coroutines (Patton/Berkey)

At FORML '83, Bob Berkey presented a paper about using coroutines for the interpreter loop and the compiler loop, instead of having the compiler loop run inside the interpreter loop. This means that executing] kicks out the interpreter loop and runs the compiler loop instead; similarly, executing [kicks out the compiler loop and runs the interpreter loop instead. The subroutine versions of these loops are present in his scheme, named COMPILER and INTERPRETER.

Bob feels this scheme is more symmetrical than the polyFORTH approach, and that it eliminates some of the counter-intuitive behavior.

This scheme still requires that multi-line colon definitions compiled from the keyboard have a] at the beginning of each line after the first.

What is Wrong With All This

These different schemes do not at all address what I consider to be the fundamental problems with the interpreter/compiler.

Fundamental Problem #1:

The compiler/interpreter has a built-in infinite loop. This means you can't tell it to just compile one word; once you start it, off it goes—and it won't stop until it gets to the end of the line or screen.

Fundamental Problem #2:

The reading of the next word from the input stream is buried inside this loop. This means you can't hand a string representing a word to the interpreter/compiler and have it interpret or compile it for you.

Fundamental Problem #3:

The behavior of the interpreter/compiler is hard to change, because all the behavior is hard-wired into one or two relatively large words. Changing this behavior can be extremely useful for a number of applications—metacompiling, for example.

Fundamental Problem #4:

If the interpreter/compiler can't figure out what to do with a word (it's not defined and it's not a number), it aborts. Worse yet, the aborting is not done directly from within the loop, but inside NUMBER. This severely limits the usefulness of NUMBER because, if the string NUMBER gets is not recognizable as a number, it will abort on you. (The Forth-83 Standard punted on this issue by not specifying NUMBER except as an uncontrolled reference word.)

Solution

As I see it, several distinct things are going on inside the interpreter/compiler. A proper factorization of the interpreter/compiler into words which each do one thing solves all these problems.

The outermost thing is the loop. The loop's job is to repetitively get the next word from the input stream and do something with it. The loop should terminate when the input stream is exhausted. [See Figure One.]

The next level down is the "do something with it." This ought to be a separate word so that it may be called by other words which would like to compile/interpret a single word. This layer is here called "COMPILE because it takes a string representing a single word and compiles (or interprets) it. "COMPILE's main job is to decide what kind of word it is dealing with. There are three choices: Either the word is already defined, or it is a literal (i.e., a

Figure One. Terminate loop when input is exhausted.

```
: NEW-INTERPRET  (S -- )
  BEGIN    BL WORD     ( str )
           MORE?       ( str f )
                       ( flag true if input stream not exhausted )
  WHILE
           "COMPILE
  REPEAT
  DROP
;
```

Figure Two. Compile (or interpret) a string.

```
: "COMPILE ( str -- ?? )
  FIND      ( str 0  |   cfa -1   |   cfa 1 )
  DUP
  IF   DO-DEFINED     ( ??  )
  ELSE DROP           ( str )
       LITERAL?       ( str false | ?? true )
       IF    DO-LITERAL         ( ?? )
       ELSE  DO-UNDEFINED       ( ?? )
       THEN
  THEN
;
```

number), or it is neither. [See Figure Two.]

Finally, at the lowest layer, is the code which does the appropriate thing for each of these three possibilities. This level is represented by the words DO-DEFINED, DO-LITERAL, and DO-UNDEFINED. It is *only* at this lowest layer that the system cares at all whether it is compiling or interpreting. One of the benefits claimed for the polyFORTH scheme is speed. This is due to the elimination of tests of the STATE variable within the loop.

Clearly, my scheme has to do something to distinguish between compiling and interpreting. An obvious solution would be to test STATE inside each DO-DEFINED, DO-LITERAL, and DO-UNDEFINED. This would slow the system, of course.

A more interesting alternative is to make each DO-DEFINED, DO-LITERAL, and DO-UNDEFINED a deferred word. (Deferred words are sometimes called execution vectors. Basically, they are like variables which hold the address of a word to execute, except that the @ EXECUTE is done automatically.)

If these words are deferred, they can be changed when the system goes from compiling to interpreting, and vice versa. [See Figure Three.]

Then [and] would be defined as in Figure Four. (IS is the word which sets the word to execute for a deferred word.)

Executing a deferred word need not be slow. Deferred word are so useful that they should be coded in assembler for speed. On my system, they are only very slightly slower than normal colon definitions.

Figure Three. Deferred versions of key words.

```
DEFER LITERAL?       ( str -- n true  |  d true  |  str false )
DEFER DO-DEFINED     ( cfa -1 | cfa 1  -- ?? )
DEFER DO-LITERAL     ( literal -- ?? )
DEFER DO-UNDEFINED   ( str -- )

: (LITERAL?  ( str -- str false  | literal true )
  >R R@ NUMBER?    ( l f )
  IF     R> DROP  TRUE
  ELSE   DROP R> FALSE
  THEN
;
' (LITERAL? IS LITERAL?
: INTERPRET-DO-DEFINED  ( cfa -1 | cfa 1 -- ?? )
  DROP EXECUTE
;
: COMPILE-DO-DEFINED     ( cfa -1 | cfa 1 -- )
  0> IF   EXECUTE    ( if immediate )
     ELSE ,          ( if not immediate )
     THEN
;
: INTERPRET-DO-LITERAL ( d -- d | n )
  DOUBLE? 0= IF DROP THEN
;
: COMPILE-DO-LITERAL ( d -- )
  DOUBLE? IF [COMPILE] DLITERAL ELSE [COMPILE] LITERAL THEN
;
: INTERPRET-DO-UNDEFINED ( str -- )
    COUNT TYPE ." ?" CR
  QUIT
;
: COMPILE-DO-UNDEFINED   ( str -- )
    COUNT TYPE ." ?" CR
  COMPILE LOSE
;
```

Figure Four. The new [and].

```
: [
  ['] INTERPRET-DO-DEFINED   IS DO-DEFINED
  ['] INTERPRET-DO-LITERAL   IS DO-LITERAL
  ['] INTERPRET-DO-UNDEFINED IS DO-UNDEFINED
  STATE OFF
; IMMEDIATE

: ]
  ['] COMPILE-DO-DEFINED     IS DO-DEFINED
  ['] COMPILE-DO-LITERAL     IS DO-LITERAL
  ['] COMPILE-DO-UNDEFINED   IS DO-UNDEFINED
  STATE ON
;
```

So What?

This may seem to be more complicated than the schemes it replaces. It certainly does have more words. On the other hand, each word is individually easy to understand, and each word does a very specific job, in contrast to the old style, which bundles up a lot of different things in one big word. The more explicit factoring gives you a great deal of control over the interpreter.

Following are some interesting things you can do with this new scheme. One of my favorite words is TH (for Temporary Hex):

```
: TH   ( --word  ?? )
  BASE @ >R  HEX
  BL WORD "COMPILE
  R> BASE !
; IMMEDIATE
```

This word temporarily sets the base to hexadecimal, interprets a word, and restores the base. It works for numbers or defined words, either interpreting or compiling.

For example:

```
DECIMAL
TH 10 .   ( system prints-->  16
10 TH .   ( system prints-->  A
: STRIP-PARITY
    ( char -- char-without-parity )
  TH 7F AND
;
```

Liberal use of this word markedly reduces the need to switch bases, especially in source code, and thus reduces the chance of errors.

[Figure Five shows] a common word that is trivial to implement with this kind of interpreter:

Here's a word [Figure Six] which allows you to make a new name for an old word. It is smart, in that when the new word is compiled, the old word will actually be compiled instead, eliminating any performance penalty. Furthermore, it even works for old words that are immediate! As you will see, the vectored DO-DEFINED does exactly the thing we want.

Finally, [Figure Seven gives] a really neat way to write keyword-driven translators. Suppose you have some kind of file that contains a bunch of text. Interspersed throughout the text are keywords that you would like to recognize, and the program should do something special when it sees a keyword. For things that aren't keywords, it just writes them out unchanged. Suppose the keywords are .PARAGRAPH, .SECTION, and .END.

I have used this technique very successfully to extract specific information from database files produced by a CAD system. Instead of outputting unrecognized words, I actually just ignored them in this application—but the technique is the same in either case.

Figure Five.

```
: ASCII (  --name  char )
  BL WORD 1+ C@ ( char )
  -1 DPL !  \ make sure it's not
            \ handled as a double number
  DO-LITERAL
;
```

Figure Six.

```
: ALIAS  ( -- )   ( Input stream:  new-name old-word)
  CREATE
    BL WORD FIND  ( cfa -1 | cfa 1  |  str false )
    DUP IF
       , , IMMEDIATE
    ELSE
      DROP  ." Can't find "  COUNT TYPE
    THEN
  DOES>  2@   ( cfa -1 | cfa 1 )
   DO-DEFINED
;
( Examples )
ALIAS D@ 2@
HERE D@   ( actually executes 2@ )
: FOO HERE D@ ;  ( actually compiles 2@ )
ALIAS FOREVER  AGAIN
: LOOP-ALWAYS BEGIN FOREVER ;
( actually executes AGAIN, which is immediate )
```

Figure Seven.

```
VOCABULARY KEYWORDS DEFINITIONS
: .PARAGRAPH
  ( whatever you want to happen when you see paragraph )
;
: .SECTION
  ( whatever you want to happen when you see paragraph )
;
: KEYWORDS-DO-UNDEFINED ( STR -- )
  COUNT TYPE
;
: .END
  ONLY FORTH
  ['] (LITERAL?                IS  LITERAL?
  ['] INTERPRET-DO-UNDEFINED IS  DO-UNDEFINED
;
ONLY FORTH ALSO KEYWORDS
: PROCESS-KEYWORDS
  ['] FALSE                    IS LITERAL?
  ['] KEYWORDS-DO-UNDEFINED  IS DO-UNDEFINED
  ONLY KEYWORDS
;
```

Mitch Bradley is President of FirmWorks, a company specializing in products and services related to Open Firmware. Open Firmware, defined by IEEE Standard 1275–1994, is a processor-independent, bus-independent architecture for boot firmware. Open Firmware is based on ANSI Forth, and its standard user interface is a Forth interpreter. Open Firmware is currently used on over a million SPARC workstations, and has been selected as the standard firmware for PCI-bus PowerPC systems, including Apple's new PCI-bus-based systems.

http://www.firmworks.com

ftp://ftp.firmworks.com

Case Cookbook

A Study of the CASE Statement

Walter J. Rottenkolber
Mariposa, California

Newcomers to Forth are sometimes dismayed by its primitiveness. It's like buying a car and receiving a crate of parts with the instructions, "Some assembly required." By design, Forth is more a mechanism for being extended into the new structures and functions required to solve a problem, than a language in the traditional sense.

Most languages come with an extensive list of preordained data types and keyword functions. Programming consists of forcing the problem onto this list, and of working around their bugs and limitations.

Over a decade ago, there arose the great Case Controversy, an attempt to expand Forth with a more familiar structure, and a tribute to Pascal's popularity. The Forth Interest Group sought to calm heated passions by sponsoring the great Case Contest. It sparked several Forth versions, and some 20 were published in volume two of *Forth Dimensions* alone. Most were variations on a theme, either of the CASE syntax or of improvements to the original. More importantly, they demonstrated the means for extending Forth to generate your own version. So, for the Forth beginner, I present the Case Cookbook.

Selectors

The ability of a program to choose alternate pathways is what turns a calculator into a computer. The simplest mechanism is the branching statement. Think of the IF THEN branch as a single-function selector, and of the IF ELSE THEN branch as a two-function selector.

The CASE statement (and its alter ego in C, *switch*) is simply a clearer format for selecting among more than two options. It's important to realize that other methods can also act as selectors; namely, jump tables and execution tables.

Branches and the Case Statement

Branches can be extended to select more than two functions. The accompanying screens show a number of branching arrangements, as well as two CASE words derived from them.

B1 uses multiple IF THENs. The disadvantage here is that, even after a function is selected, the remainder of the comparisons have to be made. This wastes time.

B2 nests IF ELSE THENs, causing the program to jump to the end of the word after a selected function is completed. But now we have all those THENs at the end. Not so bad, unless you're writing an editor and have 20 selections. Also, the DROP at the end (to eliminate the duplicated test value) will interfere with data left on the stack.

B3 has OVER duplicate the test value, instead of DUP. DROP has been moved to just after IF and the final ELSE. It's a bit more complicated, but it eliminates interference with stack data. Also, we now have OVER = IF DROP grouped together. This makes it simpler to duplicate the function of these words with a single code primitive.

Screen seven lists Dr. Charles Eaker's CASE statement—the winner of the Case Contest—in the original fig-Forth. Some enhancements suggested by Alfred Monroe are in screen eight. He used the primitive (OF) to reduce the amount of code compiled by OF. The ?PAIRS word was fig-Forth's way of checking syntax.

The enhanced version, modified for Forth-83, is in screen three. This became popular because it not only mimicked the Pascal CASE, but extended its capability to include greater-than, less-than, and range comparisons.

CASE puts the contents of CSP on the stack. Then, it saves the current data stack pointer (SP@) in CSP. Now you can track the growth of the data stack as the ELSEs leave the flag and address to be resolved. ENDCASE then uses the current and starting data stack pointers to determine the end point for a WHILE loop that generates multiple THENs. CSP is restored from the value on the stack. The end result is a code structure resembling B3, although its complexity is hidden from the programmer. The branch words are the primitives used by IF, ELSE, and THEN.

B5, in screen six, uses the word EXIT to solve the endless THEN problem. This word works like the opcode RETurn in assembler, and causes the program to exit the word at that point. Wil Baden used it to design a flexible CASE that allowed for a variety of comparison operations (screen four). This code works just fine as it is, but it doesn't *look* like a CASE statement.

Screen five shows a better version—I merged Dr. Eaker's and Wil Baden's CASE statements. By ignoring the COMPILE words, you can see that it generates the code in B5. By using [COMPILE], we can now compile IF itself, instead of its components as in the original Eaker code. I moved several words into (RANGE), trading a lower number of words compiled by RANGE for a slight loss of speed. If need be, they can be returned to inline code. An example of its use is C2 (screen six).

Comparing the code in B5 and C2 shows why the CASE statement is popular. It demonstrates the idea of abstraction, that is, gathering a group of functions into a single word. Incorporating OVER = IF DROP into OF eliminates the clutter of the underlying machinery. You can more easily focus your attention on the selector value and its <code> response.

Other CASE statements were proposed. One that achieved some attention was developed by Neptune UES for their proprietary Forth-85. Its syntax was:

```
DO-CASE   ( n )
  n' CASE   <code>   END-CASE
  n' CASE   <code>   END-CASE
  n' CASE   <code>   END-CASE
END-CASES
```

However, Eaker's version won out because it was more familiar to users.

Compiler Words

For the beginner to Forth, the CASE words demonstrate how compiler words can simplify the syntax of a more complicated function. These words behave like a macro in assembler or *define* in C. They substitute other words for themselves.

Compiler words use COMPILE and [COMPILE]. When the compiler word containing them is run, they prevent the next word in the parameter list from executing. Instead, it is compiled into the word being defined that called the compiler word. Compiler words can contain other compiler words, i.e., they can be nested.

COMPILE (word) is used for regular words, and [COMPILE] (word) is used for immediate words. The latter is necessary, as immediate words normally run even in compiling mode and have to be deactivated first. Some Forths use the word POSTPONE to combine the functions of both COMPILE words. Since compiler words substitute other words for themselves, they must be immediate words, and must only be used in colon definitions.

Try this experiment. First, compile the words in Figure One-a. Now look at those words with SEE (the Forth disassembler), as in Figure One-b. Notice the difference between T1 and T2.

Because CASE2 is an immediate word, it runs even in compiler mode. Its action is to compile DUP and + into the word being compiled, which is T2. In the new CASE (screen five), by taking the COMPILE words away, you can see what the compiler words generate:

```
OF      -->  OVER = IF DROP
ENDOF   -->  EXIT THEN
```

Wrapping Up

One advantage of the Forth way is that you can extend a structure, such as CASE, to solve new problems. Suppose you want your CASE to have a comparison like (n = n') OR (n = n''). You can develop something like:

```
: (=OR)   ( n n' n'' -- n f )
  2 PICK  TUCK = -ROT = OR ;

: =OR   ( n n' n'' -- |n )
  COMPILE (=OR)  [COMPILE] IF
  COMPILE DROP
  ; IMMEDIATE
```

Now you can write a CASE code line such as:

```
ASCII Y  ASCII y  =OR  <yes-code>  ENDOF
```

A standardized CASE statement is useful in teaching and code sharing. However, I believe you are better served by understanding the data structure and algorithm underlying statements like CASE, as this will enable you to write simpler, more creative code.

References

Charles Eaker, *Forth Dimensions* (II/3), 1980.
Alfred J. Monroe, *Forth Dimensions* (III/6), 1982.
Wil Baden, "Ultimate CASE Statement," *Forth Dimensions* (VIII/5), 1987.

Figure One-a. Define these words in your system.

```
: CASE1   DUP + ;

: T1      2 CASE1 . ;

: CASE2
  COMPILE DUP   COMPILE + ; IMMEDIATE

: T2      2 CASE2 . ;
```

Figure One-b. Disassemble the new words.

```
see t1   -->   : T2   2 CASE1 . ;

see t2   -->   : T2   2 DUP + . ;
```


Walter J. Rottenkolber bought his first computer in 1983. Early on, he experimented with fig-Forth and other languages, but gravitated to assembler until reintroduced to Forth in 1988. He notes that Forth provides the same close-to-the-silicon feeling as assembler, but without the pain. Interests include small embedded systems, programming, and computer history, about which he enjoys writing.


```
      1
 0 \ Branching
 1
 2 : ti$   ." This is " ;
 3 : one   ti$ ." one." ;
 4 : two   ti$ ." two." ;
 5 : three ti$ ." three." ;
 6 : four  ti$ ." four." ;
 7
 8 : B1  ( n)
 9    DUP  1 = IF one   THEN
10    DUP  2 = IF two   THEN
11    DUP  3 = IF three THEN
12    DUP  4 = IF four  THEN
13    DROP ;

      2
 0 \ Branching
 1
 2 : B2  ( n)
 3    DUP 1 = IF one   ELSE
 4    DUP 2 = IF two   ELSE
 5    DUP 3 = IF three ELSE
 6    DUP 4 = IF four  ELSE
 7    THEN THEN THEN THEN DROP ;
 8
 9 : B3  ( n)
10    1 OVER = IF DROP  one   ELSE
11    2 OVER = IF DROP  two   ELSE
12    3 OVER = IF DROP  three ELSE
13    4 OVER = IF DROP  four  ELSE
14    DROP THEN THEN THEN THEN ;
15
```


Code continues on next page...

```
     3
0 \ Dr. Eaker's Case Statement -- for Forth 83
1
2 : CASE    CSP @ SP@ CSP ! ;   IMMEDIATE
3 : OF      COMPILE OVER   COMPILE =   COMPILE ?BRANCH
4           )MARK   COMPILE DROP ;   IMMEDIATE
5 : ENDOF   COMPILE BRANCH   )MARK SWAP )RESOLVE ;   IMMEDIATE
6 : ENDCASE COMPILE DROP BEGIN SP@ CSP @ = 0= WHILE
7     )RESOLVE   REPEAT CSP ! ;   IMMEDIATE
8 : )OF     COMPILE OVER   COMPILE (   COMPILE ?BRANCH
9           )MARK   COMPILE DROP ;   IMMEDIATE
10 : (OF    COMPILE OVER   COMPILE )   COMPILE ?BRANCH
11          )MARK   COMPILE DROP ;   IMMEDIATE
12 : RANGE  COMPILE 2   COMPILE PICK   COMPILE -ROT
13          COMPILE BETWEEN   COMPILE ?BRANCH   )MARK
14          COMPILE DROP ;    IMMEDIATE
15
```

```
     4
\ Branching

: B4  ( n)
    DUP 1 = IF DROP one   EXIT THEN
    DUP 2 = IF DROP two   EXIT THEN
    DUP 3 = IF DROP three EXIT THEN
    DUP 4 = IF DROP four  EXIT THEN
    DROP ;
\S
\ Will Baden's Case Statement
: CASE  DUP ;
: OF    [COMPILE] IF COMPILE DROP ; IMMEDIATE
: WHATEVER  ( n)
    CASE  7 = OF ." You win" EXIT THEN
    CASE 11 = OF ." You win" EXIT THEN
    DROP ;
```

```
     5
\ Dr. Eaker/Baden/WJR Case Statement -- Forth 83
: CASE    ( n - n) ;
: OF      ( n n - | n)  COMPILE OVER  COMPILE =
          [COMPILE] IF  COMPILE DROP ;  IMMEDIATE
: ENDOF   COMPILE EXIT  [COMPILE] THEN ;  IMMEDIATE
: ENDCASE ( n)  COMPILE DROP ;  IMMEDIATE
: )OF     ( n n - |n)   COMPILE OVER  COMPILE (
          [COMPILE] IF  COMPILE DROP ;  IMMEDIATE
: (OF     ( n n - | n)  COMPILE OVER  COMPILE )
          [COMPILE] IF  COMPILE DROP ;  IMMEDIATE
: (RANGE) ( n n n - n f)  2 PICK  -ROT  BETWEEN ;
: RANGE   ( n n n - |n)  COMPILE (RANGE)
          [COMPILE] IF  COMPILE DROP ;  IMMEDIATE
```

```
     6
\ CASE example

: B5  ( n)
    1 OVER = IF DROP one   EXIT THEN
    2 OVER = IF DROP two   EXIT THEN
    3 OVER = IF DROP three EXIT THEN
    4 OVER = IF DROP four  EXIT THEN
    DROP ;

: C2  ( n)
    CASE
    1 OF one   ENDOF
    2 OF two   ENDOF
    3 OF three ENDOF
    4 OF four  ENDOF
    ENDCASE ;
```

```
     7
0 \ Dr. Charles Aker's CASE
1
2 : CASE  ( n)  ?COMP CSP @ !CSP 4 ; IMMEDIATE
3
4 : OF  ( n n' - |n)  4 ?PAIRS COMPILE OVER COMPILE =
5     COMPILE 0BRANCH  HERE 0 , COMPILE DROP 5 ; IMMEDIATE
6
7 : ENDOF   5 ?PAIRS COMPILE BRANCH HERE 0 ,
8     SWAP 2 [COMPILE] ENDIF 4 ; IMMEDIATE
9
10 : ENDCASE  ( n)  4 ?PAIRS COMPILE DROP
11     BEGIN SP@ CSP @ = 0=
12     WHILE 2 [COMPILE] ENDIF REPEAT  CSP ! ; IMMEDIATE
13
14
15
```

```
     8
0 \ Alfred Monroe's CASE Enhancements
1 : (OF)  ( n n' - n f)  OVER = IF DROP 1 ELSE 0 ENDIF ;
2 : OF    ( n n' - |n)  4 ?PAIRS COMPILE (OF)
3     COMPILE 0BRANCH  HERE 0 , 5 ; IMMEDIATE
4 : ((OF)  ( n n' - n f)  OVER ) IF DROP 1 ELSE 0 ENDIF ;
5 : (OF    ( n n' - |n)  4 ?PAIRS COMPILE ((OF)
6     COMPILE 0BRANCH  HERE 0 , 5 ; IMMEDIATE
7 : ()OF)  ( n n' - n f)  OVER ( IF DROP 1 ELSE 0 ENDIF ;
8 : )OF    ( n n' - |n)  4 ?PAIRS COMPILE ()OF)
9     COMPILE 0BRANCH  HERE 0 , 5 ; IMMEDIATE
10
11 : RANGE  ( n n' n'' - n f)
12     )R OVER DUP R) 1+ ( IF SWAP 1- )
13     IF DROP 1 ELSE 0 ENDIF ELSE DROP DROP 0 ENDIF ;
14 : RNG-OF  ( n n' n''- |n)  4 ?PAIRS COMPILE RANGE
15     COMPILE 0BRANCH  HERE 0 , 5 ; IMMEDIATE
```

Forth Vendors

In an effort to increase awareness of, and access to, Forth vendors, the Forth Interest Group is resuming the Forth Vendors List. We are gathering up-to-date information on as many vendors who offer Forth-related products as we can identify. The primary method of gathering data is by sending questionnaires, preferably by e-mail.

It is currently planned to periodically publish a subset of this information in *Forth Dimensions*. The listing below is an example of what we plan to publish, using those who have responded to date. Further use and/or publication of the information is to be determined as the project proceeds. In particular, we will be looking at maintaining a copy of the full database on-line somewhere.

One gap that has not yet been filled is a way to indicate processors supported. Many vendors support only one or two processors, while a few support a broad list. We are looking for suggestions on how to compile this information in a concise form; perhaps a separate table for those who support a large list of machines would be best.

If you have any questions, suggestions, or submissions, contact:

Lyle Greg Lisle, P. E.
L Squared Electronics
2160 Foxhunter Court
Winston-Salem, North Carolina 27106-9621
910-924-0629
L.SQUARED@GEnie.geis.com

Offerings codes:
L = Literature, S = Software,
H = Hardware, C = Consulting,
T = Training

Forth standards supported:
FIG = fig-Forth
F79 = Forth-79
F83 = Forth-83
ANSI = ANS Forth

4th Wave Computers Ltd.
C ANSI
2314 Cavendish Drive
Burlington, Ontario L7P 3P3 Canada
905-335-6844
p.caven@ieee.org

A Working Hypothesis, Inc
C
P.O. Box 820506
Houston, Texas 77282 USA
713-293-9484
70410.1173@Compuserve.com

AM Research
LSHC ANSI
4600 Hidden Oaks Lane
Loomis, California 95650–9479 USA
800-949-8051
sofia@netcom.com

Bernd Paysan
S ANSI BigForth
Stockmannstr. 14
81477 MuenchenFRG Germany
++49 89 798557
paysan@informatik.tu-muenchen.de

Blue Star Systems
S ANSI Forth/2
P.O. Box 4043
Hammond, Indiana 46324 USA
ka9dgx@interaccess.com

Delta Research
S F83 JForth
P.O. Box 151051
San Rafael, California 94915 USA
415-453-4320
phil@3do.edu

FORTH, Inc.
LSHCT ANSI polyFORTH, chipFORTH
111 N. Sepulveda Blvd. Ste. 300
Manhattan Beach, California 90266 USA
800-553-6784
ERATHER@aol.com
FORTHSA@aol.com

Forth Interest Group
SL
P.O. Box 2164
Oakland, California 94621 USA
510-893-6784
JDHALL@netcom.com

Frank Sergeant
SC ANSI Pygmy
809 W. San Antonio St.
San Marcos, Texas 78666 USA
F.SERGEANT@GEnie.geis.com

Frog Peak Music
S F83 HMSL
P.O. Box A36
Hanover, New Hampshire 03755 USA
603-448-8837
phil@3do.edu

L Squared Electronics
SC Pygtools, Pygmy
2160 Foxhunter Ct.
Winston-Salem, North Carolina 27106 USA
910-924-0629
L.SQUARED@GEnie.geis.com

Laboratory Microsystems, Inc. (LMI)
S F83 UR/FORTH
12555 W. Jefferson Blvd., #313
Los Angeles, California 90066 USA
310-306-7412
duncan@nic.cerf.net

MicroProcessor Engineering Ltd.
HCLS ANSI PowerForth, ProForth
133 Hill Lane
Southampton SO15 5AF England
+44 1703 631441
sales@mpeltd.demon.co.uk

Miller Microcomputer Services
LSHCT F79 MMSFORTH
61 Lake Shore Road
Natick, Massachusetts 01760–2099 USA
508-653-6136
dmiller@im.lcs.mit.edu

Mountain View Press, Div. of
Epsilon Lyra
LSHCT ANSI MVP-Forth
Star Rt. 2, Box 429
La Honda, California 94020–9726 USA
415-747-0760
ghaydon@forsythe.stanford.edu

Redshift Limited
S
726 No. Locust Lane
Tacoma, Washington 98406 USA
206-564-3315
RedForth@AOL.com

Rob Chapman
SbotKernel, Timbre
11120 – 178 St.
Edmonton, Alberta T5S 1P2 Canada
403-430-2605
rob@idacom.hp.com

Science Applications International Corp.
CSTH ANSI Until, LMI, Uniforth
301 Laboratory Road
Oak Ridge, Tennessee 37831 USA
615-482-9031
smithn@orvb.saic.com

T-Recursive Technology
C ANSI
221 King St. East, Suite 32
Hamilton, Ontario L8N 1B5 Canada
905-308-3698
B.RODRIGUEZ2@GEnie.geis.com

Ultra Technology
LSCT ANSI P21Forth
2510 – 10th St.
Berkekey, California 94710 USA
510 -848-2149
jfox@netcom.com

Macro Processing for Forth

Wil Baden
Costa Mesa, California

The most popular programming language automatically pre-processes source through a macro processor before compiling. This augments the power of the language tremendously. For a long time, I wanted macro processing for Forth. The problem for Forth is aggravated because it has an additional complication that macros must work when interpreting as well as when compiling.

Like so many things in life that you keep putting off, when you finally get around to doing it, it's easy. All that is needed is a variation of EVALUATE that will make substitutions for a parameter flag before evaluating. I call this EVALUATED. Given EVALUATE, EVALUATED is easy to write. EVALUATED makes substitutions in a string, and then uses EVALUATE. See Listing One.

I use the swung dash "~" (often miscalled "tilde") as the parameter flag. As such, I call it "twiddle" or "parameter." This seems the least likely character to conflict with existing Forth words. If a swung dash is needed in a macro, S" ~" can be used as a parameter.

The actual parameters for a macro are selected by looking ahead in the input source. Words like PARSE-WORD, defined

```
: PARSE-WORD ( -- string . )
  BL WORD COUNT ;
```

are used to pick up an actual parameter.

Within a macro-template, twiddle ~ is used as a placeholder for a parameter. The actual parameters are character-string (c-addr len) pairs on the stack.

Example One: 0x

Here is a useful macro that temporarily changes the value of BASE.

```
: 0x    parse-word
  S" HEX ~ DECIMAL" evaluated
  ; IMMEDIATE
```

0x takes the next word from the input source and sandwiches it between HEX and DECIMAL. [See Figure One.]

It works for input, too:

```
0x FF is 255.
```

Try:

```
-1 0x U.
```

Example Two: [0x]

0x FF always gives you 255 only when interpreting. For sedecimal numbers when compiling, [0x] should be used.

```
: [0x]    parse-word
  S" [ HEX ] ~ [ DECIMAL ]"
  evaluated ; IMMEDIATE
```

Thus:

```
: low-byte ( n -- x )
  [0x] 00FF AND ;

: RAND  ( -- random )
  RANDSEED @ ( random)
    1103515245 *   12345 +
    DUP RANDSEED !
  16 RSHIFT   [0x] 7FFF AND
;
```

Example Three: ??

?LEAVE, ?EXIT, ?NEGATE, ?DNEGATE, and so on are found for convenience in many systems. Instead, just define

```
: ??    parse-word
  S" IF ~ THEN" evaluated ; IMMEDIATE
```

Figure One.

```
0x mumble        becomes          HEX mumble DECIMAL
0x .                              HEX . DECIMAL
0x ?                              HEX ? DECIMAL
0x .S                             HEX .S DECIMAL
0x U.R                            HEX U.R DECIMAL
32 0 DO I 4 0x .R LOOP            32 0 DO I 4 HEX .R
DECIMAL LOOP
```

and write

```
?? LEAVE     ?? EXIT
?? NEGATE    ?? DNEGATE  and so on.
```

(This word was first defined by NEIL BAWD in 1986. This word also defined NEIL BAWD.)

Example Four: [']

The examples so far suggest that macros replace POSTPONE. Indeed, in some systems POSTPONE can be defined by a macro. Many state-smart definitions can be written using macros to be apathetic about state. However, it will be useful to have ['] state-smart so it can be used in definitions that are otherwise state-stupid.

```
: ['] STATE @ IF
  parse-word S" [ ' ~ ] LITERAL" evaluated
  ELSE ' THEN
; IMMEDIATE
```

Example Five: TO

Most definitions of TO are state-smart. Here, only ['] in the definition is smart:

```
: TO
  parse-word S" ['] ~ >BODY !" evaluated
; IMMEDIATE
```

Example Six: SAY

As we all know, we can use string-quote S" to get a character-string that doesn't contain a quote-character.

We can use a character other than a quote-character as the delimiter for a character-string in this round-about way in Standard Forth (using | as delimiter).

```
: raven  [ CHAR |
  PARSE Quoth the raven "Nevermore"
  | ] SLITERAL TYPE ;
```

This lets us define templates for macro expansion that contain quote-characters.

Now we can define defining words that define words that can display their own name.

```
: say
  parse-word 2DUP
  [ CHAR | PARSE : ~ ." ~ " ;| ] SLITERAL
  evaluated
; IMMEDIATE
```

Note how 2DUP has been used to make two parameters out of one.

Example Seven: SET

Here is some syntactic sugar:

```
: set  parse-word [CHAR] ;
  PARSE 2SWAP S" ~ ~ !" evaluated
; IMMEDIATE
```

2SWAP has been used to change the order of parameters.

```
set X X @ 5 *
```

becomes

```
X @ 5 * X !
```

and

```
set X 5 ; set Y X @ 10 +
```

becomes

```
5 X ! X @ 10 + Y !
```

Example Eight: AGAIN, ANDIF, ORIF, OF

When a macro doesn't have a parameter, EVALUATE should be used on the template.

```
: AGAIN S" FALSE UNTIL" EVALUATE
; IMMEDIATE

: ANDIF S" DUP IF DROP" EVALUATE
; IMMEDIATE

: ORIF  S" ?DUP 0= IF" EVALUATE
; IMMEDIATE

: OF    S" OVER = IF DROP" EVALUATE
; IMMEDIATE
```

Conclusion

Macros are a convenient way to extend the power of Forth. They are also generally easier to use than POSTPONE, >IN, and SOURCE.

When target-compiling, macros on the host let you use high-level definitions that do not exist on the target.

Macros can be used to copy code inline to avoid subroutine linking in time-critical sections.

It has not been shown here, but macro expansions can be made conditional.

This package has certain limitations and can be improved. What would you do? How should errors be handled?

Thanks to NEIL BAWD, RAUL D. MILLER, and ULRICH HOFFMANN for inspiration and guidance.

Code appears on next page...

Wil Baden is a professional programmer with an interest in Forth. He can be reached at e-mail address wilbaden@netcom.com.

Listing One. Baden's approach to macro processing.

```
 1 ( EVALUATED   EVALUATE with Parameter Substitution.  WWB 95-03-15 )

 3       VARIABLE pushback-ptr   HERE UNUSED + pushback-ptr !

 5       : pushback-char                ( char -- )
 6            pushback-ptr @ HERE 200 CHARS + < ABORT" Buffer Error."
 7            -1 CHARS pushback-ptr +!
 8            pushback-ptr @ C!                ( )
 9       ;

11       CHAR ~ CONSTANT parameter

13            : pushback-string       ( string . -- )
14                 BEGIN                         ( string .)
15                      DUP
16                 WHILE
17                      1-   2DUP CHARS + C@ pushback-char
18                 REPEAT                        2DROP
19            ;

21       : pushback-parameter      ( param . string . -- string . )
22            DEPTH 4 < ABORT" Macro Parameter Error. "
23            2>R pushback-string 2R>          ( string .)
24       ;

26  : evaluated                          ( ... string . -- ??? )
27       pushback-ptr @ >R
28            BEGIN                           ( ... string .)
29                 DUP
30            WHILE
31                 1-   2DUP CHARS + C@     ( ... string . char)
32                 DUP parameter = IF DROP  ( ... string .)
33                      pushback-parameter
34                 ELSE                     ( ... string . char)
35                      pushback-char
36                 THEN                     ( ... string .)
37            REPEAT       2DROP            ( ...)
38            pushback-ptr @ R@ OVER - 1 CHARS / EVALUATE ( ???)
39       R> pushback-ptr !
40  ;
```


ADVERTISERS INDEX

(Fast Forthward, continued from page 39.)
often need to receive kernel services that are uniquely suited to their needs.

Further, a Forth assembler can improve the performance of the extension. But what if some speedier or more expansive file I/O hooks into the kernel are needed? Then you may be forced to undertake an overhaul of the kernel, where you don't have the clarity and other advantages of a high-level programming environment.

With Forth, there is no layer "in between." Either you resort to regenerating the kernel in assembly, or you resort to a Forth and assembly mixture that talks to a less than custom-fitted kernel. The C technology is more granular in this regard. Libraries permit a very broad code base to be mutable even while it remains specified in a portable, highly standardized, high-level language.

(In C, you can also be locked out of certain layers of the code base, such as when a proprietary operating system takes up residence. Still, at least in theory, you could write your own operating system in C to show its ability to exercise control over the full expanse of the code base using a single language. This is part of the appeal of UNIX when full source is provided.)

Using lots of execution vectors in a Forth kernel gives us a somewhat similar flexible code base in a Forth environment. Kernel extensions can be coded in high-level Forth, with only slight additional calling overhead.

But in contrast to Forth, C permits equivalent code base changes without introducing any inefficiencies. For example, you would not be required to call a library-code routine from a wrapper routine. That way, I could supplant the C routine *fopen()* using a library I write myself, and it will remain a direct call away—no function pointers are required.

The receptacles for plug-and-play Forth kernel components can be vectors or deferred words. To make the Forth kernel extensible through such provisions requires the kernel developer to insert these hooks into the kernel in the first place. So you are free to extend the kernel this easily, if at all, thanks only to the far-sightedness and desire of its author to accommodate you.

As soon as you don't find a vector or deferred word where you need one (which is likely to be the case because of the usual penchant of the kernel developer to create a minimal Forth), you are back to square one. Forth and C should be able to redefine every symbol that you might want to retrofit with new functionality (C, of course, won't let you redefine any of its syntax keywords). Further, both Forth and C should let you specify the new behavior in a high-level language with no added penalty in terms of calling overhead. I believe C can provide a clear advantage over Forth in terms of creating mutable code bases at the present time.

The Long-Suffering Development Environment

I also claim that Forth's relatively weaker support of mutable code bases has been an impediment to the evolution of Forth's development environment. If this is not true in every respect, at least it is true in the respect that new development environment tools are likely to lack portability for the foreseeable future.

As evidence of progress, certain Forth vendors are supplying new Forth development environments, some of which include visual development tools. Nevertheless, I believe that the level of standardization and ease-of-change represented by C library and linking technology has no Forth equal in these vendors' systems. Accordingly, Forth development environments, and the code bases of which they are part, are likely to continue to be slow in coming. When they do come, they are more likely to be limited in platform availability, they are more likely to evolve slowly, and they are more likely to impede customization.

The need for us to be able to experiment at the kernel level remains strong. I don't see that activity subsiding due to ANS Forth, or even GUI Forths. As long as this situation cannot be reversed, we might as well move ourselves up a notch in terms of the sophistication of the resources we use to tackle kernel retooling. A more scaleable and mutable Forth kernel would seem to be appropriate.

The time has come to build a common vessel (or framework) upon which all of our individual kernel embellishments can be conveyed, either above the kernel, or inside the kernel, or below the kernel. The kernel should become less static, and should become part of a bigger, mutable code base that supports interchangeable parts. Kernel extensions and interchangeable kernel parts should be maintained using a Forth-like facility, not nonportable assembly, nor confusing metacompiling semantics.

This is an architectural problem. The time has come to take the Forth kernel further Forthward. As far as I can see, this also means taking Forth further C-ward.

Fast FORTHward

For Want of a Kernel Development Environment

Mike Elola
San Jose, California

A couple of years ago, I undertook a year-in-review look at the opinions that had run in *Forth Dimensions*. The Forth development environment emerged as our leading concern at that time. The consensus was so strong that this topic was also chosen as a theme for the 1993 FORML Conference, as well as that year's *Forth Dimensions* article contest theme.

The following year, an official ANSI-sanctioned Forth standard was born. As far as I can tell, we have not looked toward the standard to address our concerns with regard to Forth's development environment. The standard has correctly tried to stay clear of implementation issues, and that seems to include the development environment.

However much the ANSI Forth standard holds promise of greater code portability, the development environment is not so likely to benefit from it. It's also likely that Forth vendors cling to the hope of differentiating their products through the development environments they can create for them.

However, I feel there are also technical problems that prevent us from developing uniform development environments across computing platforms.

Architectural Support for Kernel Extension

Extending the kernel is the great Forth pastime. The adventures we embark upon are sometimes shared in these pages. Take the last issue; it gave us one article describing a kernel that supported objects through vocabularies. Such offerings are very educational, so no doubt they will continue to be pursued and published.

Nevertheless, outsiders can easily be critical of us for retooling the kernel so much. How do we defend ourselves against such criticism?

(Perhaps explanations of successful Forth applications seem relatively drab alongside kernel enhancements.) Forth is perfectly suited to experimentation. While we experiment, we often need to fiddle with the kernel. Naturally, we shape the kernel to our every desire. However, by pursuing our changes at the level of the kernel, the development environment becomes the domain of the individual Forth developer or vendor.

By preoccupying ourselves in this way, we become too firmly wedded to a particular kind of kernel, the kind upon which our home-grown development tools depend.

The situation could improve dramatically if we could find a way to allow kernel enhancement code to be moved around more easily. However, the kernel is typically written in assembly language. How do we obtain the desired kernel embellishments without descending into that nonportable code?

One superficial and one genuine way to avoid the need to traffick with this nonportable software layer can be identified: (1) a metacompiler can allow the kernel to be regenerated in a more Forth-like way, using the assembler conventions and text-interpretation conventions of Forth rather than those of a foreign assembler (nevertheless, even Forth assemblers are by nature nonportable); and (2) an easily retrofitted kernel can be built through a process of compilation using a translation-affording language such as C.

A third option exists that I don't think has been attempted before. A kernel can be fashioned as a framework upon which new functionality is easily hung. (The last "Fast Forthward" installment suggested a text-interpretation framework and tried to demonstrate how a preprocessor attuned to Forth could be fashioned from that framework.) More on this a bit later under "Mutable Forth Kernels."

Mutable Code Bases

Even if written in assembly language, the Forth kernel is able to impart substantial portability to Forth applications, despite its own lack of portability. To further this kernel-derived application portability, the ANS Forth standard sought to define only the interfaces (Forth word behavior) and not the implementation of many new kernel routines. The effect of this expanded, yet fixed, code base will address many portability troubles of the past.

Note that the code base represented by the Forth kernel imparts to Forth applications all their processor-native computational abilities. Often, this code base is called a Forth virtual processor.

I have repeated many times the claim that Forth needs the equivalent of C libraries and linkers. I see as their principal advantage a more rapidly evolving, yet standard, code base that can be built with a single language without resort to assembly or meta-language environments inside of a language.

(Certain function domains—including those beyond the reach of the Standard C library—are less standardized.

However, library vendors tend to establish their own *de facto* standards through the popularity of their offerings. Library vendors such as Zinc Software, the vendor of the Zinc Application Framework, are even able to surpass many of the language vendors in terms of their visibility. Furthermore, because you get the C source, you can be reasonably certain they are able use C as a single homogeneous language with which to create their code base.)

Now that we have a few Forth systems capable of talking to Windows, a Forth GUI is able to ride the wave of enhancements to various code bases. For example, when the look and feel of the Windows user interface is enhanced, those Forth applications that are moderated through Windows will benefit—if not automatically, then through simple relinking.

Let's not ignore that someone has to be able to build and release enhancements to these code bases. Forth has the drawback that its principal code base is the kernel, which is written in a processor-specific assembly language and which the author often tends to consider static. Metacompiling is Forth's handy alternative, but metacompiling is usually the purview of the elite. Contrast this with the ease of relinking inside a C programming environment, which is a standard operating procedure to everyone who learns C. Thus, C provides non-intermittent support for a continually evolving code base.

The better maintained and faster-evolving C code bases are partly due to the use of a single, consistent language for their development. In contrast, a Forth metacompiler introduces the semantics of several languages simultaneously—including a helping of "target assembler."

The kernel is a springboard to a Forth style of programming. Thus, tools such as metacompilers are a means to an end. I suspect we don't really want to use such metacompilation environments any longer than absolutely necessary to bring up an improved kernel or a retargeted application.

Code Base Evolution Within the C Programming Environment

Using C, a code base for many applications can be fully written in C. As long as the header file declarations are not in need of change, the code base can be re-created in whole or in part. Then an unchanged library client (an application) can be relinked with the new library to derive the benefits of an enhanced code base. The expanded code base might, for example, insert some memory-monitoring provision to help detect memory leaks due to improper use of dynamic memory provisions.

Not only can the code base be expanded and scaled back to improve the abilities of the developer to develop an application, but the code base can be expanded in ways that benefit end-user applications, as in the case of better looking and behaved Windows windows.

The application source code can be compiled without there first being a full disclosure of the contents of the underlying code base. The separate linking pass takes care of plugging together all the routine call and caller points. Before linking, the application, the code base, or both can be swapped out for an improved version.

This underscores the importance of the linker step in the processing typically employed by compiled languages such as C: It leads to interchangeable and insertable code-base components. (While routine interfaces must remain static, behavioral factors such as its boundary conditions, error conditions, performance, memory efficiency, side effects, and other aspects can be affected.)

As an example of an insertable component, consider the Win32 dynamic link library (DLL). If you configure this DLL to Windows, you can change the overall behavior and appearance of Windows. Because it is dynamically linked, many already compiled and linked applications can take advantage of the enhanced code base without developer intervention.

Forth does not compare as favorably.

Surely Forth has the factoring ease to create code bases that can be transitioned into and out of with amazing grace. But if the code transition involves a kernel modification, Forth has nothing comparable to offer.

Because C's Standard Library reaches so deeply into the language, things like the I/O file functions are not actually part of the language—but are part of the library base. Further, the linking technology permits you to swap out that file I/O component if you care to do so (along with numerous other components that are part of the standard C library). You can make such a transition by using high-level C to alter the library code base to suit your problem, even to the extent of supporting the special needs of embedded systems.

A true language-translation technology is also better suited to Forth kernel development, because of the flexibility it can bring. Because a kernel so written is specified in a high-level language, it can be more subject to programmer control (as opposed to vendor control). Without descending into assembly language, it is fast. Finally, while offering more portability, it can also create a kernel that is "component oriented," where each component of the kernel code base can be selectively removed and reinserted.

The time has come to build a common framework upon which our kernel embellishments can be conveyed.

Mutable Forth Kernels

Us Forth developers can benefit from a more flexible and scaleable kernel built as part of a mutable code base. Consider our intermittent need to use tools such as profilers, sophisticated debuggers and tracers, editors, optimizers, version control, and the like. Likewise, application-specific libraries might be better managed as monolithic packages, such as a package for dynamic memory management or a database library.

Can we make the Forth kernel "plug and play" with regard to these extensions? Certainly database, dynamic memory, and other Forth extensions have been layered on top of Forth. But for a serious application, these extensions

(Continues on page 37.)

Be sure to Vote!