$10 Volume XVIII, Number 3 September 1996 October

FORTH DIMENSIONS

Breaking Code

4th, an Experiment in C

Digital Input and Synchronous I/O

The Elephant Who Refuses to Be Bagged

Development Aids for New Micros

Contents

Features

Departments

Editorial

Forth Dimensions
Volume XVIII, Number 3
September 1996 October

Published by the
Forth Interest Group

Editor
Marlin Ouverson

Circulation/Order Desk
Frank Hall

In case you missed my past polemics about Forth on-line, the FIG Board report contained in this issue will provide a reminder: there are substantial benefits to joining the on-line Forth community, especially if you are a FIG member. The Board continues to enhance the rewards of FIG membership, on-line or off, as it also works to improve the organization's ability to present Forth's contemporary advantages to the rest of the world.

Java, of course, is the recent headline-grabber. It's also the latest recipient of the oft-recited "Why didn't they use Forth?" and "Forth could do that" laments. This year's Rochester Conference had a workshop on the subject and, synchronistically, the article on 4tH this month profiles an experimental system that generates bytecode for maximum portability.

We hope you enjoy this issue, even as we prepare for the next and the one after. Please consider being part of our upcoming publication plans—*FD* is looking for articles about interesting projects, problems solved, managing large teams and reams of code, overcoming resistance to Forth, ANS Forth (see "Letters") at both micro and macro levels, portability, embedded systems, tutorials, and—you get the idea. Write soon!

—Marlin Ouverson, Editor

dot-quote

I first read about Forth in 1975, but was too green to understand its underlying simplicity, and how easy it would be to at least partially implement a Forth. Just doing away with variables and using the stack and a primitive address interpreter is orders of magnitude better than "classical development." (Do people still use those mucho-insertion-force chips that have to sunbathe under UV for half an hour every time you make a change?)

Having gotten so much mileage from a TTY connection, F83 on my host system, and a bare address interpreter, I sometimes wonder if it is even right to call Forth a language. It's both far more and far less. It's a way of thinking about programming that is so simple and workable, it feels almost as compelling as using integer math to count.

I think too much fussing over the 'syntactic sugaring' can obscure the real value of the Forth paradigm. So can concern over ever fancier and more powerful Forths. Even more so than Unix, Forth realizes the power of simplicity. Lao Tzu says, "That which is without substance can enter where there is no room." This also strikes me as very Forth-like. So, being "without substance"—that is, without a lot of intellectual baggage of no direct applicability—Forth lends itself to implementation in things like microcontrollers.

Forth may be the most transparent tool I have ever seen, in *any* domain. It is whatever you make of it. Certainly there is value to having some common definition of Forth available, it makes sharing ideas and code easier. However, it strikes me as a kind of idolatry to try to standardize Forth as one would C. The essence of Forth is in its malleability and its lack of artificial boundaries between the user and the hardware.

—David Kenny (dk@winternet.com)
Adapted from comp.lang.forth

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

Subscription to *Forth Dimensions* is included with membership in the Forth Interest Group at $45 per year ($53 Canada/Mexico, $60 overseas air). For membership, change of address, and to submit items for publication, the address is: Forth Interest Group, P.O. Box 2154, Oakland, California 94621. Administrative offices:
510-89-FORTH Fax: 510-535-1295

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 $45/53/60 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."

Letters

Send your feedback, questions, criticisms, and other responses to editor@forth.org or to the editor in care of Forth Dimensions, *P.O. Box 2154, Oakland, California 94621. Letters may be edited for clarity and length.*

What ANS Forth Needs Now

Dear Marlin,

I noted with interest the article on hFORTH in *FD* XVIII/2. I acquired a copy of that system but have essentially no documentation for ANS Forth. Wonyong Koh's article is a nice overview, but is hardly adequate documentation for a new user of ANS Forth. This missing link still concerns me and, I hope, other members of FIG.

It has occurred to me that this void might be filled by a team effort devoted to producing readable documentation of ANS Forth which could be published in one or more issues of *FD* devoted entirely to that subject. Surely it would not be too much to ask of those knowledgeable members who participated in the development of the standard to join forces once again to provide usable documentation for those who wish to support the standard but have no real knowledge of it. I do not mean to underestimate the task, but rather to emphasize its importance to the survival of Forth. Without a usable and supported standard, it seems to me, Forth has nowhere at all to go.

Speaking for myself, I have not used F-PC (an excellent system for which Tom Zimmer is to be congratulated for producing) for several years, because effort spent in improving my skills on a non-standard system seems to me to be wasted. Now that we have a standard, we must use it or lose it and, frankly, I have all but despaired of having a chance to learn ANS Forth, much less use it. My 70 years do not leave much room for procrastination!

To put it rather bluntly, my membership renewal is due in a month or so, and I wish to know whether renewal is worthwhile for me.

Sincerely,
Bernard H. Geyer
bgeyer@smtp.northlink.com

Some feel that, because ANS Forth gives latitude to system implementors, most Forth users should simply learn individual systems—i.e., that attempting to understand the ANS document can be left to implementors. The thoroughness of the systems' manuals would, as usual, be left to each implementor.

From where I sit, your point is well taken. We definitely need authors willing to write about particular aspects of ANS Forth. Additionally, I encourage a cooperative effort such as the one you propose—if it were carried out via e-mail, it would only require a moderator and on-line team, carefully defining and parceling out the material, with patience, persistence, accuracy, and diplomacy.

Any takers? This could contribute to the wider understanding and adoption of ANS Forth. —*Editor*

Development Aids for New Micros Products

Richard W. Fergus
Lombard, Illinois

The New Micros company produces a line of single-board computers based on the Motorola 68HC11 microprocessor. This processor includes multiple ADC channels, several free-running timing registers, and flexible interrupts, which makes it a good candidate for many embedded real-time applications. In addition, New Micros includes a Forth dialect—MaxForth—in the CPU ROM. The hardware design of these boards is good, but I find the software and documentation lacking with regard to writing application software. After several years and numerous applications, a development platform has been assembled which is the subject of this article.

Introduction

On first inspection, it appears that MaxForth is ready to use for application development. There are EDITOR, LOAD, LIST, etc. words. This is misleading, since the EDITOR vocabulary is empty, and the LOAD and LIST words are non-functional. The documentation infers that a standard communication program or dumb terminal can be used. This is possible but awkward. My goal has been a platform which would give to program development for New Micros boards a "feel" similar to the other Forth systems, e.g., similar editor, block-oriented file structure, and load screens.

This involved adding several words to a host Forth system which handles the downloading and provides terminal emulation. Since most embedded applications should be PROMable, provisions were included for separated RAM variables and Motorola S19 record output for PROM burning. I also added a personal favorite: a MAKE-DOER function from Brodie's *Thinking Forth*, including a CHORE DOER in the keyboard loop for crude multitasking.

Host Additions

In addition to terminal emulation, two functions were added to the host Forth. One word, $LOAD, sends a host string variable plus CR to the target (the New Micros board). Similarly, BLOAD sends a full host block (1024 bytes) to the target. Both words wait for a CR response from the target before echoing the complete target response. I also added a logging function in the terminal emulation mode to copy the S19 records. The logging function could be done with most communication packages. This word usage will be described in more detail when the companion target functions are discussed.

I have not included a listing of these host words with this article, since the definitions are somewhat system and dialect dependent. I can supply a listing for either Uniforth or Pygmy dialects.

Target Alterations

One way to simplify the load/interpret sequence is to map the input buffer of 1024 bytes and change the number of characters QUERYed to 1024 so that a full block can be transferred and interpreted. To the host Forth, this function can look very similar to a standard load. Since this load function is a word in the host Forth, a load screen is written in the host system which calls other host screens to be sent (loaded) to the target system. All screens are developed with the host editor. Other than switching between terminal emulation (target system) and the host system, one would think they are dealing with one system.

The following programs assume the New Micros board is equipped with MaxForth version 3.5 and 8K x 8 RAMs in U2–U3 with base addresses of 2000 and 8000, respectively. The RAM in U2 can be replaced with a PROM after the program is developed. The MAKE-DOER as shown will not function with the earlier MaxForth version (3.3) because of a change in the <BUILDS-DOES> construct. For those who still have some version 3.3 boards, modifications for the MAKE-DOER code and other changes, including correcting the DMIN word, are included in the text.

Host-to-Target Load

Screen 1—This screen, when called (1 LOAD) on the host, reconfigures the target input buffer, dictionary, etc., then sends the remaining screens to the target as target load screens. In MaxForth, the C/L variable sets the QUERYed characters (not traditional usage). After the last screen is loaded, a "Done" is prompted and the terminal emulation (TT) is called. Obviously, the first screen is written in the host dialect, while the remaining screens are written in the target dialect.

VAR$ is a host word which defines a counted string

variable. Lines two and three define string variables which set the input buffer address and QUERY size. Until the input buffer is reset, the command strings must be limited in length (80 and 16 bytes for version 3.5 and 3.3, respectively). Lines six and seven define a long string which sets the dictionary pointer, "forgets" TASK, and relocates both stack pointers and PAD. TASK, a null word defined in low RAM, is "forgotten," thereby eliminating any dictionary reference to low memory.

The first three of the above strings are sent by line eight, which uses the host $LOAD to send a counted string variable, add a CR, wait for the target response, then echo the target's response on the host CRT screen.

BLOAD is used to send the listed screens to the target for loading and interpretation. After each screen is sent to the target, the echo from the target will be displayed on the host display. The echo includes the screen as sent by the host (straight text with no CRs or LFs), errors that may have occurred in the target interpretation, and the final "OK" prompt. When the last screen transmission is completed, the string SET is sent to reset the target dictionary pointers (APPEND) and a cold reset (COLD) is called.

The load process is completed by forgetting the host words defined on the host load screen, displaying the "Done" prompt, and calling the terminal emulation TT.

```
SCR # 1
  0 (                    MAX SETUP LOAD SCREEN                          )
  1 FORTH DEFINITIONS DECIMAL              ( Reconfigure New Micros )
  2  VAR$ TB HEX 400 1E !"                 ( Increase QUERY "C/L" )
  3  VAR$ TB+ 9A80 1C !"                   ( Move buffer TIB )
  4 (        Dict ptr    -TASK        S0         R0        PAD      SP!)
  6  VAR$ INT 2004 DP ! E6BE 38 ! 9A7F 10 ! 9F00 0E ! 9F80 22 ! F57C
  7   EXECUTE DECIMAL"
  8  TB $LOAD TB+ $LOAD INT $LOAD          ( Send above )
  9
 10   2 BLOAD  3 BLOAD  4 BLOAD  5 BLOAD   ( NMI Additions/mods )
 11   6 BLOAD                              ( S Record )
 12   7 BLOAD  8 BLOAD  9 BLOAD            ( System setup )
 13  VAR$ SET APPEND COLD"
 14  SET $LOAD FORGET TB         ( Append new, reset, & host forget )
 15 CR ." Done" CR  TT  ;S                 ( Prompt and goto target )


SCR # 2
  0 (                    FORTH WORDS REDO                               )
  1 FORTH DEFINITIONS HEX
  2 : CODE ( --- )  [COMPILE] CODE-SUB ;                  ( Redo CODE )
  3 : END-CODE ( --- )  39 C, [COMPILE] END-CODE ; ( Redo END-CODE)
  4
  5 : BEEP ( --- ) 7 EMIT ;                              ( Sound Bell )
  6 : LOAD ( --- ) BEEP ;              ( Beep and disable MAX LOAD )
  7 : LIST ( --- ) BEEP ;              ( Beep and disable MAX LIST )
  8
  9 CODE I> ( --- ) 07 C, 9783 , 0F C, END-CODE    ( Stop/save IRQ )
 10 CODE >I ( --- ) 9683 , 06 C, END-CODE       ( Restore IRQ mask )
 11 : EC!  ( C ADDR --- )     ( Redo EEC!--store only if different )
 12     OVER FF AND  OVER C@ -  IF                       ( Different? )
 13       I> EEC! >I ELSE 2DROP THEN ;          ( Disable/enable IRQ )
 14 : EN! ( N ADDR --- )                         ( Store N in EEPROM )
 15     OVER >< OVER EC!  1+ EC! ;  DECIMAL ;

SCR # 3
  0 (                    SEPARATED RAM AND EEPROM VARIABLES             )
  1 HEX  FORTH DEFINITIONS
  2 : IS ( N --- )  CONSTANT ; ( Rename CONSTANT for easy reading )
  3    84 IS 'RAM                      ( RAM address table )
  4 : RAM ( N --- ADDR )               ( Separated RAM variables )
  5     'RAM @ SWAP 'RAM +! ;          ( Get address & set next adr )
  6    86 IS 'EROM                     ( EROM address table )
  7 : EROM ( N --- ADDR )              ( Separated EEPROM variables )
  8     'EROM @ SWAP 'EROM +! ;        ( Get address & set next adr )
  9    88 'RAM !   B700 'EROM !        ( Set initial pointers )
 10 : KILL ( --- ) [COMPILE] UNDO ;   ( Rename Max.UNDO )
 11                                    ( Compile "headerless" words )
 12 : 0! ( N - ) [ FDD6 , ] ;       : 0 ( - N ) [ F65A , ] ;
 13 : 1 ( - N ) [ F656 , ] ;        : 2 ( - N ) [ F652 , ] ;
 14 : 3 ( - N ) [ F64E , ] ;        : 4 ( - N ) [ F64A , ] ;
 15 DECIMAL ;S
```

MaxForth Additions

Screen 2—Several convenience words are defined on this screen. Lines two and three redefine CODE/END-CODE to be more compatible with similar words used in other dialects.

Accidental use of the MaxForth LOAD or LIST will disrupt the system, therefore lines six and seven redefine these words to sound a "beep" as a reminder.

Lines nine through 15 are included to improve the EEPROM storage function, especially if EEPROM storage is required during program operation. Both byte (EC!) and word (EN!) EEPROM storage words are available. All EEPROM storage is restricted to only changes, in an effort to increase EEPROM life if continuous updating is needed. I> and >I control the interrupt functions which must be stopped during the EEPROM storage sequence.

Screen 3—Lines two through six provide for PROMable code with variables located in a separate RAM area. A similar format is used for EEPROM variables. Obviously,

IS has the same function as CONSTANT but it is added to make the RAM/EROM variable assignment more readable. Both RAM and EEPROM variables follow the same assignment format, e.g.:

```
<# of bytes> RAM(EROM) IS <variable name>
```

The RAM pointer ('RAM) is initialized in low RAM with room for about 175 (50 in ver. 3.3) variables. The pointer can be reset to high RAM if more space is required.

The remainder of the screen renames MaxForth's UNDO (which will be used in the MAKE-DOER construct) to KILL, and restores headers for several words which were headerless. With a little squeezing, the following lines can be added to this screen for version 3.3 operation:

```
: DLITERAL ( D - ) [ EA9F , ] ;
: @@ ( ADR - ) [ FE07 , ] ;
: DMIN ( D1 D2 - D )
( Original DMIN incorrect )
  2OVER 2OVER D< 0= IF 2SWAP THEN 2DROP ;
```

MAKE-DOER

Screen 4—I seldom see reference to the MAKE-DOER construct, which I have found very useful. MAKE-DOER performs a vectored execution which can be used to modify functionality without complicated decision strings. It can also be used for deferred definition functions. Words defined as DOERs initially do nothing. A MAKE sequence modifies the named DOER to execute the code following the MAKE <DOER>. Word definitions may contain multiple MAKE statements. There is no limit to the number of DOER variations by multiple MAKE definitions. A DOER can be changed dynamically, e.g., a MAKE <DOER> sequence within a called DOER can redefine the DOER for the next call. If needed, UNDO <DOER> resets the word to do nothing. For more detailed description, see Brodie's *Thinking Forth*.

The following modifications are required for version 3.3 operation:

Line 5, change ... @ 2+ @ ...
to: @ 4 + @

Line 9, change ... @@ ...
to: @ 2+ @

Line 12, change... @@ ...
to: @ 2+ @

Screen 5—A special DOER called CHORE is defined and installed in the KEY loop. With this DOER installed, CHORE is called each cycle of the KEY loop as the processor waits for a keypress. Most of my real-time applications use interrupt routines for process control and data collection. For periodic data handling, alert messages, etc., a flag will be set which will be sensed by the CHORE DOER code. Any CHORE action will inhibit the KEY function until the CHORE action is completed.

Motorola S19 Record

Screen 6—A standard S19 text record can be generated for any memory range with the S.REC word. This record can be read by many EPROM burners and converted to the original binary image for EPROM duplication. The primary purpose is for transferring the developed program from the RAM at 2000h to an EPROM. The text string generated is sent via the serial port to the host; therefore, the host should provide a logging function. If this is not the case, most communications programs contain a logging function which can be used to save this record.

Dictionary Setup

Screen 7—All dictionary and RAM/EROM pointers have complementary variables which become "constant" when the program is transferred to EPROM. APPEND, to

```
SCR # 4
  0 (                    DOER MAKE                                    )
  1 HEX 2 RAM IS MARKER ( STARTING FORTH p276--return stk post-inc)
  2 : DOER <BUILDS F891 'RAM @ DUP , !  2 'RAM +!  ( EXIT default )
  3     DOES> @ @ 2- >R ;                          ( Get CFA from RAM )
  4 : (MAKE) ( --- )                                 ( Vector DOER )
  5     R> 2+  DUP 2+ DUP 2+ SWAP @ 2+ @ !   ( Return stk to next )
  6     @ ?DUP IF 2- >R THEN ; ( Continue over if MARKER not zero )
  7 : MAKE ( --- )    STATE @ IF                        ( Compiling )
  8     COMPILE (MAKE) HERE  MARKER ! 0 , ( Set MARKER adr and flg)
  9     ELSE HERE    [COMPILE] ' @@ !                ( Interpreting )
 10     LATEST C@ 20 XOR LATEST C! [COMPILE] ] THEN ;  IMMEDIATE
 11 : ;AND ( --- ) COMPILE ;S HERE MARKER @ ! ;        IMMEDIATE
 12 : (UN) ( n --- ) F891 SWAP @@ ! ;       ( Restore EXIT vector )
 13 : UNDO ( --- )  [COMPILE] '            ( Disable DOER function )
 14     STATE @ IF COMPILE (UN)  ELSE (UN) THEN ;      IMMEDIATE
 15       DECIMAL ;S


SCR # 5
  0 (                    CHORE                                        )
  1 FORTH DEFINITIONS HEX    DOER CHORE
  2
  3 CREATE KEY' ( --- )          ( Include CHORE in keyboard loop )
  4   CC C, ' CHORE CFA ,        ( Load D with CHORE for execution )
  5   BD C, AT04 ,               ( Call as subroutine )
  6
  7   DE04 , EE0C , 3C C,        ( Point to Input block KEY-BC-PTR )
  8   EE00 , E600 ,              ( Get status )
  9   38 C, E402 , E803 ,        ( AND/OR bits )
 10   27EA ,                     ( Loop if nothing from keyboard )
 11   7E C, F8DB ,               ( Get character and return )
 12
 13
 14   7E B7F2 EC!  FEC0 B7F3 EN!    ( Assure Illegal Op interrupt )
 15   DECIMAL ;S
```

```
SCR # 6
  0 (                    S RECORD                                    )
  1 FORTH DEFINITIONS HEX
  2 : 2HX ( b --- ) DUP SCR +!              ( Add to checksum )
  3     S->D <# # # #> TYPE ;               ( Send ASCII hex byte )
  4
  5 : S.REC ( from to --- )
  6   SWAP BASE @ >R HEX                    ( Change base )
  7   ." S00600004844521B" CR               ( Header record )
  8   BEGIN ." S1" SCR 0!                   ( Use SCR for checksum )
  9     2DUP - 20 MIN DUP 3 + 2HX           ( Record byte count )
 10     OVER 100 / 2HX OVER FF AND 2HX      ( Starting address )
 11     0 DO DUP C@ 2HX 1+ LOOP             ( Do bytes )
 12     SCR @ NOT 2HX CR                    ( Check sum )
 13   2DUP = UNTIL 2DROP                    ( Continue )
 14   ." S9030000FC" CR R> BASE ! ;         ( End record )
 15 DECIMAL ;S

SCR # 7
  0 (                   SETUP VARIABLES                              )
  1 FORTH DEFINITIONS HEX
  2
  3
  4
  5 VARIABLE 'DP   VARIABLE 'EDP    ( Initial dictionary pointers  )
  6 VARIABLE 'FTH  VARIABLE 'ED     ( for PROM program "constants" )
  7 VARIABLE 'ASM
  8 VARIABLE *RAM  VARIABLE *EROM    ( Separated variable pointers )
  9
 10 : SET.DIC ( --- )
 11     'EDP @ 30 !   'ED @ 3E !      ( EEPROM-EDITOR dictionary )
 12     'FTH @ 38 !  'ASM @ 44 !      ( FORTH-ASSEMBLER dictionary )
 13     *RAM  @  'RAM !               ( Separated RAM )
 14     *EROM @ 'EROM !               ( Separated EROM )
 15     'DP  @ 4A ! ; DECIMAL ;S      ( Set FENCE )

SCR # 8
  0 (                   SYSTEM SETUP                                 )
  1 FORTH DEFINITIONS HEX
  2 VARIABLE 'AUTO                              ( Autostart word CFA )
  3   ' QUIT CFA 'AUTO !                 ( Autostart default is QUIT )
  4  ( **** NOTE **** Autostart word must end with QUIT )
  5 : SETUP ( --- )
  6      9FFF   0E !  9F00  22 !           ( Return stack R0--PAD )
  7      9B80   1C !   400  1E !  ( Input buffer TIB--length C/L )
  8      9B7F   10 !                        ( Parameter stack S0 )
  9      F891 [ ' CHORE CFA 4 + @ ] LITERAL !     ( CHORE to EXIT )
 10      KEY'   16 !                              ( Install CHORE )
 11      8004   DP !                    ( Dictionary pointer HERE )
 12      SET.DIC                    ( Other dictionaries-RAM-EROM )
 13      A55A 8000 !  'AUTO @ 8002 !               ( Startup word )
 14      [COMPILE] FORTH [COMPILE] DEFINITIONS
 15      [ F57C , ] ;    DECIMAL ;S              ( "COMPILE" SP! )

SCR # 9
  0 (                   APPEND-RESTORE                               )
  1 FORTH DEFINITIONS HEX
  2 : WARM ( --- ) 'DP @ DP !         ( Reset to "PROM" dictionary )
  3     UNDO CHORE SET.DIC            ( Cancel CHORE-reset pointers)
  4     ' QUIT CFA 'AUTO ! ;          ( Restore autostart default )
  5 : APPEND ( --- )                  ( Mark end )
  6     38 @ 'FTH !  3E @ 'ED  !           ( Save pointers )
  7     44 @ 'ASM !
  8     30 @ 'EDP !  HERE 'DP  !
```

be described, copies the pointer values from the developed program to these variable locations, thus saving the dictionary structures. During reset, the SET.DIC word transfers the pointers to the appropriate RAM locations.

Setup

Screen 8—A turnkey system can be provided with a two-step autostart sequence. The first step (SETUP) initializes the stack pointers, dictionaries, installs the CHORE DOER, and sets the vector for the second autostart word (located in RAM). A variable (PROM "constant") contains the CFA of a second autostart word. This word must end in QUIT and is set to a QUIT default by line three during initial load. The two autostart sequences were separated to allow for a thorough evaluation of the program before committing the last autostart sequence: If the system bombs with the last autostart in place, it may be difficult to regain control, since each reboot will produce the same condition. The second autostart can be tested by the command:

```
' <autostart word>
CFA EXECUTE <CR>
```

After program evaluation, the autostart is activated with the command:

```
' <autostart word>
CFA 'AUTO ! <CR>.
```

Boot Control

Screen 9—Several words have been included to control the boot process and allow for program development. APPEND is used at the end of a screen load to save dictionary and RAM/EROM pointers. This data is used by SETUP to reproduce the system conditions during boot. UNSET is used while debugging to disable both autostart sequences (with RAM program). Only the second autostart vec-

tor will be disabled if a PROM program is in place.

When the program is completed and ready to be committed to EPROM, PGM.S19 can be used to generate the appropriate Motorola S19 record.

Development Procedure

Application development with this host/target arrangement is not radically different from a single system. Once the screens are written, the load-debug process does involve a few extra keypresses to switch between host and target. Multiple screens can be loaded with a host load screen, or a single screen can be loaded from the host with the BLOAD word. Otherwise, the host/target processes appear similar to the user. From my experience, the most troublesome feature is awareness of which system is connected to the keyboard/CRT. My host systems prompt with a lowercase "ok," while the New Micros prompts with an uppercase "OK"—which is helpful, but I don't always pay attention!

The host screen file will consist of a host load screen and multiple target screens. The first host load screen can be common between different applications, with only the target load list modified for each target application screen set.

Minimal and Drop-Point Boards

The limited memory maps of the *minimal* and *drop-point* boards do not allow for an extended input buffer. For those not familiar with the minimal and drop-point boards, only the MC68HC11 chip (with MaxForth in PROM) and necessary glue chips are provided on the board. All programming must be contained in the CPU RAM and EEPROM (.5K each). In these cases, the word definitions can be loaded as short text strings using the host $LOAD. Host screens are written which define these strings and the following load sequence. Screen ten is an example of this technique. The word definitions are shown in a standard form as remarks followed by the definition of a character string which will define the word in the target system. Obviously, because of the limited memory, the target words are as abbreviated as possible.

Other Boards

Other Forth systems may be a bit more involved to modify than MaxForth with the non-standard use of C/L. It does require some familiarity with the system's memory mapping and facility to modify the source code. In a traditional Forth, the user variable S0 defines the address for the top of the parameter stack, which is directly below the input buffer. The S0 variable and input buffer user variable (TIB) must be moved to a lower address to provide for at least 1024 bytes for the input buffer. Changing the QUERYed characters can be more challenging. Generally, the QUERYed number is a literal in the QUERY word. Dumping the QUERY word code and locating the CFA of LIT should locate the QUERYed number (next two bytes) which should be changed to 400 hex.

Conclusion

This technique has provided a smooth development process for a number of New Micro projects. In addition to the New Micros boards, I use several other Forth packages with the same host. Adjusting for the dialect variances is frustrating enough without adjusting for a different editor or loading procedure.

R.W. "Dick" Fergus (rfergus@delphi.com) first decided to use Forth about twelve years ago, while developing radiation-monitoring equipment based on the RCA 1802 for a national laboratory. After looking for a while, he finally wrote his own and, later, wrote another for the Motorola 6800. He has also used New Micros' MaxForth and Harris RTX packages in a number of applications.

Now retired, Mr. Fergus is heavily involved in a personal severe weather (tornado) warning project which monitors electrical activity from weather fronts. Several of his Forth-based systems (RCA 1802, Motorola 6800, New Micros HC11 or HC16, Harris RTX2001, and Pygmy Forth) continuously collect and analyze data.

Mr. Fergus' development platform consists of Pygmy Forth configured as a host for the other Forth packages. He says, "I like the interactive control and limited restrictions of Forth. It allows me to build a program (language) as I see fit. There seems to be a tendency to demand an 'easier-to-use language.' I like the ability to build an efficient product which might require some 'effort' on my part."

```
 9      'RAM  @  *RAM  !                    ( Separated RAM-EROM )
10      'EROM @  *EROM !
11      A44A 2000 !                         ( Enable SETUP )
12      [ ' SETUP CFA ] LITERAL 2002 ! ;
13 : UNSET ( --- ) FF 2000 C! FF 8000 C! ; ( Disable AUTOSTART )
14 : PGM.S19 ( --- ) 2000 'DP @ S.REC ;   ( Save "PROM program" )
15 DECIMAL ;S

SCR # 10
 0 \                HV CONTROL--READ COUNTERS
 1
 2 \ : V ( n --- )                  V -> HV   Set high voltage
 3 \     B018 ! ;                   Set high voltage ( OC2 )
 4 VAR$ HV : V B018 ! ;"            \ n = v * 65536 / 5.0
 5
 6 \ : C ( --- )                    C -> CNTS Read counters & timer
 7 \    CC 2@ D.                    Read A counter and send
 8 \    D0 2@ D.                    Read B counter and send
 9 \    D4 2@ D. ;                  Read timer and send
10 VAR$ CNTS : C CC 2@ D. D0 2@ D. D4 2@ D. ;"
11 VAR$ (EWORD EEWORD"
12 : EEWORD ( --- ) (EWORD $LOAD ; \ Move MAX definition to EEPROM
13 HV   $LOAD  EEWORD
14 CNTS $LOAD  EEWORD
15 ;S
```

The Elephant Who Refuses to Be Bagged

C.H. Ting
San Mateo, California

Editor's note: For further details about the Lempel-Ziv algorithm and an update to Wil Baden's work with it in Forth, refer to Forth Dimensions *XVI/6 (1995).*

The data compression algorithm known as LZ77[1] is a very simple and elegant method to compress data files. It saves the last 4096 bytes in a circular buffer, and encodes repeating patterns (up to 18 bytes) from the input file with two-byte codes consisting of a 12-bit address and a four-bit byte count. It is very effective and compresses text files to less than one-third the original size.

Wil Baden published a Forth implementation in an earlier FORML Proceedings.[2] It worked quite well for text files containing 10K bytes. However, when I used it to compress image files 32K bytes long, the decompressed images showed streaks of noise not present in the original image. This was very perplexing, because the noise seemed to be quite random and, apparently, data dependent. If the compression/decompression algorithms were defective, one would have expected the decompressed image to be totally broken. However, the image frame was intact, and contained occasional horizontal streaks in random locations.

Careful analysis of the algorithm and the decompressed images led to the conclusion that the encoding and decoding processes were correct, but that some data stored in the circular buffer was disturbed before being retrieved during decompression. The solution was to impose very rigid management discipline when writing to the circular buffer and retrieving data from it, so that the integrity of the circular buffer is maintained throughout the encoding and decoding processes.

The revised algorithm is shown in the accompanying listing.

Wil coded his implementation completely in high-level Forth for portability. I need this compression routine to run as fast as possible on a PC, and thus recoded some of the critical routines in assembly. Optimizing the two words MATCHES and SCAN speeds up the compression by a factor of four.

References

1. J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, 23:3, 337–343, 1977.
2. Wil Baden, "How to Pack an Elephant into a Shopping Bag," *1992 FORML Conference Proceedings*, p 87–92, 1992.

C.H. Ting (tingch@ccmail.apldbio.com) was trained as a chemist specializing in molecular spectroscopy. He abandoned chemistry after discovering his calling in Forth. His most recent interests are real, virtual, and imaginary Forth engines and their applications. These interests leave him very little time for his other hobbies, including the game of Go and Bach's organ music.

Elephant Listing

```
\ OKCOMPRS.SEQ compress .bmp image to .p21 format, 25sep94cht

comment:

07oct94cht
Use a circular buffer (A000-AFFF) for lookBackBuf.
Do 4078-byte searching from current position+18.
Compression ratio is restored to about 4:1.

06oct94cht compres1.seq
```

```
Correct MatchBackBuf##, avoid patterns being overwritten.
Compress ratio is reduced to 3:1, but works correctly.

01oct94cht
Correct MATCHES-. There is still streaking in the recovered pictures.
Use 2 /STRING to scan next match, streaks gone but white dots in hair.
BLANK or ERASE in setup.  ERASE changes blue background.
Build demo1c, demo2c, demo3c and demo4c, verified on P21.

28sep94cht
Add conversion program to convert VGA color to P21 color
        convertToP21 \vpic\demo1e.bmp temp.xxx
        compress temp.xxx demo1e.p21
Scan 4096-18 bytes to find matching pattern.  P21 uses a true
circular 4096 byte buffer, and does not allow overflow.

10sep94cht
copy from LZ77.SEQ  Image compress/decompress
strip .bmp header of 118 bytes.
write to .p21 file, usage:
        compress \vpic\demo1c.bmp demo1c.p21
        compress \vpic\demo1e.bmp demo1e.p21

20sep94cht
Recode SCAN, OUNT and MATCHES
        Compress this file, original lz77 took 20 seconds.
        After recoding SCAN, OUNT and MATCHES, 4 seconds. 21sep94cht
        SCAN- searches a 16-bit pattern instead of an 8-bit pattern.
        OUNT restored to high level, 5 seconds
        MATCHES restored to high level, 8 seconds

16jun88cht
Copy from COPYFILE.SEQ  Copy one file to another
Copy from BLKTOSEQ.SEQ by Tom Zimmer.

The output file uses a handle defined as OUTHCB.  The input file opens a handle on
top of the handle stack by SEQUP, because the LINEREAD routine insists on reading
the file whose handle is pointed to by SHNDL, on top of the handle stack.  At the
end of COPYFILE, this handle is dropped by SEQDOWN.

Algorithm:
J. Ziv and A Lempel, "A Universal Algorithm for Sequential Data Compression," IEEE
Transactions on Information Theory, 23:3, 337-343, 1977
Wil Baden, "How to Pack an Elephant into a Shopping Bag," 1992
FORML Conference Proceedings, p 87-92, 1992.

comment;

empty decimal

handle outhcb                                   \ output file

' 2+ alias CELL+
: /CELL 2 ;

: +UNDER ( n1 n2 n3 -- n1+n3 n2 )
    ROT + SWAP
    ;
```

```
4096 CONSTANT lookBack#   18 CONSTANT lookAhead#
3 CONSTANT breakEven#

$9FFE constant lookBackBuf ( circular buffer from A000 to AFFF)

VARIABLE lookAheadBuf   lookAhead# ALLOT
VARIABLE holding   17 ALLOT
VARIABLE phraseFlag

\ : OUNT ( a -- a+CELL a@ )   DUP @ /CELL +UNDER ;

code OUNT ( a -- a+CELL a@)
    mov     dx, si
    pop     si
    cld
    lodsw
    push    si
    push    ax
    mov     si, dx
    next
    end-code

\ comment:
: MATCHES ( location# address length -- location# matched)
    2>R 0 ( location# matched)
    R> R@ + R> DO ( location# matched)
        2DUP + C@ I C@ -
        IF LEAVE THEN
        1+
    LOOP
    ;
\ comment;

code MATCHES-   cld
                MOV DX, SI      POP CX          MOV  BX, CX
                POP DI          POP SI          PUSH SI
          CX<>0 IF      PUSH ES
                        MOV ES, SSEG
                        REPZ CMPSB
                        0<> IF inc cx THEN
                        POP ES
                THEN
                MOV SI, DX
                sub bx, cx
                PUSH bx
                NEXT            END-CODE

CODE  SCAN2      ( addr len n -- addr' len' )
\ Scan for n through addr for len, returning addr' and len' of n.
\ addr must be between A000-AFFF
                POP AX          POP CX
                JCXZ 0 $
                POP DI
                MOV DX, ES      MOV ES, SSEG
                cld
        1 $:    scasw
                je              2 $
                dec             di              \ backup address
                and             di, $AFFF #     \ circular buffer
```

```
                loop            1 $
        2 $:    MOV ES, DX
                jcxz 3 $
                DEC DI          dec di
                and             di, $AFFF #
        3 $:
                PUSH DI         PUSH CX
                NEXT
        0 $:    PUSH CX         NEXT            END-CODE

\ change VGA color code to P21 color code
hex
create palette
   0 c, 2 c, 4 c, 6 c, 1 c, 3 c, 5 c, 7 c,
   8 c, 0a c, 0c c, 0e c, 09 c, 0b c, 0d c, 0f c,

: changeColor ( n -- n' )
   dup 0F and palette + C@
   swap u16/ 0F and palette + c@
   8* 2* +
   ;
decimal

: change1byte
    pad 1 seqhandle hread
    if pad c@ changeColor
        pad c!
        pad 1 outhcb hwrite drop
    else closeAll
        1 abort" Conversion done."
    then
    ;

: convertToP21 ( sourceFile destFile -- )
        sequp                                   \ Initialize a new handle on the
                                                \  handle stack.
        seqhandle !hcb                          \ input file spec
        outhcb !hcb                             \ output file spec
        cr ." Converting from " seqhandle count type \ announce copying
           ."  to " outhcb count type
        cr seqhandle hopen abort" Open file error"   \ open input file
        outhcb hcreate abort" Create file error"     \ make output file
        0.0 outhcb movepointer                  \ reset file pointer

        pad 118 seqhandle hread drop            \ skip .bmp header
        pad 118 outhcb hwrite drop
        begin change1byte
        again
        ;


: setup  ( setup <inputfile> <outputfile> <return> )
        sequp                                   \ Initialize a new handle on the
                                                \  handle stack.
        seqhandle !hcb                          \ input file spec
        outhcb !hcb                             \ output file spec
        cr ." Read from " seqhandle count type  \ announce copying
           ." , write to " outhcb count type
        cr seqhandle hopen abort" Open file error"  \ open input file
```

```
        outhcb hcreate abort" Create file error"    \ make output file
        0.0 outhcb movepointer                      \ reset file pointer

\        118. seqhandle movePointer                 \ skip .bmp header

        0 lookBackBuf !
        lookBackBuf CELL+ lookBack# lookAhead# + 1- BLANK
        lookAhead# lookAheadBuf !
        0 holding CELL+ C!
        1 holding !
        128 phraseFlag !
        ;

: runOut ( -- )
        holding @UNT  outhcb hwrite
        holding @ - ABORT" Write file error"
        0 holding CELL+ C!
        1 holding !
        128 phraseFlag !
        ;

: ReadAhead ( unmatched -- lookAheadBuf@)
    >R
    lookAheadBuf @UNT
    R@ /STRING seqhandle hread
    DUP 0= IF CR ." Read file end" THEN
    R> + ( lookAheadBuf@) DUP lookAheadBuf !
    ;

: MatchLookBack## ( -- location matched)
    lookBackBuf @UNT + 18 + ( location to start searching)
    DUP 2 ( breakEven# 1-) 2>R
    4078 ( location length )
    BEGIN ( location length)
        lookAheadBuf CELL+ @ SCAN2
\       lookAheadBuf CELL+ C@ SCAN
        DUP
    WHILE
        2DUP ( . . location length)
        lookAheadBuf @UNT ROT MIN
        MATCHES- ( . . location matched)
        DUP R@ > IF
            2R> 2DROP   2DUP 2>R
            DUP lookAheadBuf @ = IF
                2DROP 2DROP ( )
                2R> ( location matched)
                EXIT ( . . location matched)
            THEN
        THEN    2DROP ( location length)
        2 /STRING SWAP $AFFF AND SWAP
    REPEAT  2DROP 2R>
    ;

: PutMatchingPhraseCode ( location matched -- matched)
    DUP breakEven# < IF
        2DROP ( )
        lookAheadBuf CELL+ C@ holding @UNT + C!
        1 holding +!
        1 ( matched) >R
```

```
    ELSE ( location matched)
        >R ( location) ( lookBackBuf CELL+ - ( position)
        4095 AND
        R@ breakEven# - 4096 * + FLIP
        holding OUNT + ! ( r)
        2 holding +!
        phraseFlag @ holding CELL+ C@ OR holding CELL+ C!
    THEN ( )
    phraseFlag @ 2/ ( phraseFlag@)
    ?DUP IF phraseFlag !
    ELSE ( ) RunOut THEN
    R> ( matched)
    ;

: UpdateLookBackBuf! ( matched -- )
    lookAheadBuf CELL+ DUP +UNDER DO ( )
        I C@ lookBackBuf OUNT + C!
        lookBackBuf @ ( n)
        DUP lookAhead# 1- < IF
            I C@ lookBackBuf OUNT + lookBack# + C! ( n)
            THEN
        1+ dup lookBack# >=
        if ." ."
        then
        $AFFF AND lookBackBuf ! ( )
    LOOP
    ;

: ShiftLookAheadBuf ( matched -- unmatched)
    >R ( )
    lookAheadBuf OUNT ( address lookAheadBuf@)
    R> /STRING ( address unmatched)
    >R ( address)
    lookAheadBuf CELL+ R@ CMOVE ( )
    R> ( unmatched)
    ;

: compress
    setup
    0 ( unmatched)
    BEGIN
        ReadAhead ( lookAheadBuf@)
    WHILE ( )
        MatchLookBack## ( location matched)
        PutMatchingPhraseCode ( matched)
        DUP UpdateLookBackBuf!
        ShiftLookAheadBuf ( unmatched)
    REPEAT ( )
    RunOut
    closeAll
    CR ." Compress done."
    ;

comment:

Decompression is to reverse the compression process.
Read a phraseFlag byte
Repeat 8 times:
    If phraseBit is 0, read next byte
        Append byte to output file
        Append byte to lookBackBuf
```

```
    Else read next word and decompose it to address and length
        Find string in lookBackBuf
        Append string to output file
        Append string to lookBackBuf
Repeat until done

comment;

: getc ( -- c )
    PAD 1 seqhandle hread
    0= ABORT" End of read file."
    PAD C@
    ;

: OutputByte ( -- )
    getc
    lookBackBuf OUNT + C!
    PAD 1 outhcb hwrite
    0= ABORT" Write file error."
    lookBackBuf @ ( n)
    DUP lookAhead# 1- < IF
        PAD C@ lookBackBuf OUNT + lookBack# + C! ( n)
        THEN
    1+ dup lookBack# >=
    if ." ." then
    $AFFF AND lookBackBuf ! ( )
    ;

: OutputString ( -- )
    getc dup 16 / breakEven# + >R ( length)
    15 AND 256 * ( high nibble of address)
    getc + lookBackBuf CELL+ + R> ( address length)
    2DUP outhcb hwrite drop ( write to output file)
    OVER + SWAP DO
        I C@ lookBackBuf OUNT + C!
        lookBackBuf @ ( n)
        DUP lookAhead# 1- < IF
            I C@ lookBackBuf OUNT + lookBack# + C! ( n)
            THEN
        1+ dup lookBack# >=
        if ." ." then
        $AFFF AND lookBackBuf ! ( )
    LOOP
    ;

: decompress ( -- )
    setup
    BEGIN PAD 1 seqhandle hread
    WHILE PAD C@ ( get phraseFlag)
        8 0 DO
            DUP 128 AND
            IF OutputString
            Else OutputByte
            THEN
            2*
        LOOP
        DROP ( discard phraseFlag)
    REPEAT
    closeall
    CR ." Decompression done."
    ;
```

4tH, an Experiment in C

Hans Bezemer
The Hague, Netherlands

History

The first time I encountered Forth was back in the eighties, when I was using a Sinclair ZX Spectrum, the European equivalent of the Timex TS2000. In those days, I was into learning as many computer languages as I could. Fortunately, most were available for the Spectrum, including Forth. The only drawback was that many compilers did not leave much room for code, since they filled most of the available RAM. Except Forth. Forth was a very small, but very complete programming environment. There were some serious drawbacks, too. It had pretty bad documentation (so I didn't understand too much about the language), the editor was awkward to use on a 32 × 21 display, and it wrote blocks to tape.

Later, I bought myself a real floppy-disk drive. The only problem was that Forth didn't support that device. So I disassembled the entire Forth compiler and rewrote the I/O routines. To my own surprise, Forth ran as if it was designed for disk. And I slowly developed a taste for this strange, but powerful, language. It was very easy to interface assembly with Forth. One could decompile the code. And, since I could write Forth programs in an external 64-character-wide editor, that problem was also taken care of.

After my Spectrum was shelved, I had an occasional look at Forth but didn't write any serious programs in the language anymore. I rather wrote them in C. But Forth was not dead yet. I would return through the back door.

It all began with a function that could evaluate a simple arithmetic expression. Since I wanted a small and compact program, I used Reverse Polish Notation. It worked fine until I ran into a problem that it couldn't handle. I needed something far more powerful. I thought about it for quite a while, but I just didn't get the right ideas. I ran into other scripting languages, but they were not quite what I wanted. I wanted a small package with a very simple API and flexible memory management. It should be highly portable and easy to program. Finally, it shouldn't crash when a programming error was made. Instead, it should return to the calling program with the appropriate error code and the location where the error was detected.

There were several reasons why I finally came up with a Forth-like implementation. First, parsing Forth is pretty straightforward. It's easy to write and easy to maintain. Second, Forth is quite fast and doesn't require a lot of memory. Third, Forth requires only a handful of primitives, and I could catch all errors there and then. Finally, Forth is a well-documented language and users can tap from these sources.

It took me eighteen months to come up with a toolkit that met most of these requirements. The first version had a very primitive parser that required a non-standard implementation of the `."` word. Source and object had to remain in memory concurrently, and it was not possible to save object code. Apart from `."`, this version of 4tH had no string capabilities. A special word allowed the user to input numbers. All output was generated by `printf()` calls. A single routine resolved all branching, but although it was small and didn't require a stack, it had a lot of drawbacks. First, the syntax had to be non-standard to make it work. Second, it required a second pass. Third, it was unable to detect all syntax errors. The implementation of `HEX`, `DECIMAL`, and `OCTAL` was outright clumsy. It was loosely based on the Forth-79 Standard.

The second version had limited string capabilities, and object code could be saved. Because it merely dumped chunks of memory, the object code was not portable. However, it completely mimicked the way Forth generates and outputs numbers, and all output was channelled through a function called `emit()`. The implementation of `HEX`, `DECIMAL`, and `OCTAL` was quite solid now.

The third version, which is described here, had a very intelligent parser, which made a fully Forth-compatible implementation of `."` possible. Furthermore, it resolved branching in the same way Forth does, thus eliminating the second pass. The object code was saved in a machine-independent format, making it portable across virtually all platforms. The string capabilities were further enhanced. Finally, most of the Core Wordset of ANS Forth was supported.

What is 4tH?

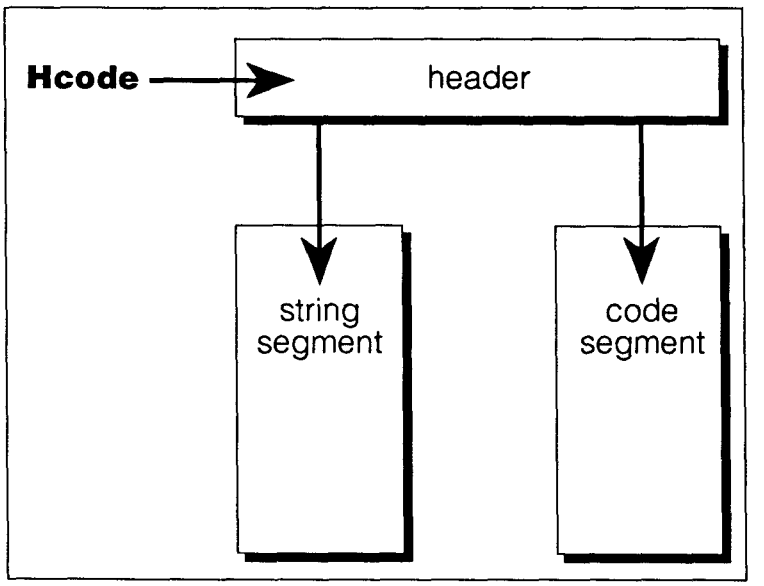
4tH is very different from all other Forth implementations. One might find some of its characteristics in other

Forths, but this combination is unique.

4tH contains a lot of Forth, although it is translated to C. 4tH contains two interpreters, just like ordinary Forth, but they behave quite differently. 4tH can be called token-threaded, but it has no dictionary.

In fact, 4tH is a Forth using conventional compiler technology. It is fast, compact, and highly portable. 4tH is completely written in C and has so far been ported to MS-DOS, MS-Windows, and several Unixes without any problems.

Before you turn away in disgust, note that 4tH has some interesting properties. 4tH isn't a standalone compiler—it is a library, designed to be called from a C program. But if you want to make a compiler, you can do that in a few lines of C. 4tH produces bytecode (like Java) which can be used without any modification on every platform 4tH has been ported to. 4tH supports both the Forth-79 Standard and ANS Forth, although some deviations from these standards were necessary in order to simplify the language.



How Does it Work?

Forth usually has two interpreters, the text interpreter and the address interpreter. The text interpreter passes strings from the terminal or from mass storage and looks up each word in the dictionary. When a word is found, it is executed by invoking the second level, the address interpreter. Some Forth words change mode from interpreting to compiling, or vice versa. Others always execute.

4tH works fundamentally differently. The text interpreter essentially just compiles. The only words executed there are the words that always execute, called IMMEDIATE words. Everything else is simply compiled. Therefore, the text interpreter is called *the compiler*. The address interpreter works just like Forth, although there are no words that can do any compiling. It will be referred to as *the interpreter*.

4tH source can be stored in files, environment variables, static strings, etc. As a matter of fact, 4tH can compile anything that can be converted to a string stored in dynamic memory. Each compilation creates a standalone structure in memory that can be reused, discarded, or swapped to mass storage.

The API, which will be discussed in more depth later on, is therefore very simple. Pass a string to the compiler and it will return a pointer to a structure, which is called *H-code*. You can pass this pointer to several other functions that will either interpret it, save it to mass storage, free it from memory, or decompile it.

The compiler first calls a pre-parser. This replaces all white space with null characters, and counts the number of words and the number of strings. This information is used by 4tH to allocate its resources. Since 4tH is kept as small and simple as possible, no word uses more than a single token. And, while 4tH now knows how many words there are, it can safely allocate its token space.

The compiler also uses a symbol table. This is set to a minimal size in order to allow small programs to compile without any errors. After that, the number of words determines how large the symbol table will actually be. A special stack is created for all flow control; since the size of this stack depends only on the level of nesting required, the size is fixed.

Strings are handled in a special way. They remain in the memory allocated to the 4tH source, but are shifted to the front during compilation. After compilation, the symbol table and the branch stack are discarded, and the token space and string space are shrunk to their actual size. Note that all of this is fully transparent to the programmer.

The interpreter is even simpler. It just matches the tokens with the corresponding piece of C code, and executes until there are no more tokens left to execute. It returns the top of the stack to the calling C program. C variables and constants can be transferred to the interpreter. Inside 4tH, they appear as 4tH variables which can be used like any other 4tH variable.

4tH is fast. It compiles up to 50,000 lines per minute on a humble Intel 80486/33 MHz. Benchmarks prove it executes code up to four times faster than conventional C-based Forths. 4tH uses very little memory. With only 40K memory to spare, it compiles 20K of source and still has 10K left free when all resources are allocated. The resulting H-code takes up no more than 10K, even under the worst alignment possible. 4tH is compact. The same 20K source is compiled and saved to mass storage as a 7K binary file.

H-code

H-code is a very complex, three-part structure. First, the header: The header describes the run-time environment (created by the interpreter), the tokenspace, and stringspace. When H-code is created (e.g., by the compiler), a pointer to the header is returned.

Second, the token space, which is called the Code Segment. A 4tH token is also a structure. It consists of the actual token and an optional argument. E.g., the token is a BRANCH instruction and the argument is the address it has to jump to. Since about 50% of the tokens require an argument, overhead is kept to a minimum.

Finally, the string space, which consists of ASCIIZ strings and is called the String Segment. The argument of the corresponding token is simply an offset to the beginning of the String Segment.

Another major format used by the 4tH toolkit is called the *H-code executable* or HX file. An HX file is not simply an image of the structure in memory. Since there are many different compilers and machine architectures, HX files would not be portable. In fact, they are portable. A 4tH source file can be compiled to by a 4tH compiler running under Unix and the resulting HX file can be run by the MS-Windows version of the 4tH interpreter.

An HX file contains all information about the 4tH environment, including version and release information. This prevents incompatible HX files from being read by the wrong application. The Code Segment is packed. All numbers are saved in a machine-independent way. The String Segment is saved as is.

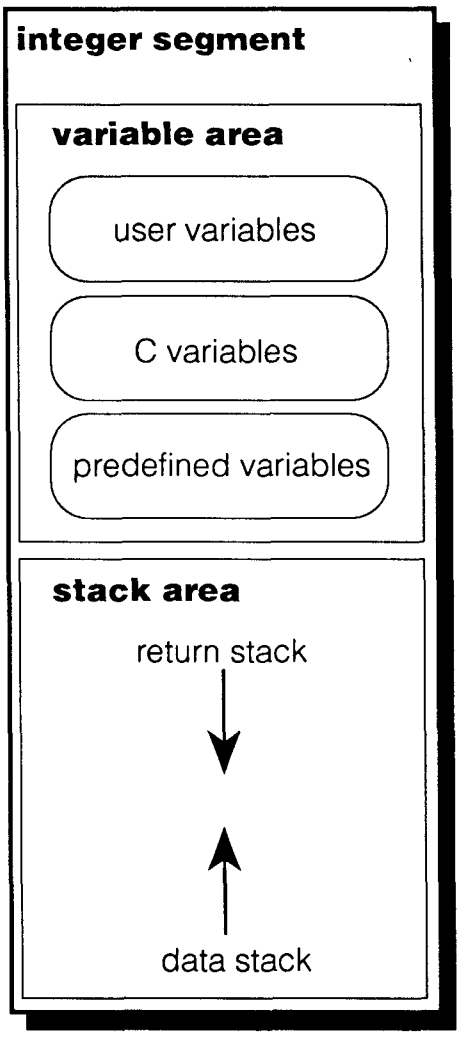
The Run-Time Environment

The interpreter creates two more segments. First, the Integer Segment, which is divided into two major areas. The Stack Area contains both the Return and the Data Stack. The Data Stack grows upward and the Return Stack grows downward. This means application programs with different requirements can run without having to modify the interpreter.

The Integer Segment also contains the Variable Area. All variables and values are stored here. To the application programmer, it is fully transparent whether a variable is predefined, preloaded, or created by the application program. The same words are used to access them.

The second segment is called the Character Segment and contains the TIB, PAD, and allocated memory. This is the place where strings, buffers, and such can be found.

When the program has executed, these segments are freed. Only the final value on the stack is returned to the calling C program. A program can terminate because there are no more tokens to execute or because an error occurred. 4tH checks everything: output, storing, fetching, stack. In theory, no application can bring it to its knees.



The Language

Although 4tH's architecture differs a lot from any classic Forth implementation, the language is much the same. Deviations from the standard were reluctantly made, and only when necessary. But 4tH had to be easy to use, too. And Standard Forth has—at least in the view of the author—some hard-to-understand constructs.

A user wants to create his program. He doesn't want to be bothered with whether a number is greater than 65,535. When he programs in standard Forth and such is the case, he has to switch to the double-word set. If he alters his program, he might have to rewrite a considerable portion of it!

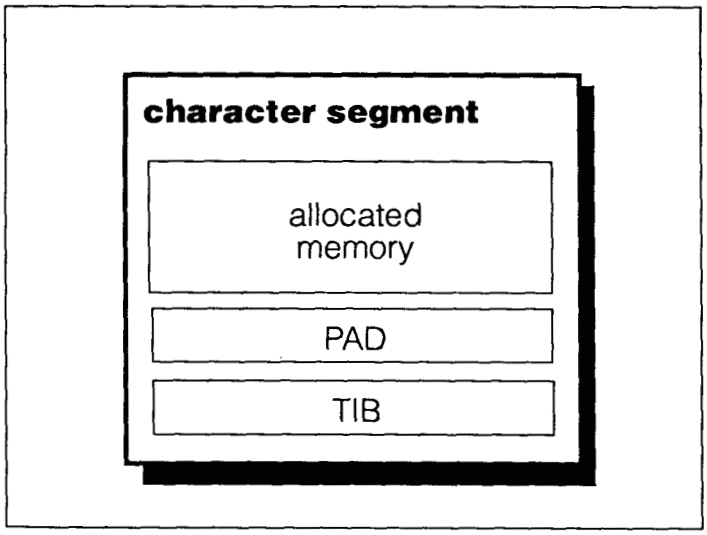
In 4tH, signed 32-bit numbers are used. Nothing else. There is no double-word set or mixed-word set. If conversion is necessary (like offsets or characters), it is done transparently. There is also no floating-point word set. If one is needed, it has to be added to 4tH.

You also won't find a `<BUILDS DOES>` construct. The reason for this is twofold. First, it would make the compiler far more complicated. Second, a poll among members of the Dutch Forth User Group showed that, although most users know about its existence, few are able to apply it properly! So why bother? If necessary, it can be emulated anyway.

In order to accommodate 4tH's segmented structure, some modifications had to be made. Every segment has its own fetch, store, and allocate commands. The segments created by the compiler are read-only. The following table shows which words act on which segment.

	String	Code	Integer	Character
Fetch:	COPY	'@	@	C@
Store:	N/A	N/A	!	C!
Allocate:	N/A	N/A	ALLOT	ALLOCATE

The word '@ is used to fetch the argument of constants



arrays in the Code Segment. Constant arrays are created by the word CREATE, e.g.,

```
CREATE LIMITS 220 , 340 , 100 , 190 ,
```

The is the only allowed use of the word CREATE. To fetch the second element of LIMITS one has to write:

```
LIMITS 1 + '@
```

Arrays of variables are created by issuing:

```
VARIABLE SPEED 10 CELLS ALLOT
```

The second element of SPEED can be initialized by:

```
15 SPEED 1 CELLS + !
```

Which is quite natural to native Forth users. But note that words like CELLS and CHARS are just dummies, added to make porting 4tH code easier. This may raise some eyebrows:

```
80 ALLOCATE CONSTANT A_STRING
```

The word ALLOCATE allocates a number of bytes in the Character Segment and leaves the starting address of the space allocated on the stack. Thus, A_STRING points to the beginning of an 80-character string. 4tH allows you to write something like this:

```
CREATE A_STRING 80 CHARS ALLOT
```

But this will not do what a die-hard Forth programmer might expect. In fact, it will create a constant A_STRING that points to the offset in the Code Segment where it was compiled and add 80 more variables to the last-defined variable or value.

Finally, deep stack manipulators or CASE constructs are not implemented, since the author considers it bad style. Leo Brodie agrees!

A Peek Under the Hood

The key to this is the compiler, which tries to mimic Forth but works very differently. Let's take a look at this small piece of sourcecode:

```
cr
begin
      times @ 1- dup times !
until
```

This will compile into:

```
[62]  CR          (0)
[63]  VARIABLE    (2)
[64]  @           (0)
[65]  1-          (0)
[66]  DUP         (0)
[67]  VARIABLE    (2)
[68]  !           (0)
[69]  0BRANCH     (62)
```

The bracketed numbers represent the offset in the Code Segment, followed by the name of the token and the argument within parentheses. It shows that compilation of this particular piece of code started at offset 62 in the Code Segment. The variable TIMES is the third compiled variable. And although it seems that 0BRANCH branches back to offset 62, it doesn't. After the Instruction Pointer is set to 62, it will be incremented, so it will actually branch to offset 63. Why? Because it made the interpreter faster without adding too much overhead to the compiler. Remember, it is a single-pass compiler!

All defining words, like :, VARIABLE, VALUE, CONSTANT, etc., add an entry to the symbol table. The symbol table is a very simple structure, containing nothing but the name, the token, and its argument. When the name is encountered, it is simply replaced by the token and the argument. CONSTANT will add an entry containing the LITERAL token and its value. VARIABLE will add an entry containing the VARIABLE token and the current number of variables. ALLOT simply increases the number of variables. The total number of variables will be saved in the header, so the interpreter will know the size of the Variable Area it has to create.

The word : will add a symbol-table entry containing the offset in the Code Segment to the defined word and a CALL token. The CALL token pushes the current value of the Instruction Pointer on the Return Stack and jumps into the defined word. In order to prevent the program from entering the word before it is called, a BRANCH instruction is compiled that jumps over the defined word. At the end of the defined word, you will find an EXIT token that restores the value of the Instruction Pointer. Since the Instruction Pointer is incremented afterwards, it continues execution after the CALL token.

This explains why ' NAME EXECUTE works. ' NAME compiles to a LITERAL token containing the argument value of the symbol table entry of NAME. EXECUTE pushes the current value of the Instruction Pointer on the Return Stack, and puts the TOS in the Instruction Pointer. EXIT never knows the difference. Since all built-in words cannot be found in the symbol table, you cannot compile:

```
' +
```

But:

```
: _+ + ; ' _+
```

is perfectly valid.

All compiling is done in memory, so some pretty dirty tricks can be used. One of them is the *literal expression*. A literal expression is simply an expression that compiles to a LITERAL token. That includes all numbers, all constants, and expressions like CHAR &, ' name, and 80 ALLOCATE. A word like CONSTANT expects a literal expression, like:

```
65 CONSTANT MAX_SPEED
```

The number 65 will compile to a LITERAL token with the argument containing 65. CONSTANT removes that literal from the compilant and uses it to create the correct symbol table entry. ALLOCATE also expects a literal expression, and will compile to a literal expression itself.

This also means that something like:

```
55 10 + CONSTANT MAX_SPEED
```

won't compile, since + is not a valid literal expression. Of course:

```
CHAR & CONSTANT AMPERSAND
```

is. The word , (comma) does something similar. It, too, expects a "literal expression." But instead of erasing the token from the compilant, it changes it from LITERAL to NOOP without affecting the argument.

All branching is resolved by the C equivalent of ?PAIRS, BACK, etc. A special stack is created at compile time. It contains the offset to the corresponding token and a reference. Nothing special here, except that absolute offsets are used instead of relative ones.

The API

The first thing that has to be done when using the 4tH toolkit is including the header file and defining a pointer to the H-code:

```
#include <4th.h>
Hcode *Compilant;
```

The next thing is to create a source and compile it. Although the 4tH toolkit provides functions to enable the programmer to read and compile more complex structures, this example is kept pretty straightforward:

```
Compilant = comp_4th (strdup (".\" Hello
world\" cr"));
```

This will compile the classic "Hello world" program. That's all. The variable Compilant now contains a valid pointer to the compiled program. No cleaning up is necessary. But what if something has gone wrong? Right. If comp_4th() couldn't compile a thing, it returns a NULL pointer. If it could compile something, it returns that something. But that something might still not be everything..

In order to give the programmer full control, an error messaging system is included. It looks very much like the familiar errno. There is a predefined variable called ErrNo4th. If ErrNo4th does not equal zero, something is wrong. There is an string array called ErrList4th which optionally can be linked in. If ErrNo4th is used as an index to ErrList4th, the proper error message will be selected.

There is a third predefined variable, called ErrLin4th. It contains the offset to the token in the Code Segment where things went wrong. All these variables work in all API functions.

Next, the program has to be executed. Since we're all experts here, errors are checked. The final value on the stack is discarded. A single value (100) is transferred to the interpreter:

```
if (!ErrNo4th) (void) exec_4th
(Compilant, 1, (cell) 100);
```

This can be executed as many times as needed. The compilant is still in memory. In order to free the compilant this statement is added:

```
free_4th (Compilant);
```

Why no error checking? The function free_4th() does all checking, so none is needed. Saving, loading, and decompiling is just as easy.

Why 4tH?

It is clear that 4tH has its own niche. It is not a replacement for a full-fledged ANS Forth compiler. But if you need an easy-to-use, fast, safe, and economical scripting language, 4tH might be your first choice.

The language can be extended or modified easily. Arrays of H-code pointers create new possibilities—like compiling, saving, freeing, and reloading 4tH applets as you go, by simply building a swapping API on top of the existing one. Or pushing it even further by removing the run-time checks. Everything is possible with a little imagination.

Most of all, I wanted to show that the combination of classic compiler technology and Forth technology can create something that surpasses the limitations of both. If I fired the imagination of somebody out there, it was all worthwhile.

4tH will eventually be available for FTP on taygeta.com but, for the moment, it can be obtained from the author directly.

Hans "the Beez" Bezemer (hbezemer@vsngroep.nl) graduated from college in 1983, with a degree in biology and geography. After a very short carreer in education, he worked as an application programmer with the Dutch Ministery of Transport, Public Works, and Water Management. In 1987, he switched to the VSN groep, a company that provides about 75% of Holland's public transit services. In 1990, he became that company's Technology and Systems Management Advisor. Since January 1st, 1996, he took another job, still within the same company: Information Manager of the Finance Department. Privately, Hans has published some shareware programs, and has been using Forth and C since the mid-eighties. He was born in 1960 and lives in Den Haag, better known in English as The Hague (where the international tribunal on former Yugoslavia is held) in the Netherlands.

Forth Solves 150-Year-Old Problem:

Breaking Code

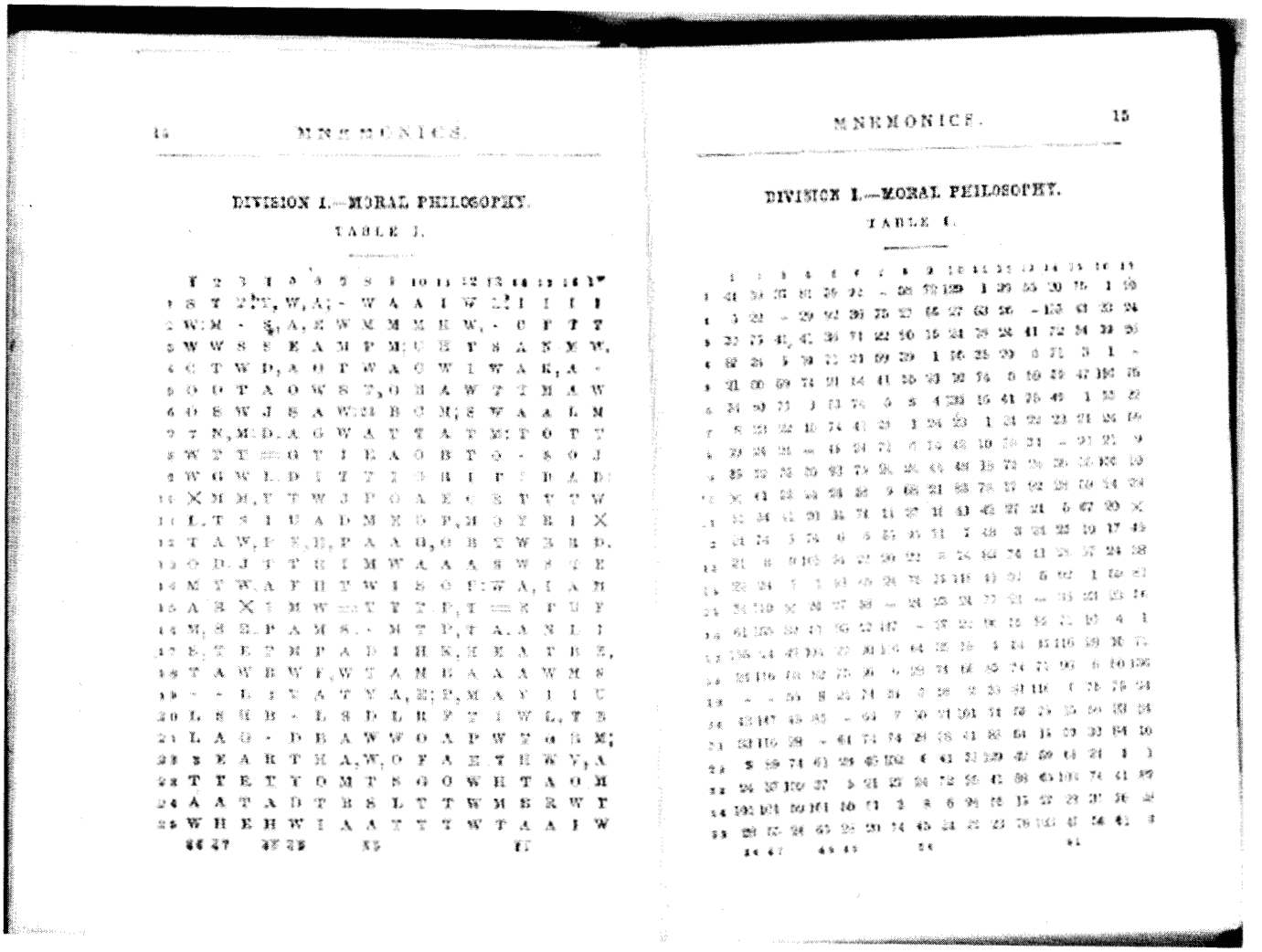
Wil Baden
Costa Mesa, California

This month's article describes an unusual application—unusual for Forth or any other programming language.

I have in my possession a little book, 3½ by 5½ inches, 136 pages, printed in the early 1850s. It fits easily in most shirt pockets. Except for the title page and five blank pages, each page consists of a matrix of 25 numbered lines by 17 numbered columns. On the left-hand pages the elements of the matrix are filled with a capital letter, a small number, or a special symbol. On the right-hand pages the elements are filled with a one-to-three digit number, the letter "S", or a special symbol matching that on the left-hand page. Sometimes a punctuation character is with a letter.

As you have already surmised, the book is written in code. The code book is a 36-page pamphlet with lists of numbered words—a list for every letter but "X".

To read the book, you take a letter from the left-hand page and the number from the corresponding row and column on the right-hand page, put them together, and look the result up in the code book. Thus, T24 is "the", O21 is "of", A74 is "and".



14 MNEMONICS.

DIVISION I.—MORAL PHILOSOPHY.

TABLE I.

MNEMONICS. 15

DIVISION I.—MORAL PHILOSOPHY.

TABLE I.

Some of the text runs in columns, top to bottom. On the same page text also goes in columns, bottom to top, in the same columns with text going top to bottom. Elsewhere the text goes in rows, left to right.

This month's article describes the ad hoc tools, written in Forth, used to decode and make a translation of the book.

I do not own the book, but it was lent me to make the translation. Because of the enormous effort to read the book, it is believed that, after almost 150 years, this is the first complete translation. Without a computer it is just too laborious.

Forth is well suited for this application. As I discovered more and more things about the contents of the book, the program was modified accordingly. The interactivity and fast compilation of Forth was vital.

Also, my over-the-counter stock of pet words is designed for this kind of application.

Step I
Enter the Vocabulary

The first step was to transcribe the vocabulary.

To make it easier, ENTER first opens a file with a single letter as name for output, and then repeatedly displays the letter, the next sequence number, a space, the letter again, and waits for keyboard input. After receiving input everything is echoed to the output file.

```
t1 tTabernacle
    ----------
t2 table
    ----
t3 take
    ---
t4 takes
    ----
t5 t
```

(My input underhyphened.)

The first letter is displayed so that all you have to do is complete the word. With words of high frequency in English, my fingers would type the first letter anyway, so ENTER was modified to take care of this. This also let me capitalize words I felt should be capitalized. Just a Return by me closes the output file, sets up for the next list, and exits.

The filenames were single letters. Before doing ENTER, I could change the letter by "CHAR *n* first-letter C!" when I wanted.

To see what had been written, I used LISTED.

Finally, I used the [code in Figure One] to catenate the individual files.

Step II
Load the Vocabulary

Loading the vocabulary places each word into dataspace. When the initial letter changes, the word's address is put into WORD-POINTERS so the word can be found by its first letter and sequence number.

The vocabulary takes 20K of dataspace for 2514 words.

To decode a word, such as "S226.", find where in dataspace the words with that first letter begin, and do "COUNT CHARS +" one less than sequence-number times. (S226 is the worst case.) To the string whose address you now have, append the rest of of the string you're decoding.

This simple method was chosen because I was impatient to see results. I later revised the method to remember where each word was, and so have a much faster lookup. This much more efficient method reduced the time to translate 30,000 words from eight seconds to six seconds. Whoopee. So it's not worth bothering with.

TEST was used interactively to test the lookup.

For example,

```
TEST T24 O21 A74, T59 S226. ;;

should yield
the
of
and,
to
sight.
```

Step III
Decode a Word

The book had to be transcribed. This is what the transcription looks like.

```
[P. 88]
[1]  N  I  T  1st T  F  A  G  M  T  C
     37 72 24 S   55 43 56 27 40 59 73
[3]  T  T  A  O  T  P. - A  I
     59 24 51 21 25 10 - 74 75
[10] C  T  P. -
     6  59 14 -
```

Et cetera.

Here is the translation.

> [88.1] Now is the 1st time for all good men to come [88.3] to the aid of their parties.
>
> And it [88.10] came to pass.

There were approximately 30000 words to transcribe. This took me ten days. I did a double-page at a time, and checked the translation after each page. As I discovered

Figure One. Catenating the files.

```
S" vocabuly" <== OUT
:: S |OUT << LISTED| a b c d e f g h i j k l m n o p q r s t u v w y z
;; CLOSE OUT
```

things, the program was modified. I'd type the column or row number within brackets, a line of letters from the left-hand page, and the corresponding line of numbers from the right-hand page. I'd then use Text-to-Speech to proof-read the numbers. I'd verify that the letters and numbers were matched one-to-one.

Page numbers were given when the page changed. Note that columns don't have to be consecutive. There is a hard-to-see mark to show which columns to do.

Any errors I made or that were in the original were glaringly obvious. Because of the code book, there were no spelling errors, and a wrong word would show up as nonsense. (Exception: "Who" for "Whom" in one place.)

I apologize that I am obligated not to reveal any of the hidden mysteries in the book. After all it was written in code to keep and conceal them.

The title page in plain English is meant to obfuscate.

```
W R I T T E N   M N E M O N I C S
        ILLUSTRATED BY COPIOUS EXAMPLES FROM
      Moral Philosophy, Science and Religion
```

Step IV
Format Text

The output has to be properly formatted for sentences and paragraphs. These are typical text-formatting words.

The first version of +TYPE was

```
: +TYPE     DUP MORE    TYPE ;
```

The rest of the definition was added as the need was seen.

Step V
Parse a Line of Code

The heart of decoding is merging the letters and the numbers. PARSE-WORD is used to pick up the numbers and other codes from the second line. TOKENIZE is used for the letters and symbols from the first line.

TOKENIZE identifies the next graphic string from a character string, and leaves the rest of the character string for later.

SSKIP and SSCAN are implementation factors of TOKENIZE. They are like F83's BL SKIP and BL SCAN except any invisible character is considered instead of just BL.

PAGE.LINE inserts page and line number in the form "[page.line] ".

DECODE-LINE recognizes the following codes:

n (a number)
S
- or --
.. or **
x
(added by me for superscripts, *etc.*, in the text)

Step VI
Decode a File

DECODED is the application.

In the transcription there are these kinds of lines apparent:

blank line;
remark (a line not beginning with "[");
page number (a line beginning with "[P")
a left-hand column (begins with number in brackets);
a right-hand column (line after a left-hand column).

If you have a left-hand column or row, you then read a right-hand column or row, and merge the two with DECODE-LINE.

Appendix

In ThisForth the user input device and user output device are implemented internally as variables for Standard C Library file pointers. The default values are standard input and standard output. Forth fileids are Standard C Library file pointers.

"*fileid* STREAM" saves the current value and associated status of the user input device on a short stack, and assigns fileid for the user input device.

UNSTREAM restores the previous value and status. "0 STREAM" is used to assign standard input. SOURCE-ID returns the current value for the user input device, with 0 being returned for standard input.

"*fileid* DISPLAY" assigns fileid for the user output device. "0 DISPLAY" assigns standard output for the user output device.

The following are convenience words for file handling.

```
: OPENED  OPEN-FILE ABORT" Can't open " ;
: INPUT   R/O OPENED ;
: OUTPUT  W/O OPENED ;
: CLOSED  ?DUP IF CLOSE-FILE
          ABORT" Can't close. " THEN ;
```

Because STREAM assigns the user input device,

```
REFILL   SOURCE     WORD  PARSE
PARSE-WORD   CHAR
```

can be used for input from any file. For example, see the definitions of LISTED and LOAD-VOCABULARY.

DISPLAY gives us formatted output to a file. The following can all be used:

```
EMIT  TYPE   CR   ."   .   U.  .R  U.R
```

et cetera.

Many stock words in ThisForth are macros. Here are some useful ones.

```
?? a-word
```

becomes

```
IF a-word THEN
```

```
n TH a-word
```

becomes

```
n CELLS a-word +
```

```
S=
```

becomes

```
COMPARE 0=
```

This lets pinhole optimization work.

The following are useful for file handling.

fileid << *a-word*
becomes
fileid DISPLAY *a-word* 0 DISPLAY

CLOSE *foo*
becomes
foo CLOSED 0 TO *foo*

filename <== *foo*
becomes
filename CLOSE *foo* OUTPUT TO *foo*

Thus, to copy a file:

```
S" newfile" <== OUT
S" oldfile" OUT << LISTED
CLOSE OUT
```

To save the output of WORDS:

```
S  wordlist.txt <== OUT
OUT << WORDS CLOSE OUT
```

==> and >> work similarly for input.

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

```
 1 ( BREAKING CODE                          Wil Baden 1996 )

 3 ( STEP I.  Enter the Vocabulary. )

 5 VARIABLE line-number  0 line-number !
 6 CREATE first-letter   [CHAR] a C,
 7 0 VALUE OUT ( Fileid )

 9 : enter                              ( -- )
10      first-letter 1 <== OUT
11      BEGIN                                 ( )
12           1 line-number +!
13           first-letter C@ EMIT
14           line-number @ .

16           first-letter C@ EMIT

18           REFILL DROP
19           SOURCE                           ( s .)
20           DUP
21      WHILE
22           OUT DISPLAY
23                first-letter C@ EMIT
24                line-number @ .

26                OVER C@ BL OR first-letter C@ <> IF
27                     first-letter C@ EMIT
28                THEN

30                TYPE                        ( )
31                CR
32           0 DISPLAY
33      REPEAT                         (     s .)
34      2DROP                                 ( )
35      CLOSE OUT
36      1 first-letter C+!
37      0 line-number !
38 ;

40      : LISTED  2 needed             ( filename . -- )
41           INPUT STREAM                     ( )
42                BEGIN REFILL WHILE
```

```
43                      SOURCE TYPE CR
44                  REPEAT
45              SOURCE-ID UNSTREAM CLOSED
46      ;

49 ( STEP II.      Load the Vocabulary )

51 CREATE word-pointers   32 0 DO   0 ,   LOOP

53 : to-letter-code     31 AND ;

55 : load-vocabulary                    ( -- )
56      0 first-letter C!
57      S" vocabuly" INPUT STREAM
58          BEGIN REFILL WHILE
59              PARSE-WORD                  ( s .)
60              OVER C@ to-letter-code first-letter C@ <> IF
61                  OVER C@ to-letter-code first-letter C!
62                  HERE first-letter C@ TH word-pointers !
63              THEN
64              2DROP PARSE-WORD
65              S,                      ( )
66          REPEAT
67      SOURCE-ID UNSTREAM CLOSED
68 ;

70 LOAD-VOCABULARY

72 ( STEP III.      Decode a Word. )

74 : decode-word                  ( code . -- decode . )
75      OVER C@ to-letter-code TH word-pointers @
76      ROT ROT                         ( addr code .)
77      1 /STRING 0. 2SWAP >NUMBER      ( addr n . code .)
78      2>R DROP                        ( addr n)( R: code .)
79          ?DUP 0= IF
80              COUNT DROP 1
81          ELSE
82              1 ?DO COUNT CHARS + LOOP
83              COUNT                   ( addr .)
84          THEN
85      2R> S+                          ( decode .)( R: )
86 ;

88 : test please " :: s |decode-word type cr| " ;

91 ( STEP IV.  Format Text. )

93      : is-digit      [CHAR] 0 -      10 U< ;
94      : is-lower      [CHAR] a -      26 U< ;
95      : is-upper      [CHAR] A -      26 U< ;
96      : is-visible    [CHAR] ! -      94 U< ;
97      : to-lower      DUP is-upper IF     BL +  THEN ;
98      : to-upper      DUP is-lower IF     BL -  THEN ;

100     72 VALUE LL     ( Line Length )
```

```
101      VARIABLE col    ( # Print-Columns Written )
102      VARIABLE CAP    ( Flag for Capitalization )

104           : end-of-sentence          ( s . -- flag )
105                     1- CHARS + C@         ( last_char_of_word)
106                     CASE [CHAR] . OVER =
107                     ORIF [CHAR] : OVER =
108                     ORIF [CHAR] ? OVER =
109                     ORIF [CHAR] ! OVER =
110                     THENS NIP
111           ;

113      : +CR      CR   0 col ! ;
114      : MORE     DUP col @ + LL > ?? +CR    col +! ;
115      : +TYPE    CAP @ IF
116                     OVER C@ to-upper 2 PICK C!
117                THEN

119                2DUP end-of-sentence CAP !

121                DUP MORE   TYPE          ( )
122      ;
123      : +SPACE   1 col +!   col @ LL < ?? SPACE ;

125 ( STEP V.  Parse a Line of Code. )

126           : sskip                    ( s n -- s+k n-k )
127                BEGIN
128                     DUP ANDIF OVER C@ is-visible NOT THEN
129                WHILE
130                     1 /STRING
131                REPEAT
132           ;

134           : sscan                    ( s n -- s+k n-k )
135                BEGIN
136                     DUP ANDIF OVER C@ is-visible THEN
137                WHILE
138                     1 /STRING
139                REPEAT
140           ;

142      : tokenize                      ( s n -- s'+k n'-k s' k )
143           sskip                           ( s' n')
144           2DUP sscan                      ( s' n' s'+k n'-k)
145           DUP >R 2SWAP R> -               ( s'+k n'-k s' k)
146      ;

148      VARIABLE  page-number

150      : page.line                     ( -- )
151           line-number @ 0 <#
152                [CHAR] ] HOLD   #S   [CHAR] . HOLD
153                2DROP page-number @ 0
154                #S   [CHAR] [ HOLD
155           #> DUP MORE TYPE +SPACE         ( )
156      ;
```

```
158 : decode-line                    ( letters . -- )
159       page.line
160       REFILL 0= ABORT" *** MISSING NUMBER-LINE *** "
161       BEGIN
162            PARSE-WORD                        ( letters . code .)
163            DUP
164       WHILE
165            CASE OVER C@ is-digit
166            IF   2>R tokenize OVER 1 2R> S+
167                 2SWAP 1 /STRING S+
168                 decode-word +TYPE +SPACE ( letters .)

170            ELSE S" S" 2OVER S=
171            IF   2DROP ( letters .)
172                 tokenize +TYPE +SPACE

174            ELSE S" -" 2OVER S= ORIF S" --" 2OVER S= THEN
175            IF   2DROP ( letters .)
176                 tokenize 2DROP
177                 col @ ?? CR +CR
178                 TRUE CAP !

180            ELSE S" .." 2OVER S= ORIF S" **" 2OVER S= THEN
181            IF   2DROP ( letters .)
182                 tokenize                  ( letters . token .)
183                 OVER C@ to-lower 2 PICK C!
184                 +TYPE +SPACE              ( letters .)

186            ELSE S" x" 2OVER S=
187            IF   2DROP ( letters .)
188                 tokenize +TYPE
189                 col @ ?? CR +CR
190                 TRUE CAP !

192            ELSE S" #" 2OVER S=
193            IF   2DROP ( letters .)
194                 tokenize +TYPE +SPACE
195                 TRUE CAP !
196            ELSE
197                 TRUE ABORT" *** ILLEGAL CODE *** "
198            THENS
199       REPEAT                         ( letters . token .)
200       2DROP 2DROP
201 ;

203 ( STEP VI.     Decode a File. )

205 : decoded                           ( filename . -- )
206       0 col !
207       TRUE CAP !
208       INPUT STREAM                          ( )
209            BEGIN REFILL WHILE
210                 SOURCE -TRAILING >PAD    ( s .)
211                 CASE DUP 0=
212                 IF   ( It's a blank line. )
213                      2DROP col @ ?? +CR
```

```
214                     TRUE CAP !

216                ELSE OVER C@ [CHAR] [ <>
217                IF   ( Doesn't begin with "[".
218                     ( Just copy it. )
219                     col @ ?? CR   TYPE +CR

221                ELSE OVER CHAR+ C@ [CHAR] P =
222                IF   ( Second character is "P".
223                     ( So it's a page-number. )

225                     2DROP PARSE-WORD ( "[P. ")
226                     2DROP PARSE-WORD ( "nn]")
227                     0. 2SWAP >NUMBER 2DROP ( nn 0)
228                     DROP page-number ! ( )

230                     0 line-number !

232                ELSE OVER CHAR+ C@ is-digit
233                IF   ( It's a line-number. )
234                     1 /STRING 0. 2SWAP >NUMBER     ( n . s .)
235                     1 /STRING 2SWAP                ( s . n .)
236                     DROP line-number !             ( s .)
237                     decode-line

239                ELSE
240                     TRUE ABORT" *** UNKNOWN LINE TYPE *** "
241                THENS
242           REPEAT
243      SOURCE-ID UNSTREAM CLOSED
244      col @ ?? CR
245 ;

247      CREATE filename     256 CHARS ALLOT
248      S" Work" filename PLACE

250      : CK filename COUNT decoded ;

252 \ CK

254 : RUN
255      S" topdown" <== OUT
256      S" overture" OUT << DECODED CLOSE OUT

258      S" bottomup" <== OUT
259      S" scenario" OUT << DECODED CLOSE OUT
260 ;

262 \ Procedamus in pace.    Wil Baden   Costa Mesa, California
```

A Report of the Board Meeting June 23–24, 1996

FIG Board Moves to Increase Member Benefits

Elizabeth D. Rather
Manhattan Beach, California

The Forth Interest Group Board of Directors met in conjunction with the Rochester Forth Conference held in Toronto, Canada, June 23 and 24.

Major actions include approval of further membership benefits, clarification of benefits for corporate members, review of *Forth Dimensions* advertising rates, planning of promotional activities, and review of plans for managing the FIG office and activities.

The new list of member benefits reflects the growing electronic capabilities of FIG, supplied through FIG President Skip Carter's Taygeta Internet site. The full list now includes:

- Six issues of *Forth Dimensions*
- Support of the annual FORML conference (proceedings are available at a discount for members who can't attend)
- 10% discount on FIG retail items (books, disks, back issues of *FD*, etc.)
- 10% discount for early registration for FORML (prior to November 1)

The new member benefits reflect the growing electronic capabilities of FIG

- Résumé referral service for programmers seeking jobs
- Contact with local Forth programmers through local chapters
- Electronic services:
  - Free personal web page (maximum size 100K)
  - Free e-mail forwarding service
  - Discounted domain registration ($25 for members and $50 for non-members, plus actual Internic registration charges)
  - Access to "members-only" site, with special interest mail groups and a growing list of other features
  - Résumés posted in the public areas of the site
  - Vast FTP software library, including the Forth Scientific Library and much more.

In addition, for only $125 per year, corporate memberships include:

- Five copies of each issue of *Forth Dimensions*, providing useful Forth information for the whole Forth programming team.
- Free corporate listing, with a 50-word description, in *Forth Dimensions* to increase corporate visibility in the Forth community and to aid in recruiting Forth programmers.
- 10% discount on advertising rates for advertising products and services as well as recruiting ads.
- A link from the FIG web site to a designated corporate web site, for better electronic access.
- All other regular member benefits.

The Board hopes individual members will suggest that their companies sign up as Corporate members. For example, in the many companies now building Forth teams developing Open Firmware drivers or Internet-related products, a corporate membership would provide an excellent, low-cost opportunity to help support their new Forth programmers and, at the same time, to increase company visibility in the Forth community to assist in recruiting.

For further information on the electronic services listed above, send e-mail to skip@forth.org. For information regarding the other benefits, call the FIG office.

Advertising rates were reduced substantially, in order to increase volume and to encourage new advertisers; however, the prior liberal discount schedules offered to some long-time advertisers will be replaced by a 10% discount offered to corporate members. Thus, existing advertisers should see little, if any, increase and new advertisers may be attracted. In addition, a new "ninth page" format was added. The new rates may be obtained from board member Jeff Fox (jfox@netcom.com).

A general review of procedures relating to subscription renewals produced several changes aimed at ensuring that members are adequately alerted to their renewal deadlines. Currently, advance notice of expiration is provided only on the address inserts in the *Forth Dimensions* envelope; in future, direct postcards will be sent. In addition, both the address inserts and renewal acknowledgments will feature a clip-out "membership card" to help members keep track of the member number, because it now provides valuable access to the "members-only" web pages and other membership benefits.

FIG increases visibility at the Embedded Systems Conference

FIG is planning to increase its visibility at the Embedded Systems Conference this Fall, with an improved booth offered by FORTH, Inc. and, if possible, sessions on Forth-related topics such as robotics, Open Firmware, and the new "Open Terminal Architecture" for smart card transaction terminals in Europe. The booth will be coordinated by Jeff Fox, with assistance from Mike Elola and Elizabeth Rather.

Other planned outreach strategies involve improved chapter coordination, providing brochures vendors can include in product shipments, and strategies for reaching the many new Forth programmers involved in the growing market for Open Firmware, Internet systems, and other Forth-related embedded systems.

The next Board of Directors meeting is scheduled for September 14. Members may reach board members with comments or suggestions via e-mail to figboard@forth.org or through the FIG office.

Chapter News
Southern Ontario FIG Chapter

In lieu of our normal quarterly meeting, the Southern Ontario FIG chapter put on a conference this June—the 1996 Rochester Forth Conference, to be exact. This is the first time the Forth Institute's annual conference has been held outside Rochester, New York, and is the first Forth conference ever held in Canada.

Chapter coordinator Nick Solntseff and member Brad Rodriguez made this proposal to the Institute's Larry Forsley at the 1995 Rochester conference. Nick, as Program Chair, solicited papers and tutorials, and will edit the conference proceedings. Brad, as Facilities Chair, arranged meeting and residence rooms. Ken McCracken suggested Ryerson Polytechnic University, in the heart of Toronto, as a venue, and made the initial contact with Ryerson. Elliott Chapin handled publicity, Rob McDonald provided Canadian customs information to the vendors, and Ken Kupisz assembled a visitor's guide (with assistance from Walter Elehew). J.D. Verne, Robin Ziolkowski, and Wendy Rodriguez provided extra help during the four-day conference.

Our two invited speakers were Mitch Bradley, speaking on new developments in Open Firmware; and Chuck Moore, on Forth hardware. Twenty-five papers were presented, including a series of papers on the Open Terminal Architecture being developed for European cash cards. For an extra "draw," we offered nine tutorials: HTML, HTTP and CGI, Java, Forth Hardware, Robotics, Forth under Windows, Metacompilation, and two on Open Firmware. Also, three members of the Ryerson Computer Science department were invited to participate; they expressed a fresh interest in Forth after the conference.

We have been invited to have future FIG chapter meetings at Ryerson. This is fortunate, since our chapter coordinator, Nick Solntseff, has just retired from McMaster University (our meeting place for the last several years). Meetings will continue to be held quarterly, on Saturdays.

—Brad Rodriguez

Report from "Rochester-in-Toronto"

Rochester Forth Conference

Nicholas Solntseff
Toronto, Ontario, Canada

Editor's note: Three major Forth conferences are held each year. FORML convenes every November at Asilomar, a woodland facility on the seashore near Monterey, California. euroForth gathers Forth practitioners to a different European city annually. The Rochester Forth Conference previously met in Rochester, New York; this year marks its first move from that city. These conferences are like masters classes, rich in content and interaction. Their proceedings are published and can be ordered from the Forth Interest Group. But much of the value of these conferences is obtained only in person, and we encourage you to attend.

The Rochester Forth Conference is alive and well! Not only has there been an upturn in attendance, while organizers of other conferences are bemoaning falling numbers, but the environment of a university in the midst of graduation ceremonies and a major city just one block away contributed to an exciting four days. The conference facilities at Ryerson Polytechnic University proved to be excellent, which is to be expected of an educational establishment with a School of Hospitality and Hotel Management. Ryerson runs a full-scale hotel where some of the conference attendees stayed. The student residences were also well-designed, with a private bathroom shared between two rooms. Air conditioning, although not really needed because of the generally cool and rainy weather this year, provided a welcome change from the slowly decaying Rochester University dormitories where I have stayed at previous conferences!

The conference ran very smoothly because of the Ryerson conference services and the work contributed by Brad Rodriguez and other members of the Southern Ontario FIG Chapter over the previous six months. As someone remarked, Larry and Brenda Forsley were seen for the first time ever at every meal time!

As for the technical side, I sensed that Forth has finally reached a stage in its development that clearly shows that the world needs Forth, if only because Mitch Bradley's Open Firmware, Sun's Java Development System, Bernard Hodson's software genes, and Europay (a major European cash-transaction consortium) all involve software that ultimately runs on a byte-code abstract or virtual machine to ensure platform independence. This common thread was recognized in the last session of the conference, and "Abstract Machines" was chosen as the theme for the 1997 Rochester Forth Conference (again to be held somewhere outside Rochester).

Mitch Bradley and Chuck Moore were invited speakers reporting on the latest development in their respective areas. The Europay work was presented by Elizabeth Rather, Jon Lee, Stephen Pelc, and Peter Johanes (Europay). Ten tutorials were offered on topics ranging from HTML, Java, Open Firmware, Forth hardware, and robotics. The last was given by Skip Carter, who brought a six-legged walking robot to Canada, not without a rather difficult passage through Customs and Immigration!

Forth's state of development clearly shows that the world needs Forth.

Working groups were put together as usual and, of the ones I could attend, I would like to highlight two—Portable Development Tools and Forth on Java. The former broke up after an hour with very little achieved except the realization that no one—practitioner or academic theoretician—has yet formulated the software-engineering principles required for this development; the latter emphasized the fact that Java is a close relative to Forth, and that there is room to implement a Forth compiler to Java byte-code instructions.

To summarize: Rochester-in-Toronto proved to be a worthwhile conference and I trust an even better conference will be held next year. I hope to see more of you next time!

Using Forth to manipulate the real world

Forthware

Digital Input and Synchronous I/O

Skip Carter
Monterey, California

Introduction

We now turn to the problems and issues involved with getting digital data *into* our application from the outside world. As before, we begin learning the principles by using the parallel port on the PC.

The PC Parallel Port Revisited

First, let's take another look at Table One (repeated from the first column, *FD* XVII/5).

Table One. The PC parallel port.

DB-25 Pin	Signal	Direction	Port	Bit
1	Strobe*	out	#Command	0
2	$Data_0$	out	#Data	0
3	$Data_1$	out	#Data	1
4	$Data_2$	out	#Data	2
5	$Data_3$	out	#Data	3
6	$Data_4$	out	#Data	4
7	$Data_5$	out	#Data	5
8	$Data_6$	out	#Data	6
9	$Data_7$	out	#Data	7
10	Ack*	in	#Status	6
11	Busy	in	#Status	7
12	Paper_out	in	#Status	5
13	Select_out	in	#Status	4
14	Auto_Feed*	out	#Command	1
15	Error*	in	#Status	3
16	Init*	out	#Command	2
17	Select_in*	out	#Command	3
18 to 25	Ground	NA	NA	NA

So far, we have only concerned ourselves with the #Data output port lines (and the Busy status input line). We will now use those other lines. We will presume the lowest common denominator type of port and that the data direction of the pins of #Data is fixed to output only. The five pins of the #Status port are usually fixed to input only. The four pins of the #Command port are *open collector*. An open collector line can be treated somewhat like a bus, with many devices (also open collector) that can potentially drive it. The state of an open collector line will be high only if none of the connected devices are driving it low (a pullup resistor, typically of a value like 2.2KΩ, between the line and 5 volts, should be used to assure that the line is well defined when not being pulled low). The open collector lines can be used as either input or output lines.

A Simple Example: Reading a Switch

Reading a simple switch is just a matter of reading the port (either #Status or #Command that has the switch attached to it), see Listing One. There are, of course, a few minor complications. First, as mentioned above, we should be sure to use pullup resistors on the open collector lines. Second, bit 7 of the #Status port is inverted in hardware so that it reads a zero when the line is high and reads a one when the line is low; bits 0, 2, and 3 of #Command are also inverted. Further, on an MS-DOS system, if your Forth reads the I/O ports via calls through the BIOS, bits 3 and 6 of #Status are also inverted. If you are using PFE V0.9.14 and Linux, you *might* have to make some minor patches to the source file support.c (this depends upon the version of the GNU C compiler you are using; I am using V2.7.0). If you can properly read the switch only the first time the "file" to the I/O ports is opened, and you get the same value for all subsequent reads, you will have to apply a diff patch *[see page 38]* to support.c and rebuild Forth.

An Elaborate Example: Synchronous Communication

We will now look at a significantly more complicated example. The cost of embedded systems is a sensitive function of the number of chips required to implement them. The chip count starts climbing rapidly when one accounts for the CPU, memory, the peripheral devices themselves, the chip-select logic, and all the miscellaneous glue logic that is necessary. Even if cost is not a major concern, a design involving fewer chips is likely to be more reliable than one with many chips. This is one reason why highly integrated chips are so popular for such systems. Highly integrated chips are rather expensive, so one approach that is often used is to design the peripheral chips so they can interface directly with the CPU, thus eliminating all the "glue." With this approach, the number of pins required to implement the interface becomes a consideration: the fewer the pins, the better. As a consequence of this, devices that use bit-synchronous serial communication with a controller have become common. Several microcontrollers provide support for a synchro-

Listing One. pport.fth

```
\ pport.fth              Parallel Port input for the PC

\ This is an ANS Forth program for manipulating the PC
\ parallel port under MSDOS or Linux requiring:
\         1. The File Access word set
\         2. the conditional compilation words in the PROGRAMMING-TOOLS word set
\         3. For use under MSDOS, the word
\             : MSDOS  ;
\            must be defined before loading this file
\         4. It is assumed that the inverse of COUNT is DROP 1-

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

\ $Author:   skip  $
\ $Workfile:   pport.fth  $
\ $Revision:   1.1  $
\ $Date:   13 Jul 1996 02:35:08  $

\ ================================================================

\ adapted from the Forth Scientific Library
\ assumes that the inverse of COUNT is DROP 1-
\ (this assumption could be avoided if C" was allowed to
\  be interpreted, but only the FILE S" is)

: DEFINED   ( c-addr u -- t/f )     \ returns definition status of
     DROP 1- FIND SWAP DROP         \ a word, true if it's there
;

\ ================================================================

S" MSDOS" DEFINED [IF]

S" fcontrol.seq" INCLUDED          \ from FD XVII/2


: initialize ( -- )     ; IMMEDIATE          \ nothing to do here
: close-port ( -- )     ; IMMEDIATE

#PORT       CONSTANT #DATA

[ELSE]            \ assume Unix/Linux

S" /usr/local/lib/forth/ports.fth" INCLUDED

: initialize ( -- )       \ open up the parallel I/O port
    init-port TO #IOPORTS
;

HEX

378       CONSTANT #DATA                   \ set as appropriate

[THEN]
```

(Listing One continues on next page.)

```
\ ================================================================

#DATA 1+  CONSTANT #STATUS
#DATA 2 + CONSTANT #COMMAND

DECIMAL

27 CONSTANT ESC                \ the escape character

: .binary ( n -- )           \ display in binary
    BASE @ SWAP
    2 BASE !
    8 U.R
    BASE !
;

: .hex ( n -- )           \ display in hex
    BASE @ SWAP
    HEX
    4 U.R
    BASE !
;


\ read specified port, repeating at each keystroke until ESC
: test_input ( n -- )

    initialize

    CR

    BEGIN
      DUP pc@ .binary CR
      KEY ESC =
    UNTIL

    DROP
    close-port
;

\ ================================================================

: test_status ( -- )

    CR ." Reading #STATUS port at " #STATUS .hex

    #STATUS test_input
;


: test_command ( -- )

    CR ." Reading #COMMAND port at " #COMMAND .hex

    #COMMAND test_input
;
```

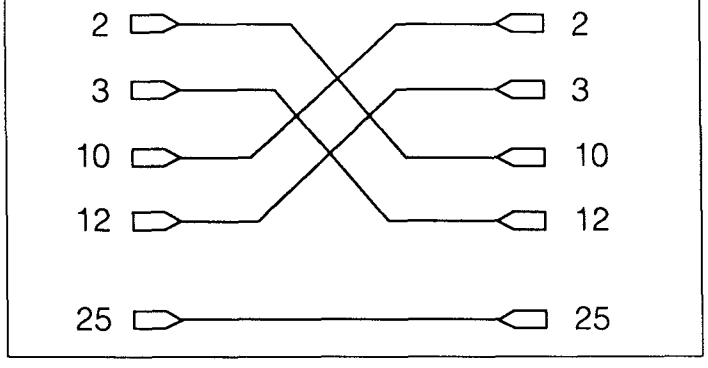
nous protocol, as do EEPROMS, and A/D and D/A chips. Several synchronous protocols are in common usage, I²C and SPI are probably the most common. We will look at how to implement SPI here.

Devices using SPI are classified as either master or slave devices. They use three wires for communication: MOSI (master out and slave in), MISO (master in and slave out), and a clock line. The MOSI line is the output line for the master device and is the line the slave device reads data in from. The MISO line is the output line for the slave device and is the input line for the master. The transfer of data onto the two lines is bitwise and occurs at previously agreed-upon phases of the clock signal. There can be multiple slaves, but there is only one master at a given time. The SPI master drives the clock and MOSI lines—these lines are used as inputs by the slave(s). The slave drives the MISO line. If there are multiple slaves, there must be some arrangement (hardware or software) to decide which slave is allowed to drive MISO. While the master is sending data out on MOSI, the selected slave is sending data out on MISO. So, at the end of a transfer, the master's input buffer matches the slave's output buffer, and the slave's input buffer is a copy of the master's output buffer. Depending upon the application, the master can be a fixed device, or the roles of master and slave can shift among the devices. Like RS-232, SPI is not really a standard, but more of a conceptual approach. There are variations in the clock polarity, the clock phases at transfer time, the bit order of a transfer, and the number of bits that constitute a single transfer. In our example, we will choose one common set of choices:

- The clock line will be low when idle.
- The master will write a bit on the rising clock edge.
- The slave will write a bit on the falling clock edge.
- A single transfer will be eight bits.
- The most significant bit is transferred first.
- The slave is selected for the entire duration of the transfer of all bytes.

It would be nice if we used the open collector lines for MOSI, MISO, and clock. That way, we could decide at runtime whether to be master or slave, and use the lines as either input or output, as appropriate. Unfortunately, there is a problem with this idea. A pullup resistor must be put on the line in order to assure that the line really goes up to 5 volts when it is not being driven low. The need for this resistor introduces a minor mechanical inconvenience: where do we physically locate these resistors for what, otherwise, would just be a simple cable? One place is inside the hood for the DB-25 connector on one end of the cable; one just has to provide for a 5 volt source inside this hood, too. A much more serious problem is that the pullup resistors limit the switching rate on the lines, which will result in very slow transmission rates.

Figure One. Cable for PC-to-PC SPI Communicaton.



As a consequence of these problems, we will avoid using the open collector lines for SPI. Instead, we will use the following: MOSI will be #Data bit 0, MISO will be #Status bit 6, the clock line will be #Data bit 1 for the master and be attached to #Status bit 5 on the slave side. If we make our cable have five wires (MOSI, MISO, two clock lines, and ground) as shown in Figure One, we can still choose which side is master at runtime (one clock line will always be idle); if we use slave select control, we need to add two more wires (this is not really necessary for PC-to-PC communication, and possibly not for some other devices either). Some simple devices can only be used as a master or only as a slave, so it's useful to be able to run in either mode. Many devices have SPI implemented in hardware and can, therefore, run extremely fast; since we are implementing SPI in high-level code, our system might not be capable of running so quickly. Since the data transfer rate in both directions is controlled by the master, it is desirable to run in master mode from the slowest device

Listing Two is an implementation of SPI, for master or slave mode, that will run either under MS-DOS or Linux. I put one-microsecond pauses after each clock edge; this is something you can experiment with. With these pauses, I can reliably transfer data between my 120 MHz Pentium and my 40 MHz '386, with either machine being the master. Without the pauses, I can only transfer with the '386 as the master. I have also used this code, in master mode, to communicate with the SPI port on the Motorola 68332. This, and many other Motorola microprocessors, contains a Queued Serial Module (QSM) which is, effectively, a serial I/O coprocessor. SPI using the QSM consists of setting up the configuration registers, filling the transmit buffer, and then letting the QSM run in the background. The CPU is free to do other things during the transmission, and it is notified when the transfer is complete via an interrupt or by polling a status register (depending upon the configuration settings). The QSM is very flexible in the range of configurations it can be put into. For less flexible devices, you may need to play with the timing, clock polarity, and phase. Some devices have different timing requirements for interbit timing, compared to interbyte timing. Some allow them to be selected for the entire duration of the transfer, while others require they be selected once for each byte (and deselected in between). A further variation to be aware of, is some devices have open collector SPI pins, while others do not.

Conclusion

This installment gets us into the basics of digital input. Next time, we will look at some further aspects of getting data into our code from the outside world. Please send your comments, suggestions and criticisms to me through *Forth Dimensions* or via e-mail at skip@taygeta.com.

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

diff patch for support.c — see page 34 for details of use

```
1190c1190,1192
<   return fseek (fid->f, pos, SEEK_SET) ? errno : 0;
---
>   /* return fseek (fid->f, pos, SEEK_SET) ? errno : 0; */
>   return lseek( fileno(fid->f), pos, SEEK_SET) == -1 ? errno : 0;
>
1207c1209,1211
<     fseek (fid->f, 0, SEEK_CUR); /* then seek to this position */
---
>     /* fseek (fid->f, 0, SEEK_CUR); */ /* then seek to this position */
>     lseek( fileno(fid->f), 0, SEEK_CUR);
>
1239c1243,1246
<   m = fread (p, 1, *n, fid->f)
 ---
>
>   /* m = fread (p, 1, *n, fid->f); */
>   m = read( fileno(fid->f), p, *n);
>
```

Listing Two. spi.fth

```
\ spi.fth          SPI using the PC Parallel Port

\ This is an ANS Forth program for SPI I/O on the PC
\ parallel port under MSDOS or Linux requiring:
\         1. The File Access word set
\         2. the conditional compilation words in the PROGRAMMING-TOOLS word set
\         3. The word USEC ( us -- )  is required to cause a delay
\            for the specified number of microseconds
\         4. For use under MSDOS the word
\             : MSDOS  ;
\            must be defined before loading this file
\         5. It is assumed that the inverse of COUNT is DROP 1-

\ Uses the following I/O lines of #DATA (base) and #STATUS (base + 1)
\
\     Master                    Slave
\  Pin      Bit     Name      Pin    Bit
\   10    Status-6  MISO       2     Data-0
\    3     Data-1   Clock     12    Status-5
\    2     Data-0   MOSI      10    Status-6
\    4     Data-2   Select    13    Status-4  ( active low )
\   25              Ground    25

\ For PC-PC communications the Select is not really necessary,
\ but for the Motorola QSM and other devices its needed

\               (c) Copyright 1996, Everett F. Carter Jr.
\                   Permission is granted by the author to use this software for
\                   any application provided this copyright notice is preserved.

\ $Author:   skip  $
\ $Workfile:   spi.fth  $
\ $Revision:   1.1  $
\ $Date:   13 Jul 1996 02:36:36  $

\ ==============================================================

\ adapted from the Forth Scientific Library
\ assumes that the inverse of COUNT is DROP 1-
```

```
: DEFINED   ( c-addr u -- t/f )     \ returns definition status of
      DROP 1- FIND SWAP DROP        \ a word, true if its there
;

\ ================================================================

S" MSDOS" DEFINED [IF]

S" fcontrol.seq" INCLUDED          \ from FD XVII/2

: initialize ( -- )     ; IMMEDIATE          \ nothing to do here
: close-port ( -- )     ; IMMEDIATE

#PORT       CONSTANT #DATA

[ELSE]            \ assume Unix/Linux

S" /usr/local/lib/forth/ports.fth" INCLUDED

: initialize ( -- )       \ open up the parallel I/O port
    init-port TO #IOPORTS
;

HEX

378       CONSTANT #DATA                    \ set as appropriate

[THEN]

\ ================================================================

#DATA 1+  CONSTANT #STATUS
#DATA 2 + CONSTANT #COMMAND

HEX

\ bitmasks and shift offsets

40 CONSTANT READ_MASK
 6 CONSTANT READ_SHIFT

 1 CONSTANT WRITE_MASK
 0 CONSTANT WRITE_SHIFT

10 CONSTANT SELECT_MASK

2          VALUE clock_mask
0          VALUE transfer-byte     \ execution vector
0          VALUE select            \ execution vector
0          VALUE deselect          \ execution vector

DECIMAL

16 CONSTANT BUFSIZE        \ the size of the I/O buffers

    BUFSIZE VALUE N

CREATE inbuf  BUFSIZE ALLOT
CREATE outbuf BUFSIZE ALLOT
```

(Listing Two continues on next page.)

```
: show-buffer ( addr n -- )

  CR
  0 DO I OVER + C@ .  LOOP
  DROP
  CR
;

\ generate additive sequence, for building dummy data
: gas  ( addr n -- )  0 DO I OVER  C! 1+ LOOP  DROP ;

: spi-setup ( -- )

    initialize           \ open/setup I/O port
    4 #DATA pc!          \ set idle levels, and select off
;

: strobe-clock ( -- )          \ drive clock line high then low
                               \ master does this
    #DATA pc@              \ get current value

    \ generate above value with clock high and clock low
    clock_mask OR DUP clock_mask XOR
    SWAP #DATA pc!

    ( pause momentarily here )
    1 usec

    #DATA pc!

    1 usec
;

: wait-on-clock-hi ( -- )      \ wait for leading clock edge
                               \ slave does this
    \ now wait for clock to go high
    BEGIN
      #STATUS pc@ clock_mask AND
    UNTIL
;

: wait-on-clock-lo ( -- )      \ wait for trailing clock edge
                               \ slave does this
    \ wait here for the clock to be low
    BEGIN
      #STATUS pc@ clock_mask AND 0=
    UNTIL
;
: read-bit ( -- x )         \ lsb of x is new bit

    #STATUS pc@ READ_MASK AND
    READ_SHIFT RSHIFT
;

: write-bit ( x -- )         \ write lsb of x

    WRITE_SHIFT LSHIFT
    WRITE_MASK AND           \ shift and mask to get just bit to send
```

```
    WRITE_MASK -1 XOR
    #DATA pc@ AND              \ set output bit to 0

    OR
    #DATA pc!                  \ send output bit
;

: master-transfer-byte ( x -- y )      \ write x, read y

    0

    0 7 DO
          1 LSHIFT
        OVER I RSHIFT write-bit    \ write before rising edge
        strobe-clock
        read-bit OR                \ read after falling edge
    -1 +LOOP

    SWAP DROP
;

: slave-transfer-byte ( x -- y )       \ write x, read y

    0

    0 7 DO
          1 LSHIFT
        wait-on-clock-hi       \ read after rising edge
          read-bit OR
        OVER I RSHIFT
        wait-on-clock-lo       \ write after falling edge
        write-bit

      -1 +LOOP

    SWAP DROP
;

: assert-select ( -- )    \ master
    0 #DATA pc!           \ set idle levels, and select ON
;

: deassert-select ( -- )
    4 #DATA pc!           \ set idle levels, and select off
;
: wait-on-select ( -- )   \ slave

    wait-on-clock-lo

    BEGIN
      #STATUS pc@ SELECT_MASK AND 0=
    UNTIL
;

: nothing ;
: transfer-data ( n -- )      \ send outbuf data, receive to inbuf

    select EXECUTE      \ select once for whole loop,
```

(Listing Two continues on next page.)

```
                        \ some devices prefer to select for each individual
                        \ transfer, move this inside the loop in that case
    0 DO
      outbuf I + C@
      transfer-byte EXECUTE
      inbuf I + C!
    LOOP

    deselect EXECUTE     \ see note on select above, move this
                         \ inside the loop too if select is moved
;

: setup ( -- )

    inbuf BUFSIZE 0 FILL
    outbuf  BUFSIZE gas
;

\ ================================================================

: test ( -- )

    setup
    BUFSIZE transfer-data
    close-port

    inbuf BUFSIZE show-buffer
;

: master ( -- )

    2 TO clock_mask

    ['] master-transfer-byte TO transfer-byte
    ['] assert-select        TO select
    ['] deassert-select      TO deselect

    spi-setup
;

: slave ( -- )

    32 TO clock_mask

    ['] slave-transfer-byte TO transfer-byte
    ['] nothing             TO deselect
    \ use 'nothing' below if NOT using
    \ the select line for the slave (e.g. PC to PC )
    ['] wait-on-select      TO select
    \ ['] nothing             TO select

    spi-setup
;

\ usage:   master test
\ or:      slave  test
```

Listing Three. usec timer for F-PC.

```
\ usec.seq                  Pause for a specified number of microseconds

code delay ( dus -- )       \ specified delay is a DOUBLE
     mov ax, # $8600
     pop cx
     pop dx
     int $15
     next
     end-code

: usec ( us -- )
    S>D delay
;
```

Listing Four. Recap of Linux parallel port access.

```
\ ports.fth          Forth Code to control parallel printer port
\                       see Ken Merk, Forth Dimensions July 1995
\  Uses IOPORTS device plus offset           EFC   March   1996
\  Converted for PFE under Linux             EFC   October 1995

\ This is an ANS Forth program requiring:
\         1. The File Access word set
\         2. The word FLUSH-FILE from the File Access Extensions word set

\  Note: in order to use this code
\        1. The device /dev/ioports should exist, it is a copy
\           of standard device /dev/port
\        2. The permissions on the /dev/ioports device should be:
\           crw-rw-rw-, and the group should be 'users' or
\           a locally defined group

\ $Author:   skip  $
\ $Workfile:   ports.fth  $
\ $Revision:   1.0  $
\ $Date:   11 Jul 1996 10:40:44  $

\ ==============================================================
: init-port ( -- n )
         S" /dev/ioports" R/W BIN OPEN-FILE
        ABORT" Unable to open I/O ports at /dev/ioports"
;

\ init-port VALUE #IOPORTS
-1 VALUE #IOPORTS
CREATE cbuf    8 ALLOT

: close-port ( -- )
        #IOPORTS CLOSE-FILE
        ABORT" Unable to close I/O ports at /dev/ioports"
;

: pc!  ( n port -- )
          S>D #IOPORTS REPOSITION-FILE THROW
          cbuf C! cbuf 1 #IOPORTS  WRITE-FILE DROP
        #IOPORTS FLUSH-FILE DROP
;

: pc@  ( port -- n )
          S>D #IOPORTS REPOSITION-FILE THROW
          cbuf 1 #IOPORTS READ-FILE 2DROP cbuf C@
;
```

Asilomar
FORML CONFERENCE

The original technical conference for professional Forth programmers and users.

18th annual FORML Forth Modification Laboratory Conference
Following Thanksgiving November 29—December 1, 1996

Asilomar Conference Center
Monterey Peninsula overlooking the Pacific Ocean
Pacific Grove, California, USA

Experimenting with the ANS Forth Standard

The ANS Forth standard has been out for two years, and the review process will start in another two years. FORML, with it's charter as Forth's "Modification Laboratory," is the appropriate place to let others know what your experiences have been as a developer or user while there's time for your ideas to spread.

Papers are sought that report on your experience writing ANS Forth programs and systems. That is, on your experiments. By calling attention to the successes and the problems now, before the review process begins, others will repeat your experiments, confirming or refuting your hypotheses.

Please, whether your ANS experiment was one line or a thousand, whether it succeeded or failed, or can be described in one page or ten, bring it to this year's FORML Conference to share with the world. As always, papers on any Forth-related topic are welcome.

Mail abstract(s) of approximately 100 words by October 1, 1996 to FORML, PO Box 2154, Oakland, CA 94621 or e-mail to FORML@ami.vip.best.com. Completed papers are due November 1, 1996.

John Rible, Conference Chairman **Robert Reiling, Conference Director**

Advance Registration Required • Call FIG Today 510-893-6784

Registration fee for conference attendees includes conference registration, coffee breaks, and notebook of papers submitted, and for everyone rooms Friday and Saturday, all meals including lunch Friday through lunch Sunday, wine and cheese parties Friday and Saturday nights, and use of Asilomar facilities.

Conference attendee in double room—$440 • Non-conference guest in same room—$320 • Children under 18 years old in same room—$190 • Infants under 2 years old in same room—free • Conference attendee in single room—$570

The Asilomar Conference Center combines excellent meeting and comfortable living accommodations with secluded forests on a Pacific Ocean beach. Early registration is recommended, space for this conference is limited.

Forth Interest Group members and their guests are eligible for a ten percent discount on registration fees.

Registration and membership information available by calling, fax or writing to:

Forth Interest Group, PO Box 2154, Oakland, CA 94621
voice 510-893-6784, fax 510-535-1295

Conference sponsored by the Forth Modification Laboratory, a Forth Interest Group activity.