DISK LIBRARY
Contributions from the Forth Community

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

FLOAT4th.BLK V1.4 Robert L. Smith C001 – $15
Software floating-point for fig-, poly-, 79-Std., 83-Std. Forths. IEEE short 32-bit, four standard functions, square root and log.
★★★ IBM, 190Kb, F83

Games in Forth C002 – $12
Misc. games, Go, TETRA, Life... Source.
★ IBM, 760Kb

A Forth Spreadsheet, Craig Lindley C003 – $12
This model spreadsheet first appeared in *Forth Dimensions* VII/1,2. Those issues contain docs & source.
★ IBM, 100Kb

Automatic Structure Charts, Kim Harris C004 – $15
Tools for analysis of large Forth programs, first presented at FORML conference. Full source; docs included in 1985 FORML Proceedings.
★★ IBM, 114Kb

A Simple Inference Engine, Martin Tracy C005 – $15
Based on engine in Winston & Horn's book on LISP, takes you from pattern variables to complete unification algorithm, with running commentary on Forth philosophy & style. Incl. source.
★★ IBM, 162 Kb

The Math Box, Nathaniel Grossman C006 – $15
Extended double-precision arithmetic, complete 32-bit fixed-point math & auto-ranging text. Incl. graphics. Utilities for rapid polynomial evaluation, continued fractions & Monte Carlo factorization. Incl. source & docs.
★★ IBM, 118 Kb

AstroForth & AstroOKO Demos, I.R. Agumirsian C007 – $12
AstroForth is the 83-Standard Russian version of Forth. Incl. window interface, full-screen editor, dynamic assembler & a great demo. AstroOKO, an astronavigation system in AstroForth, calculates sky position of several objects from different earth positions. Demos only.
★ IBM, 700 Kb

Forth List Handler, Martin Tracy C008 – $15
List primitives extend Forth to provide a flexible, high-speed environment for AI. Incl. ELISA and Winston & Horn's micro-LISP as examples. Incl. source & docs.
★★ IBM, 170 Kb

8051 Embedded Forth, William Payne C050 – $30
8051 ROMable Forth operating system. 8086-to-8051 target compiler. Incl. source. Docs are in the book *Embedded Controller Forth for the 8051 Family.* (Included with item #216.)
★★★ IBM HD, 4.3 Mb

68HC11 Collection C060 – $25
Collection of Forths, tools and floating-point routines for the 68HC11 controller.
★★★ IBM HD, 2.5 Mb

F83 V2.01, Mike Perry & Henry Laxen C100 – $30
Editor, assembler, decompiler, metacompiler. Source and shadow screens. Manual available separately (items #217, #235). Base for other F83 applications.
★ IBM, 83, 490 Kb

F-PC V3.6 & TCOM 2.5, Tom Zimmer C200 – $37
A full Forth system with pull-down menus, sequential files, editor, forward assembler, metacompiler, floating point. Complete source and help files. Manual for V3.5 available separately (items #350, #351). Base for other F-PC applications.
★ IBM HD, 83, 3.5Mb

F-PC TEACH V3.5, Lessons 0–7, Jack Brown C201 – $15
Forth classroom on disk. First seven lessons on learning Forth, from Jack Brown of B.C. Institute of Technology.
★ IBM HD, F-PC, 790 Kb

VP-Planner Float for F-PC, V1.01, Jack Brown C202 – $15
Software floating-point engine behind the VP-Planner spreadsheet. 80-bit (temporary-real) routines with transcendental functions, number I/O support, vectors to support numeric co-processor overlay & user NAN checking.
★★ IBM, F-PC, 350 Kb

F-PC Graphics V4.6, Mark Smiley C203 – $20
The latest versions of new graphics routines, including CGA, EGA, and VGA support, with numerous improvements over earlier versions created or supported by Mark Smiley.
★★ IBM HD, F-PC, 605 Kb

PocketForth V6.4, Chris Heilman C300 – $18
Smallest complete Forth for the Mac. Access to all Mac functions, events, files, graphics, floating point, macros, create standalone applications and DAs. Based on fig & *Starting Forth.* Incl. source and manual.
★ MAC, 640 Kb, System 7.01 Compatible.

Kevo V0.9b6, Antero Taivalsaari C360 – $20
Complete Forth-like object Forth for the Mac. Object-Prototype access to all Mac functions, files, graphics, floating point, macros, create standalone applications. Kernel source included, extensive demo files, manual.
★★★ MAC, 650 Kb, System 7.01 Compatible.

Yerkes Forth V3.67 C350 – $30
Complete object-oriented Forth for the Mac. Object access to all Mac functions, files, graphics, floating point, macros, create standalone applications. Incl. source, tutorial, assembler & manual.
★★ MAC, 2.4Mb, System 7.1 Compatible.

Pygmy V1.4, Frank Sergeant C500 – $30
A lean, fast Forth with full source code. Incl. full-screen editor, assembler and metacompiler. Up to 15 files open at a time.
★★ IBM, 320 Kb

KForth, Guy Kelly C600 – $30
A full Forth system with windows, mouse, drawing and modem packages. Incl. source & docs.
★★ IBM, 83, 2.5 Mb

Mops V2.6, Michael Hore C710 – $30
Close cousin to Yerkes and Neon. Very fast, compiles subroutine-threaded & native code. Object oriented. Uses F-P co-processor if present. Full access to Mac toolbox & system. Supports System 7 (e.g., AppleEvents). Incl. assembler, manual & source.
★★ MAC, 3 Mb, System 7.1 Compatible

BBL & Abundance, Roedy Green C800 – $37
BBL public-domain, 32-bit Forth with extensive support of DOS, meticulously optimized for execution speed. Abundance is a public-domain database language written in BBL. Incl. source & docs.
★★★ IBM HD, 13.8 Mb, hard disk required

Version-Replacement Policy

Return the old version with the FIG labels and get a new version replacement for 1/2 the current version price.

★ - Starting ★★ - Intermediate ★★★ - Advanced

MORE ON FORTH ENGINES

Volume 10 (January 1989) **810 – $25**
RTX reprints from 1988 Rochester Forth conference, object-oriented cmForth, lesser Forth engines. *87 pp.*

Volume 11 (July 1989) **811 – $25**
RTX supplement to *Footsteps in an Empty Valley*, SC32, 32-bit Forth engine, RTX interrupt utility. *93 pp.*

Volume 12 (April 1990) **812 – $25**
ShBoom chip architecture and instructions, neural computing module NCM3232, pigForth, binary radix sort on 80286, 68010, and RTX2000. *87 pp.*

Volume 13 (October 1990) **813 – $25**
PALs of the RTX2000 Mini-BEE, EBForth, AZForth, RTX-2101, 8086 eForth, 8051 eForth. *107 pp.*

Volume 14 **814 – $25**
RTX Pocket-Scope, eForth for muP20, ShBoom, eForth for CP/M & Z80, XMODEM for eForth. *116 pp.*

Volume 15 **815 – $25**
Moore: new CAD system for chip design, a portrait of the P20; Rible: QS1 Forth processor, QS2, RISCing it all; P20 eForth software simulator/debugger. *94 pp.*

Volume 16 **816 – $25**
OK-CAD System, MuP20, eForth system words, 386 eForth, 80386 protected mode operation, FRP 1600 – 16-bit real-time processor. *104 pp.*

Volume 17 **817 – $25**
P21 chip and specifications; PIC17C42; eForth for 68HC11, 8051, Transputer. *128 pp.*

Volume 18 **818 – $30**
MuP21 – programming, demos, eForth. *114 pp.*

Volume 19 **819 – $30**
More MuP21 – programming, demos, eForth. *135 pp.*

Volume 20 **820 – $30**
More MuP21 – programming, demos, F95, Forth Specific Language Microprocessor Patent 5,070,451. *126 pp.*

Volume 21 **821 – $30**
MuP21 Kit, My Troubles with This Darn 82C51, CT100 Lab Board, Born to Be Free, Laws of Computing, Traffic Controller and Zen of State Machines, ShBoom Microprocessor, Programmable Fieldbus Controller IX1, Logic Design of a 16-Bit Microprocessor P16. *98 pp.*

MISCELLANEOUS

T-shirt, "May the Forth Be With You" **601 – $24**
White design on dark blue shirt, or green design on tan shirt. Specify size—small, medium, large, x-large—on order form.

DR. DOBB'S JOURNAL back issues

Annual Forth issues, including code for Forth applications.

September 1982, September 1983, Sepember 1984 (3 issues) **425 – $25**

FORTH INTEREST GROUP

100 Dolores St., Suite 183 • Carmel, California 93923 • office@forth.org

For credit card orders or customer service:
Phone Orders ***831.37.FORTH***
weekdays *831.373.6784*
9.00 am – 1.30 pm PST ***831.373.2845*** *(fax)*

Name ______________________
Company ______________________
Street ______________________ voice ______________
City ______________________ fax ______________
State/Prov.____________ Zip________ e-mail ______________
Nation

Non-Post Office deliveries: include special instructions.	The amount of your sub-total...	the shipping & handling
Surface U.S. & International	Up to $40.00	$7.50
	$40.01 to $80.00	$10.00
	$80.01 to $150.00	$15.00
	Above $150.00	10% of Total
International Air		40% of Total
Courier Shipments		$15 + courier costs

PRICES MAY CHANGE WITHOUT NOTICE

Item	Title	Quantity	Unit Price	Total

☐ CHECK ENCLOSED *(payable to: Forth Interest Group)*
☐ VISA/MasterCard:

Card Number ______________________ exp. date ________

Signature ______________________

sub-total	
10% Member Discount Member#	
Sales tax* on sub-total *(California only)*	
Shipping and handling *(see chart above)*	
Membership* in the Forth Interest Group ☐ New ☐ Renewal	
TOTAL	

✖ MEMBERSHIP IN THE FORTH INTEREST GROUP

The Forth Interest Group (FIG) is a worldwide, non-profit, member-supported organization. Your membership includes a subscription to the bi-monthly magazine *Forth Dimensions*. FIG also offers its members an on-line database, a large selection of Forth literature, and other services. Cost is $45 per year for U.S.; all other countries $60 per year. This fee includes $39 for *Forth Dimensions*. No sales tax, handling fee, or discount on membership.

When you join, your first issue will arrive in four to six weeks; subsequent issues will be mailed to you every other month as they are published—six issues in all. Your membership entitles you to a 10% discount on publications and FIG functions. Dues are not deductible as a charitable contribution for U.S. federal income tax purposes, but may be deductible as a business expense.

PAYMENT MUST ACCOMPANY ALL ORDERS

PRICES: All orders must be pre-paid. Prices subject to change without notice. Credit card orders will be sent and billed at current prices. Checks must be in U.S. dollars, drawn on a U.S. bank. A $20 charge will be added for returned checks.

SHIPPING & HANDLING: All orders calculate shipping & handling based on order dollar value. *Special handling available on request.*

SHIPPING TIME: Books in stock are shipped within seven days of receipt of the order.
SURFACE DELIVERY:
U.S.: 10 days
other: 30–60 days

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

Fast service by fax: 831.373.2845

```
: SENDMSG ( object member-id -- )   OVER ?OBJECT
   OVER CELL- @ MEMBER? 0= THROW LATE-BINDING ;

: CALLING ( -- )
   BL WORD COUNT MEMBERS SEARCH-WORDLIST 0= ABORT" Member not defined"
   STATE @ IF
      POSTPONE LITERAL  POSTPONE SENDMSG
   ELSE  SENDMSG  THEN ; IMMEDIATE

{ -----------------------------------------------------------------
Each class has a namespace of members which belong to it. Members
exist as unique identifiers in a single wordlist. All are
immediate. All know their own XT, which is used as a unique
identifier for a method name.

DO-MEMBER is the execution behavior of a member. This is complicated
   by the need to re-cast the xt for non-class evaluation if
   it is a member, but not a member of THIS class.

CREATE-MEMBER makes a new member in THIS class' namespace.  The member
   knows its xt and name -- because these are kept in its body.

MEMBER is a defining word which a) returns the xt of an existing
   member or b) creates a new member and returns its xt.
----------------------------------------------------------------- }

: DO-NONMEMBER ( addr -- )
   -MEMBERS FIND DUP IF
      0< STATE @ 0<> AND IF COMPILE, ELSE EXECUTE THEN
      CSTATE @ IF +MEMBERS ELSE 0 >THIS  THEN   EXIT THEN
   DROP  COUNT TYPE  ."  not found" 0 >THIS  -1 THROW ;

: DO-MEMBER ( member-addr -- )
   @+ VISIBLE-MEMBER? IF
      NIP REFERENCE-MEMBER  EXIT THEN
   DO-NONMEMBER ;

: CREATE-MEMBER ( -- xt )
   >IN @ >R   CREATE-XT IMMEDIATE ( xt) DUP ,    R> >IN !
   BL WORD COUNT STRING,  DOES> DO-MEMBER ;

: MEMBER ( -- xt )
   >IN @  BL WORD COUNT MEMBERS SEARCH-WORDLIST IF  NIP EXIT  THEN  >IN !
   GET-CURRENT >R  MEMBERS SET-CURRENT
   ['] CREATE-MEMBER CATCH R> SET-CURRENT THROW ;

{ -----------------------------------------------------------------
Late binding behaviors

When members are explicitly referenced at runtime, these are the
routines that are called for the different type objects.  RUN-COLON
is used for both the colon and defer types.

RUN-DATA adds the offset in the member list data field to the object
   whose base address is on the stack.

RUN-OBJECT sets the current class according to the

----------------------------------------------------------------- }
```

```
: RUN-DATA ( object 'data -- addr )   @ +  0 >THIS ;

: RUN-OBJECT ( object 'data -- addr )   2@  SWAP >THIS  + ;

: RUN-COLON ( object 'data -- )
   SWAP >S THIS >C  @ EXECUTE  C> S> 0 >THIS ;

{ ----------------------------------------------------------------------
Early binding compilers

When members are referenced at compile time, code to execute a specfic
behavior is compiled.  Each of the different member types needs its
own early binding compiler.

Terminal methods, which are the final member name in a phrase, clear
the class namespace from the Forth search order.

Rreferences to an embedded objects are not terminal, but change the
active namespace to reflect the class which defined the object.

END-REFERENCE removes the class namespace from the Forth search order.

COMPILE-OBJECT compiles  "SELF" LIT +  and changes the namespace.

COMPILE-DATA compiles  "SELF" LIT +  .

COMPILE-COLON compiles  "SELF" >S "THIS" >C xt C> S>  .

COMPILE-DEFER compiles   "SELF" >S "THIS" >C member RESOLVED C> S>  .
-------------------------------------------------------------------- }

: END-REFERENCE ( -- )
   CSTATE @ DUP >THIS  ?EXIT -MEMBERS ;

: COMPILE-OBJECT ( 'data -- )   "SELF"
   2@ ?DUP IF POSTPONE LITERAL POSTPONE + THEN  >THIS +MEMBERS ;

: COMPILE-DATA ( 'data -- )   "SELF"   \ 'data: offset
   @ ?DUP IF POSTPONE LITERAL POSTPONE + THEN  END-REFERENCE ;

: PRE-COLON  ( -- )   "SELF"  POSTPONE >S  "THIS" POSTPONE >C ;
: POST-COLON ( -- )   POSTPONE C>  POSTPONE S>  END-REFERENCE ;

: COMPILE-COLON ( object 'data -- )   PRE-COLON
   @ COMPILE, POST-COLON ;

: COMPILE-DEFER ( object 'data -- )   PRE-COLON
   2 CELLS - @ POSTPONE LITERAL POSTPONE RESOLVED  POST-COLON ;

{ ----------------------------------------------------------------------
PRIVATE, PROTECTED, and PUBLIC set which kind of members follow.
Private words are bracketed by PRIVATE ... PUBLIC  or  PRIVATE ...
PROTECTED. Protected words are bracketed by PROTECTED ... PUBLIC  or
PROTECTED ... PRIVATE.

END-CLASS concludes a class definition by clearing CSTATE and restoring
   the search order as best it can.

BUFFER: reserves n bytes of data space in the current class.
DEFER: compiles a virtual member for the current class that has a
   default behavior. Used like a colon definition.  When a reference
   to the routine is made, it will late-bind in the current class
   or subclass for a more recently defined version (defined via :)
   and execute that if found. Otherwise, it will execute its default
   behavior.  The stack effect for all routines with the same name
   should be the same!

: defines a new executable member which
; terminates. Just like Forth!

BUILDS creates an embedded object of a specific class in the current
   class. When referenced, all methods of its class are available.

SUPER allows the reference of a parent's member.
COMMON allows access to a word in the underlying system that has
   been obscured by a class member.
------------------------------------------------------------------- }

GET-CURRENT ( *) CC-WORDS SET-CURRENT

   : PUBLIC    ( -- )   0 OPAQUE ! ;
   : PROTECTED ( -- )   1 OPAQUE ! ;
   : PRIVATE   ( -- )   2 OPAQUE ! ;

   : END-CLASS ( -- )   0 RE-OPEN  -CC ;

   : SUPER ( -- )
      THIS >SUPER @ >THIS POSTPONE SELF ; IMMEDIATE

   : COMMON  -CC  BL WORD DO-NONMEMBER  +CC ; IMMEDIATE

   : BUFFER: ( n -- )   MEMBER  THIS SIZEOF
      ['] RUN-DATA ['] COMPILE-DATA  NEW-MEMBER
      THIS >SIZE +! ;

   : VARIABLE ( -- )   THIS SIZEOF ALIGNED THIS >SIZE !
      [ +CC ] CELL BUFFER: [ -CC ] ;

   : CVARIABLE ( -- )
      [ +CC ] 1 BUFFER: [ -CC ] ;

   : DEFER: ( -- member runtime compiler colon-sys )
      OPAQUE @ 0 2 WITHIN 0= ABORT" Can't DEFER: in private"  MEMBER
      ['] RUN-COLON ['] COMPILE-DEFER  :NONAME ;

   : : ( -- member class-sys colon-sys )   MEMBER
      ['] RUN-COLON ['] COMPILE-COLON  :NONAME ;

   : ; ( member runtime compiler colon-sys -- )
      POSTPONE ; ROT ROT  NEW-MEMBER ; IMMEDIATE

   : BUILDS ( class -- )      MEMBER  THIS SIZEOF
      ['] RUN-OBJECT ['] COMPILE-OBJECT NEW-MEMBER
      ( class) DUP ,  SIZEOF THIS >SIZE +! ;

GET-CURRENT CC-WORDS <> THROW ( *) SET-CURRENT
```

Listing Two

```
{ ====================================================================
(C) Copyright 1999 FORTH, Inc.   www.forth.com
Examples of extensions
==================================================================== }

{ --------------------------------------------------------------------
OBJ-SIZE returns the size and base address of an object from its xt.

INDEXED[ ] generates an address from a base given a size and index.
>DATA[ ] returns the address of the nth object in the array at the xt.

BUILDS[ ] creates a named array of objects.  The structure of an indexed
   named object in memory is:
      | xt | class | data[ 0] | data[ 1] | ... | data[ n-1] |
-------------------------------------------------------------------- }

: OBJ-SIZE ( xt -- addr size )   >BODY CELL+ CELL+ @+ SIZEOF ;

: INDEXED[ ] ( n base size -- addr )   ROT * + ;

: >DATA[ ] ( n xt -- object )   OBJ-SIZE INDEXED[ ] ;

: BUILDS[ ] ( n class -- )
   CREATE-XT IMMEDIATE ( xt) , OBJTAG , ( class) DUP ,  SIZEOF  * /ALLOT
   DOES> [ '] >DATA[ ] (OBJECT) ;

{ --------------------------------------------------------------------
Between CLASS and END-CLASS, we want constants to simply return
their value when executed. For instance,
   CLASS FOO
      28 CONSTANT LC
      LC POINT BUILDS[ ] ARRAY
   END-CLASS
but when accesses interpretively outside class definition, it would
have to be used as
   FOO BUILDS SAM
   SAM LC
which means that an unnecessary object address is on the stack,
present simply to set the context for the named constant. sigh...
This behavior is target compilable because the tc will require a
twin of the constant anyway.

RUN-CONSTANT discards the object address and reads the constant
   from the member list entry.

COMPILE-CONSTANT compiles a forced drop of the required object
   address followed by the literal value of the constant.
-------------------------------------------------------------------- }

: RUN-CONSTANT ( object 'data -- n )
   NIP @ ;

: COMPILE-CONSTANT ( 'data -- )   "SELF"  POSTPONE DROP
   @  POSTPONE LITERAL  END-REFERENCE ;

{ --------------------------------------------------------------------
CREATE has similar problems to CONSTANT when used in a class.
RUN-CREATE discards the object address and skips over the
   unused data field in the member list entry.
COMPILE-CREATE compiles a drop, then compiles a convoluted
   reference to the memory following the member list entry.
   This is necessary, because we can't assume a constant
   address for the run-time system and must generate relocatable
   code.  The only things that are constant are: the distance from
   the body address of the class to the actual address of the
   data, and the handle (xt) of the class to which the data
   belongs.  So, the code, assuming the member entry address
   on the stack, is:
      [ THIS ] LITERAL >BODY [ THIS >BODY - CELL+ ] LITERAL +
------------------------------------------------------------------ }

: RUN-CREATE ( object 'data -- )
   NIP CELL+ ;

: COMPILE-CREATE ( 'data -- )   "SELF"  POSTPONE DROP
   THIS POSTPONE LITERAL  POSTPONE >BODY
   THIS >BODY - CELL+ POSTPONE LITERAL  POSTPONE +  END-REFERENCE ;

{ -----------------------------------------------------------------
RUN-OBJECT[] resembles RUN-OBJECT, but indexes an array of objects.

COMPILE-OBJECT[] compiles the literal offset to the start of the
   array, then a reference to the INDEXED[] routine for the class.
------------------------------------------------------------------ }

: RUN-OBJECT[] ( n object 'data -- addr )
   2@  ROT + ROT ROT  DUP >THIS  SIZEOF * + ;

: COMPILE-OBJECT[] ( 'data -- )   "SELF"
   2@  ?DUP IF POSTPONE LITERAL POSTPONE + THEN
   DUP >THIS  SIZEOF POSTPONE LITERAL POSTPONE INDEXED[] ;

{ -----------------------------------------------------------------
CONSTANT CREATE and BUILDS[] create new member list entries.

CONSTANT uses the data field for the constant value.

CREATE reserves but doesn't use the first cell of the data field; data
   following the CREATE will extend the data field of the entry.

BUILDS[] uses the first cell of the data field for the offset from the
   container's data space start to the array start, and the second
   cell of the data space to hold the class of the contained object.
------------------------------------------------------------------ }

GET-CURRENT ( *) CC-WORDS SET-CURRENT

   : CONSTANT ( n -- )
      MEMBER SWAP ['] RUN-CONSTANT ['] COMPILE-CONSTANT NEW-MEMBER ;

   : CREATE ( -- )
      MEMBER CELL ['] RUN-CREATE ['] COMPILE-CREATE NEW-MEMBER ;

   : BUILDS[] ( n class -- )   MEMBER  THIS SIZEOF
     ['] RUN-OBJECT[] ['] COMPILE-OBJECT[] NEW-MEMBER
      ( class) DUP ,  SIZEOF * THIS >SIZE +! ;

GET-CURRENT CC-WORDS <> THROW ( *) SET-CURRENT
```

Embedding 4tH bytecode

1. Introduction

The blending of Forth and C is a hot topic. Last FORML was even completely dedicated to it. In practice, it is not always that easy. The newest release of 4tH offers a new and easy way of doing just that, enabling the programmer to integrate Forth programs with C source easily, while the user never even knows he is actually using a Forth program!

2. Bytecode

Bytecode has been here for a long time. Even the UCSD Pascal compiler, which was quite popular in the seventies, used bytecode. If you have used Windows 3.x, the chances are you used to run bytecode programs every day, because earlier versions of Visual Basic created bytecode programs. Nowadays, bytecode is more popular than ever, because Java is based on that very same technology.

What is bytecode? Bytecode is machine code for a virtual processor. This virtual processor is usually created in software and interprets the bytecode. Of course, this slows down execution, but in real-life applications the performance is usually still acceptable.

A *virtual processor* (or virtual machine, as it is usually called) can be embedded in a program. Your browser probably contains one. Once you have loaded a Java applet from the World Wide Web, the virtual machine in your browser starts executing it.

A Visual Basic program may seem like a plain Windows executable, but all it does is start the appropriate VBRUNxxx.DLL, which contains the virtual machine that actually executes the bytecode of the program.

4tH [4tH was discussed in more depth in *Forth Dimensions* (XVIII.3) and *Forthwrite UK* (issue 96).] is not an ordinary Forth system. It is much more like an ordinary compiler, but instead of native code it creates bytecode. The bytecode 4tH creates looks like the code field of a single Forth word. In fact, 4tH has two bytecode formats: one in memory that can be executed, and one that can be saved to disk. The former is called *Hcode* and the latter is called *HX code*.

Of course, there is a reason for these two very different formats. Hcode was created for speed. It uses the native data-type formats of the processor, so interpretation is kept to a bare minimum. Since these native data-type formats differ from processor to processor, portability of this bytecode is restricted.

With 4tH version 3.0a, HX code was introduced. The HX code format is processor independent; it can be ported from processor to processor and from operating system to operating system without recompilation. Yes, you can compile a 4tH program on an RS/6000 running AIX, and run it under an Intel machine running Windows NT 4.0.

3. The 4tH library

The current 4tH distribution includes several programs which enable you to compile and run 4tH programs; but, in fact, these programs depend heavily on the 4tH library, which contains the actual compilation and execution functions. You can use the same functions in your own C programs if you link them with the 4tH library.

The 4tH library contains functions that load HX files, save HX files, run Hcode, decompile Hcode, or compile 4tH sources. A basic 4tH interpreter first calls the function that loads the HX file, and passes the resulting Hcode pointer to the execution function. However, the HX file is an external file which needs to be distributed along with the 4tH interpreter. This is not always what you want.

4. Embedding HX code

HX code was a good thing to start with, because it contains several safeguards against improper use and is highly portable. Furthermore, with the proper loading sequence, you could treat it just like an ordinary HX file.

The next thing to do was to create a program that converted the bytecode to C source. A first attempt to do that was done in late 1997. The bytecode was simply converted to unsigned chars, which were stored in a static string variable. This proved to be the proper approach.

Then a loading sequence had to be created. I already had one: the one that read an HX file from disk. All I had to do was modify it to read HX code from a string variable. Because only one static function in the loading function was dedicated to reading a byte from disk, I simply had to create another one that read a byte from a static string.

Finally, I had to put it all together. A C program was designed which could load and execute the bytecode. This would enable the user to create a standalone program that ran the embedded bytecode. This program was embedded into the conversion program, which could now either create a static string with the bytecode or a complete C source. Just in time to be included into the 4tH version 3.3a distribution!

5. Using embedded HX code

Making a standalone native executable is very easy, and it will work on all platforms to which 4tH has been ported, including MS-DOS, MS-Windows, and most Unixes. You can even do it without ever reading a book on C, because the HX2C conversion program takes care of that. HX2C is a command-line utility that takes two arguments: the name of the HX file and the name of the C source file it has to create. You can make a rule for the Unix *make* utility that compiles your 4tH source, converts the resulting HX file to C source, and compiles it to a native executable.

Hans Bezemer • hansoft@bigfoot.com
Den Haag • The Netherlands

"The Beez" has used Forth and C since the mid-1980s, and has authored several shareware programs and the freeware 4tH compiler. You can download 4tH from *ftp.taygeta* or from its own web site at *http://visitweb.com/hansoft.*

The C source created by HX2C is pretty straightforward (see Listing One). After all the red tape comes the bytecode. The `main()` function follows. First the HX code is loaded into memory by the `inst_4th()` function, which is virtually identical to the `load_4th()` function. It checks the integrity of the HX code and its compatibility with the linked virtual machine. If everything is all right, the virtual machine is invoked by calling `exec_4th()`. After the program has terminated, error messages (if any) are displayed, memory is freed by `free_4th()`, and the result is returned to the C program that called it.

But you can do a lot more with embedded HX code. You can embed several pieces of HX code and let them interact. However, this requires more inside knowledge of C. For this purpose, HX2C can generate only the static string containing the bytecode. Take a look at Listing Two. This code contains two pieces of bytecode. The first one adds two numbers, in this case 5 and 7. The result of this addition is stored in the variable `Result`. There is no need to keep the HX code in memory, so it can be freed by `free_4th()`. The second piece of code performs a division; in this case, it divides the contents of variable `Result` by 6. The result is stored in `Result` again, and can be displayed by the standard `printf()` function.

This proves you can seamlessly mix Forth and C with very little effort. There are no restrictions whatsoever to the use of the rest of the 4tH API, since `inst_4th()` returns an ordinary Hcode pointer. For instance, you can still use `load_4th()` to load additional HX-files.

6. The future

First of all, I want to use embedded code myself in a non-trivial program. Second, I'd like to experiment a bit further with this concept. It would be fun to duplicate the architecture of Visual Basic and convert the 4tH library to a DLL. In this version, the virtual machine and loading sequence has to be included in every executable, which is 10 KB overhead, at least. Finally, I'd like to see if the direct creation of Hcode in an executable has any advantages; it is quite volatile and requires special treatment, but doesn't have to be loaded or discarded.

What has this all to do with Forth? You haven't seen a single line of Forth code up to now. In my view, it has everything to do with Forth. Forth works best in niches other languages can't reach. Only the Forth concept makes it possible to embed small pieces of bytecode so easily and so efficiently. Forth works best in places where you can't see it. Bytecode is just another example.

Listing One

```
/*
** This file was generated by the HX to C converter
** Copyright 1997,9 by J.L. Bezemer
*/

#ifdef USRLIB4TH
#include <4th.h>
#include <sys/cmds_4th.h>
#else
#include "4th.h"
#include "cmds_4th.h"
#endif

#include <stdlib.h>

static bytecode EmbeddedHX [] = {
'\x00', '\x3a', '\x03', '\x00', '\x00', '\x00', '\x00', '\x00',
'\x00', '\x00', '\x00', '\xff', '\xff', '\xff', '\x7f', '\x01',
'\x00', '\x00', '\x00', '\x00', '\x00', '\x02', '\x00', '\x00',
'\x00', '\x00', '\x0d', '\x00', '\x00', '\x00', '\x00', '\x00',
'\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00',
'\x05', '\x00', '\x00', '\x00', '\x00', '\x00', '\x02', '\x48',
'\x65', '\x6c', '\x6c', '\x6f', '\x20', '\x77', '\x6f', '\x72',
'\x6c', '\x64', '\x21', '\x00', '\x91', '\x00'
};


#ifndef ARCHAIC
  int main (int argc, char **argv)
#else
  int main (argc, argv) int argc; char **argv;
```

```
#endif

{
  cell   Result;                         /* holds the result from the program */
  Hcode* Object;                         /* Hcode pointer */
                                         /* load the file */
  if ((Object = inst_4th (EmbeddedHX)) != NULL)
     {
       fflush (stderr);                  /* flush any messages */
                                         /* now execute it */
       Result = exec_4th (Object, argc, argv, 1, (cell) Version4th);

       fflush (stdout);

       if (Object->ErrNo)                /* show exit messages */
          fprintf (stderr, "Exiting; word %u: %s\n", Object->ErrLine,
                   errs_4th [Object->ErrNo]);
       else
          if (Result != CELL_MIN)
             fprintf (stderr, "Exiting; result: %ld\n", Result);

       Result = (Object->ErrNo ? EXIT_FAILURE : EXIT_SUCCESS);
       free_4th (Object);                /* discard the object */
       return ((int) Result);
     }
  return (EXIT_FAILURE);
}
```

Listing Two

```
#include <stdlib.h>
#include <stdio.h>
#include "4th.h"

static bytecode Addition [] = {
'\x00', '\x3a', '\x03', '\x00', '\x00', '\x00', '\x00', '\x00',
'\x00', '\x00', '\x00', '\xff', '\xff', '\xff', '\x7f', '\x01',
'\x00', '\x00', '\x00', '\x00', '\x00', '\x0b', '\x00', '\x00',
'\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00',
'\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00',
'\x3d', '\x00', '\x03', '\x00', '\x00', '\x00', '\x3d', '\x00',
'\x00', '\x00', '\x00', '\x00', '\x0b', '\x07', '\x3d', '\x00',
'\x03', '\x00', '\x00', '\x00', '\x3d', '\x00', '\x01', '\x00',
'\x00', '\x00', '\x0b', '\x07', '\x0b', '\x3d', '\x00', '\x02',
'\x00', '\x00', '\x00', '\x08', '\x8e', '\x00'
};

static bytecode Division [] = {
'\x00', '\x3a', '\x03', '\x00', '\x00', '\x00', '\x00', '\x00',
'\x00', '\x00', '\x00', '\xff', '\xff', '\xff', '\x7f', '\x01',
'\x00', '\x00', '\x00', '\x00', '\x00', '\x0b', '\x00', '\x00',
'\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00',
'\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00',
'\x3d', '\x00', '\x03', '\x00', '\x00', '\x00', '\x3d', '\x00',
'\x00', '\x00', '\x00', '\x00', '\x0b', '\x07', '\x3d', '\x00',
'\x03', '\x00', '\x00', '\x00', '\x3d', '\x00', '\x01', '\x00',
'\x00', '\x00', '\x0b', '\x07', '\x0e', '\x3d', '\x00', '\x02',
'\x00', '\x00', '\x00', '\x08', '\x8b', '\x00'
};

int main (int argc, char** argv)
{
Hcode*      Instance;
cell        Result;
                        /* load addition bytecode */
Instance = inst_4th (Addition);
                        /* execute: add 5 to 7 */
Result = exec_4th (Instance, 0, NULL, 2, 5, 7);
                        /* free instance */
free_4th (Instance);
                        /* load division bytecode */
Instance = inst_4th (Division);
                        /* execute: div Result by 6 */
Result = exec_4th (Instance, 0, NULL, 2, Result, 6);
                        /* free instance */
free_4th (Instance);
                        /* print Result and exit */
printf ("Result: %ld\n", (long) Result);
      return (EXIT_SUCCES);
}
```

Toolbelt #8, continued from page 28.

```
         THEN
         1-
      REPEAT DROP            ( widn ... wid1 n)
      SET-ORDER              ( )
   R> DROP ;

: +ORDER ( wid -- )
   DUP >R  -ORDER  GET-ORDER  R> SWAP 1+  SET-ORDER ;

{ ------------------------------------------------------------

CREATE-XT   WORDLIST:
====================

ANS provides no way for a created word to know its own xt.
This is needed for portability in this object package.
CREATE-XT provides a means to CREATE an entity in the
dictionary that can know its xt.

WORDLIST: <name>  provides named word lists.

------------------------------------------------------------- }

: CREATE-XT ( "name" -- xt )
   >IN @  CREATE  >IN !
   BL WORD COUNT  GET-CURRENT  SEARCH-WORDLIST 0= THROW ;

: WORDLIST:  ( "name" -- )  WORDLIST CONSTANT ;
```

F-PC Forth, v3.60

PIC Assembler

How about a single-pass assembler with no jump distance limits? Hopefully, that has been achieved here. Having an interest in the PIC17C44 microcontroller, I wrote a simple assembler using F-PC. The assembler didn't have any forward or backward label functions, as I didn't know how to do that yet. I proceeded to hand count the addresses for GOTO forward and backward jumps, and used a lot of NOPs between routines; not a very efficient method, to say the least.

The application was for electronic setting circles for amateur telescopes using optical sensors for right ascension and declination, and containing the Messier catalog and a bright star catalog. After getting that system working, I turned my attention to the label function problem.

Any articles I found on the subject were either too limited or too complicated to be of use. There must be an easy way to do this for a RISC microcontroller instruction set. It took me about a day and a half to figure out a method. I decided on using numbers instead of names (try making up 200 names). Numbers are used to index into a table for stored GOTO addresses and are used as *n* JF (jump forward) or *n* JB (jump backward).

When an *n* JF is encountered at a GOTO instruction, *n* address of JF-TBL is fetched and the first byte (count) is used as an offset to store the present GOTO instruction address. For any *n* from 0 to 255, each has a count byte and 16 two-byte addresses, so there can be 16 forward jumps of any one number to the same address. If more than 16 forward jumps are needed to the same address, another *n* JF is used

Figure One. Jump-forward routine.

```
CREATE JF-TBL 8448 ALLOT       \ Table for forward reference.
                               \ Room for 256 groups of
                               \ 16 two-byte addresses.
JF-TBL 8448 0 FILL

\ Get multiple forward jumps to same address.
: JF                ( n -- k ) \ Jump forward. k is GOTO data.
       DUP
       17 * JF-TBL +           \ Base address of n.
       DUP DUP C@              \ Count of n.
       DUP 2* 1+               \ Offset.
       ROT +                   \ JF-TBL + base + offset.
       WORDCOUNT @             \ Present program address.
       SWAP !                  \ Store to base n + offset.
       1+ SWAP !               \ Increment count,
                               \ leave n for GOTO.
```

Figure Two. Forward-jump routine.

```
\ C-BUF address is where assembled code is stored.
\ Store jump-to address to multiple GOTO's.
: FJ                ( n -- )   \ Forward jump.
       17 * JF-TBL +           \ Base address of n.
       DUP
       C@ 0                    \ Get count.
       ?DO DUP I 2* 1+ +       \ Calculate offset.
           @ 2*                \ Get GOTO addr 2* = C-BUF addr.
           C-BUF               \ Base addr;
                               \ start of EEPROM pgm "0".
           +                   \ Add GOTO address offset.
           WORDCOUNT @         \ Present FJ address.
           $2800 OR            \ Add GOTO instruction.
           SWAP !              \ Overwrite n JF GOTO.
       LOOP DROP ;
```

Figure Three. Backward-jump routine.

```
CREATE BK-TBL 512 ALLOT        \ Table for backward reference.
                               \ Room for 256 backward jumps.
BK-TBL 512 0 FILL
: BJ                ( n -- )   \ Backward jump.
       WORDCOUNT @             \ Present BJ address.
       BK-TBL ROT 2* + ! ;     \ Store address for later
                               \ use of GOTO.
```

Richard M. Mayer • rmayer123@aol.com
Waltham, Massachusetts

The author became interested in Forth in about 1978. He discovered F-PC Forth when it was fairly new and has stuck with it. An electrical engineer, he held R&D positions with LFE inc., ADCOLE, dbx and KINTEK before retiring in 1994.

and *n*(*a*) FJ *n*(*b*) FJ (forward jump) is placed at the designation address.

The jump forward routine is shown in Figure One.

Later in the assembly, an *n* FJ (forward jump) will be encountered. When this happens, *n* address of JF-TBL is fetched; the present address of the forward jump is added to the GOTO instruction and written to *n* JF of program memory, overwriting the old *n* JF GOTO. The forward jump routine is shown in Figure Two.

The backward jump is somewhat easier and takes less memory. All that is needed is to store the present address of *n* BJ (backward jump) to *n* BK-TBL. The backward jump routine is shown in Figure Three.

When *n* JB (jump backward) is encountered, all that is needed is to fetch *n* BK-TBL data for the GOTO instruction. There are no limits to the number of backward jumps of any one *n*. The jump backward routine is shown in Figure Four.

The result of all this is an assembler with no jump distance limits and which assembles in a single pass.

Figure Five is an example of a source code listing.

I am offering to the public domain the ASCII files 16ASM84.SEQ, 84EEPROM.SEQ, and 17ASM44.SEQ. 16ASM84.SEQ is the assembler, and contains error traps for forward and backward jumps and simple subroutine generation. 84EEPROM contains a written description of a schematic for a programmer board driven by the printer port of a PC. The files work together. 17ASM44.SEQ is similar to 16ASM84.SEQ.

Figure Four. Jump-backward routine.

```
: JB                  ( n -- k )  \ Jump backward. k is GOTO data.
     2* BK-TBL + @ ;              \ Get backward jump address
                                  \ for GOTO.
```

Figure Five. Example source code.

```
AUTOEDITOFF                           \ F-PC system word.
0 VALUE LO-TBL                        \ Assembler use.
0 VALUE HI-TBL                        \      "
0CH TMP-DATA PAST                     \ 16F84 use.
0DH TMP-DATA PRESENT                  \      "
1 JF                        GOTO      \ 0000H Reset vector
                            NOP
                            NOP
                            NOP
2 JF                        GOTO      \ 0004H Interrupt vector.
.
1 FJ      0 DD-PORT-A                 \ Port A outputs.
        C0H DD-PORT-B                 \ Port B bit 6-7 inputs;
                                      \ 0-5 outputs.
                            CLRW
        PAST                CLRF
        PRESENT             CLRF
          "                  "
3 JF                        GOTO      \ Jump to run program.

 PADDR SPLIT !> HI-TBL  !> LO-TBL           \ Save PCH and PCL
                                            \ of INDEX.
 PADDR CALL-SUB INDEX
        PCL                 MOVWF     \ Load computed offset.
         0                  RETLW     \ 0   0->5 bit 7 SEGMENT code.
         "                   "                   "
        18H                 RETLW     \ 9        "
```

STRETCHING STANDARD FORTH - #24

Linked List and Ordered List

The first two uses of object-oriented Forth for me were FILES and LINKED-LISTS. This article covers LINKED-LIST and ORDERED-LIST.

A *linked list* is a sequence of addresses with each address holding the next address, except the last address, which has 0.

The addresses are the *nodes* of the linked list.

The first node is the *head* of the list. From it, we can get to the rest of the nodes.

The nodes, other than the head, have information associated with them. This information, the *payload,* generally starts in the cell just above the node. We will follow that convention.

A payload has identifying material. In the programming here, the identifying material is at the start of the payload. This is the *item* of the node. For me, the item is usually a counted string. The contents of the string is the *name* of the node.

SWOOP is required for the class definitions. This will be part of SwiftForth.

If you don't already have SWOOP, then Tool Belt #7 and the SWOOP source in this issue have definitions that will be used here. In particular, it has **>LINK** to attach a new node to a list, and a definition of **STRING,** also.

To attach a new item to a list, we use **NEW-ITEM** defined with **>LINK** and **STRING,**.

The addresses held in the nodes of a list are offsets from the address of their containing node. This allows the memory containing the list to be saved and moved to a different location. This is necessary for saving and re-loading the system.

To look up an item in a list, start from the head and check each item for equality. **ITEM-SCAN=** does that. **OVER +** converts the relative address to an absolute address. **0** remains **0**.

In my work, I need or want the items to be in ordered sequence—"sorted." Therefore, I look for an item that is less than or equal to the one I have. When I find one, I compare it once more for equality, and return *previous* 0 or *previous item* . **ITEM-SCAN<=** does that.

If all items were referenced the same number of times, this would make the look-up twice as fast.

For simplicity, I generally keep lists of text items ordered.

```
{ ----------------------------------------------------------
The following words are in COMMON so they can be
used outside of a class.

ITEM-SCAN=  scans the items in an Linked List for an item
   that is equal.          ( str len head -- prev item|0 )

ITEM-SCAN<=  scans the items in an Ordered List for an item
   that is less or equal.  ( str len head -- prev item|0 )

NEW-ITEM  adds a new item to a list.
                                 ( str len prev -- item )

TOUPPER  is a Forth equivalent of the Standard C Library
   toupper.                               ( char -- flag )
---------------------------------------------------------- }

: ITEM-SCAN=       ( str len head -- prev item|0 )
   DUP 2SWAP 2>R               ( prev head) ( R: str len)
       BEGIN  NIP DUP @ DUP WHILE OVER +
              DUP CELL+ COUNT  2R@ COMPARE 0=
       UNTIL  CELL+
       THEN                    ( prev item|0)
   2R> 2DROP ( R: ) ;

: ITEM-SCAN<=      ( str len head -- prev item|0 )
   DUP 2SWAP 2>R               ( prev head) ( R: str len)
```

Wil Baden • wilbaden@netcom.com
Costa Mesa, California

Wil Baden, after many years of profane language, has retired to Standard Forth. For the source for this article, send e-mail requesting Stretching Forth #24: "Linked List and Ordered List."

```
      BEGIN  NIP DUP @ DUP WHILE OVER +
             DUP CELL+ COUNT  2R@ COMPARE 0< NOT
      UNTIL  CELL+ DUP COUNT  2R@ COMPARE 0= AND
      THEN                       ( prev item|0)
   2R> 2DROP ( R: ) ;

: NEW-ITEM        ( str len prev -- item )
    >LINK ( str len)
    HERE >R  STRING,  ALIGN  R> ( item) ;

: TOUPPER    ( char -- flag )
    DUP [ CHAR] a - 26 U< 32 AND XOR ;
```

LINKED-LIST

To make list handling easier, I define a class, and subclasses of, LINKED-LIST.

In the class **LINKED-LIST** there are protected definitions that will be used in the class and all its subclasses. These definitions will not be visible outside the class and its subclasses.

There will be a set of all VARIABLE and BUFFER: words for each object of the class.

```
CLASS LINKED-LIST

{ ---------------------------------------------------------
RESERVING  has the amount of data space to be alloted.

HEAD  is the head of the linked list.          ( -- addr )

NEXT  is the node pointer that points to the next node
   when listing.                               ( -- addr )

FIND  does an ordered scan through the list.  Deferred.
                                ( str len -- prev item|0 )
--------------------------------------------------------- }

PROTECTED

    VARIABLE RESERVING

    VARIABLE HEAD

    VARIABLE NEXT

    DEFER: FIND   ( str len -- prev item|0 )
        HEAD ITEM-SCAN<= ;
```

There are also public definitions that are visible inside and outside the class and its subclasses.

The deferred ones are virtual definitions and may get new definitions in subclasses.

When these are used outside a class they must be immediately preceded by a reference to an appropriate object.

```
{ ---------------------------------------------------------
INIT  re-initializes the list.  Deferred.

RESERVE  sets the amount of data space to be reserved when
   a new-item is added.  Default 0.        ( n [obj] -- )

.LINE  displays a line from the list.  Deferred.
                                        ( item [obj] -- )
REWIND  sets node pointer of the list to the beginning.
   Deferred.

READ  goes to the next item in the list.  Deferred.
                              ( [obj] -- false | item true )
---------------------------------------------------------- }

PUBLIC

    DEFER: INIT    ( -- )  0 HEAD ! ;

    : RESERVE    ( n -- )  0 MAX  RESERVING ! ;

    DEFER: .LINE  ( addr -- ) COUNT TYPE 3 SPACES ;

    DEFER: REWIND   HEAD NEXT ! ;

    DEFER: READ      ( -- false | item true )
        NEXT @
        DUP @ DUP IF +  DUP NEXT !  CELL+ DUP
        ELSE            NIP REWIND
        THEN ;
```

We have a definition for looking up an item in a list, and four definitions for adding an item to a list.

When deferred words in the definitions change in subclasses, these definitions do not change.

```
{ ----------------------------------------------------------
ITEM  looks up a string as an item in a list.
                                ( str len [obj] -- item|0 )

ADD-ITEM  adds an item to a list if it's not already
   there, and returns the address.
                                  ( str len [obj] -- item )

ADD  adds an item to a list if it's not already there.
                                       ( str len [obj] -- )

ADD-NEW-ITEM  adds an item to a list, and returns the
   address.                       ( str len [obj] -- item )

ADD-NEW  adds an item to a list.     ( str len [obj]  -- )

Adding an item to an ordered list inserts it in the proper
position to keep it ordered.
---------------------------------------------------------- }

    : ITEM   ( str len [obj] -- item|0 )  FIND NIP ;

    : ADD-ITEM   ( str len [obj] -- item )
        2DUP FIND  ( str len prev item)
        DUP IF  NIP NIP NIP
        ELSE    DROP ( str len prev) NEW-ITEM ( item)
```

```
                RESERVING @ /ALLOT
        THEN  ;

    : ADD   ( str len [obj] -- )  ADD-ITEM DROP ;

    : ADD-NEW-ITEM    ( str len [obj] -- item )
        2DUP FIND  ( str len prev item)
        DROP  ( str len prev) NEW-ITEM ( item)
        RESERVING @ /ALLOT ;

    : ADD-NEW    ( str len [obj] -- )  ADD-NEW-ITEM DROP ;
```

Very often, items are single words with no space characters. It is convenient to get them as the next input word.

```
{ -------------------------------------------------------
ITEM'  gets the next input word as an item and looks it
   up in the list.             ( [obj] "name" --- item|0 )

ADD-ITEM:  gets the next input word as an item, adds it
   to the list if it's not already there, and returns
   the item.                      ( [obj] "name" -- item )

ADD:  gets the next input word as an item, and adds it to
   to the list if it's not already there.
                                  ( [obj] "name" -- item )

ADD-NEW-ITEM:  gets the next input word as an item, adds it
   to the list, and returns the item.
                                  ( [obj] "name" -- item )

ADD-NEW:  gets the next input word as an item, and adds it
   to the list.                   ( [obj] "name" -- item )
-------------------------------------------------------- }

    : ITEM'  TOKEN >QPAD COUNT ITEM ;

    : ADD-ITEM:   TOKEN >QPAD COUNT ADD-ITEM ;

    : ADD:   TOKEN >QPAD COUNT ADD ;

    : ADD-NEW-ITEM:   TOKEN >QPAD COUNT ADD-NEW-ITEM ;

    : ADD-NEW:  TOKEN >QPAD COUNT ADD-NEW ;
```

We want some definitions to display the items in a list.

```
{ -------------------------------------------------------
ITEMS  displays the items in a list.

LIST  displays the items in a list using  .LINE  deferred
   word.                                      ( [obj] -- )

#ITEMS  counts the items in a list.         ( [obj] -- n )
-------------------------------------------------------- }
```

```
    : ITEMS
        CR  REWIND
        BEGIN  READ WHILE  COUNT ?TYPE 3 SPACES  REPEAT ;

    : LIST   CR  REWIND  BEGIN  READ WHILE  .LINE  REPEAT ;

    : #ITEMS   0  REWIND  BEGIN  READ WHILE  DROP 1+  REPEAT ;

END-CLASS
```

ORDERED-LIST

The class LINKED-LIST uses a single ordered list. When the list has many items, the performance will suffer.

A method that has worked well for me is to define a class of multiple ordered lists. This is a subclass of LINKED-LIST.

One approach I considered was 256 ordered sublists—one for each possible first character. That's a lot of lists for the items I use.

The items I deal with are usually English words or lines of text. I take the first character, convert lower case to upper case, translate characters below A to 0, A–Z to 1–26, and characters above Z to 27. This gives 28 sublists, each an ordered list. All the items in a sublist precede those in the next sublist.

```
LINKED-LIST SUBCLASS ORDERED-LIST

{ -------------------------------------  -------------------
|HEADS|  is the number of sublists.                ( -- n )

SUBHEADS  is the array of sublists.             ( -- addr )

HASH  hashes the items.         ( str l   -- str len hash )

FIND  looks up a string in the string'  sublist.
                                 ( str    -- prev item|0 )

MORE  gets the next item, passing from  e sublist to the
   next.                          ( --  lse | item true )

HEAD  has the index of the active subl  .      ( -- addr )
--------------------------------------  ------------------ }

PROTECTED

     28 CONSTANT |HEADS|

     |HEADS| CELLS BUFFER: SUBHEADS

     : HASH                      ( str len  - str len hash )
         OVER C@  TOUPPER  64 -  0 MAX    MIN ;

     : FIND                       ( str le   -- prev item|0 )
         HASH CELLS SUBHEADS + ITEM-SCA   ;

     : MORE                         ( -- f.  se | item true )
         NEXT @
         DUP @ DUP IF +  DUP NEXT !  CE   TRUE ( item true)
         ELSE  NIP
         THEN ;
```

The deferred words in LINKED-LIST get new definitions. The other words of LINKED-LIST are available in ORDERED-LIST using the new definitions of deferred words.

```
PUBLIC

    : INIT   ( [obj] -- )  SUBHEADS |HEADS| CELLS ERASE ;

    : .LINE   ( item [obj] -- )  COUNT TYPE CR ;

    : REWIND   ( [obj] -- )  0 HEAD !  SUBHEADS NEXT ! ;

    : READ   ( [obj] -- false | item true )
        BEGIN  MORE DUP ?? EXIT DROP             ( )
            HEAD @ 1+ DUP |HEADS| < WHILE         ( index)
            DUP HEAD !  CELLS SUBHEADS + NEXT !  ( )
        REPEAT                                    ( index)
        DROP 0  REWIND ;

END-CLASS
```

The following words are COMMON.

```
ITEM-SCAN<=   ITEM-SCAN=   NEW-ITEM
```

The following words are in class LINKED-LIST and ORDERED-LIST.

```
#ITEMS   .LINE   ADD   ADD-ITEM   ADD-ITEM:   ADD-NEW
ADD-NEW-ITEM   ADD-NEW-ITEM:   ADD-NEW:   ADD:   INIT
ITEM   ITEM'   ITEMS   LIST   READ   RESERVE   REWIND
```

Tool Belt #8

"EVALUATE Macros" has the following.

```
TOKEN   >QPAD   (:   ??
```

Ordered List Examples

Sorting a File

In this task, a file is read and its sorted lines are displayed.

The file is read by an object of class **INPUTFILES**. The opening of the file isn't shown here because there are many ways you might want to choose it.

The class **INPUTFILES** is re-opened to define **SORT** as a member.

The lines of the file are sorted by being added to an ordered list. For this, **LOANER** is an ordered list that is initialized, filled, displayed, and discarded.

As the list will consume data space, the sort process saves **HERE** before initializing the list.

LOANER ADD-NEW inserts the line in its proper position in the ordered list. **REPEAT** goes back to the BEGINning of the loop.

After the last line has been read and inserted in the list, **LOANER LIST** displays it. **LIST** uses deferred method **.LINE** to do the display. The default expands the item with **COUNT** and then does **TYPE CR**.

After the list is displayed, **HERE** is restored from the saved value. This makes the contents of **LOANER** garbage. This doesn't matter, because **LOANER** will be initialized at its next use.

To sort the lines of a file after opening **SRC** with it:
SRC SORT

```
INPUTFILES BUILDS SRC
ORDERED-LIST BUILDS LOANER

\ <file-obj> SORT  displays the sorted lines of the file.

INPUTFILES RE-OPEN

: SORT
    HERE >R
        LOANER INIT
        REWIND
        BEGIN  READ WHILE  LOANER Add-New  REPEAT
        LOANER LIST
    R> HERE - ALLOT ;

END-CLASS
```

Count Word Frequencies

Rather than sorting the lines of a file, we want to count the number of occurrences of the different words in it.

For this we define **Item&Quantity** as a subclass of ordered list. After the item of the list, we will make space for the quantity. In other applications, we may make space for further things. The definition of the cell for the quantity is deferred so different positions for it may be used in other lists. In the same way, **NAME** and **.NAME** are deferred.

In this new class, we replace the definition of **.LINE**. It will display the quantity when space has been allotted for it.

The definition of **LIST** is not changed.

The name of the object is **Word-Counts**.

```
ORDERED-LIST SUBCLASS Item&Quantity

    DEFER: QTY   ( item -- addr )  COUNT +  ALIGNED ;

    DEFER: NAME   ( item -- str len )  COUNT ;
```

Wil Baden • Costa Mesa, California
wilbaden@netcom.com

Wil Baden, after many years of profane language, has retired to Standard Forth. For the source for this article, send e-mail requesting Stretching Forth #25: Ordered List Examples.

```
   : .NAME    ( item -- )   NAME ?TYPE SPACE ;

   : .LINE    ( item -- )
        DUP .NAME
        RESERVING @ IF  DUP QTY @ .  THEN
        DROP
        2 SPACES ;

END-CLASS

Item&Quantity BUILDS Word-Counts   CELL Word-Counts RESERVE
```

We may want to count words in several files, so we do not initialize the list with each source file. The list can be displayed with **Word-Counts LIST** after collecting from any or all files.

When traversing a source file, instead of inserting each line, **Tally-Words-in-the-Line** is called for each line.

In **Tally-Words-in-the-Line**, the alphabetic words are extracted from each line. Each alphabetic word is inserted in **Word-Counts** with **ADD-ITEM**, which yields the address of the item. When a new item is inserted, the quantity has 0 as its value. The quantity is incremented with **++**.

Extracting the alphabetic words is done with **SCAN[**, **SKIP[**, **IS-ALPHA**, and **SPLIT**. Definitions for **SCAN[** and **SKIP[** are in Tool Belt #8.

IS-ALPHA tests a character for alphabetic. **SCAN[IS-ALPHA]SCAN** advances in the line up to the next alphabetic character.

We remember where we are in the line with **2DUP**, and use **SKIP[IS-ALPHA]SKIP** to advance to the next non-alphabetic character. **SPLIT** splits the line into two parts, the top part being the alphabetic word that has been isolated, the other part being the rest of the line.

Display the words with their counts using:
`.Word-Counts LIST`

```
: Tally-Words-in-the-Line                   ( str len -- )

    BEGIN  SCAN[  Is-Alpha ]SCAN  DUP WHILE           ( str len)

        2DUP SKIP[  Is-Alpha ]SKIP SPLIT  ( str2 len2 str1 len1)
        Word-Counts Add-Item  Word-Counts QTY ++     ( str len)

    REPEAT 2DROP ;

: Tally-Words                               ( -- )
    SRC REWIND
    BEGIN  SRC READ WHILE                      ( str len)
        Tally-Words-in-the-Line                ( )
    ?REPEAT ;
```

Here is the beginning of the word counts of DPANS94.

```
Tally-Words  Word-Counts LIST

a 1980   AAAA 1   abandoned 1   abbreviation 3
abbreviations 1   ABC 3   ABCD 2   abilities 1
ability 10   able 5   abort 49   Aborted 1   about 27
above 32   ABS 9   absence 6   absent 1   absolute 10
Abstain 1   abstract 1   abstraction 2   ABUFFER 1
Academic 2   accented 1   accept 29   acceptable 3
acceptance 3   accepted 5   accepting 1   accepts 4
Access 69   accessed 15   accesses 4   accessible 4
accessing 10   accommodate 2   accommodated 5
```

```
accomplish 3   accomplished 6   accomplishing 1
accordance 2   according 6   accordingly 1
Accredited 1   accuracy 6   accurate 2   accurately 2
accustomed 1   achieve 2   achieved 2   achievements 1
achieves 1   acknowledges 1   acknowledging 2
acknowledgment 1   ACM 1   acquire 3   acquired 3
acquiring 1   across 6   acted 1   acting 1   action 17
. . .
```

Most Frequent Words

The listing we started to display was too much of too little.

So instead of displaying all of the words alphabetically, let's show a few of them in order of greatest frequency.

An object, named **TOPPER**, to do that is given later. Now we'll use **!**, **ADD**, and **LIST** from it to define another method in **Item&Quantity**.

! seems to work like common **!**, but it does much more. For one thing, the size of the value is checked. It also initializes the working arrays.

ADD takes a pointer to a counted string and an associated quantity, and merges them into the working array. The maximum number of pairs is set by **!**.

LIST displays the saved pairs.

The new method in **Item&Quantity** is **TOP**.

```
Item&Quantity RE-OPEN

: TOP   ( n -- )
    TOPPER !
    REWIND  BEGIN  READ WHILE
        DUP QTY @ TOPPER ADD
    REPEAT
    TOPPER LIST ;

END-CLASS
```

Now we can see the 31 most-frequent words of our data.

```
31 WORD-COUNTS TOP

the 4857   of 2155   a 1980   to 1555   is 1398   and 1217
in 1058   Forth 740   Word 732   that 670   be 646
for 615   by 603   R 600   If 519   or 515   an 510
are 498   n 473   Stack 443   This 432   X 430   set 429
as 411   Core 406   U 400   Data 394   with 389   on 388
not 374   c 366
```

In this listing, the special words are buried in the very frequent general words. So let's eliminate the uninteresting words from the output.

In information retrieval, uninteresting words are called *stopwords* . We will put a bunch of them in an ordered list.

HOT is the name of the new method, and looks a lot like **TOP**.

```
ORDERED-LIST BUILDS STOPWORD

Item&Quantity RE-OPEN

: HOT   ( n -- )
    TOPPER !
    REWIND  BEGIN  READ WHILE
        DUP NAME STOPWORD ITEM 0= IF
            DUP DUP QTY @ TOPPER ADD
        THEN DROP
    REPEAT
    TOPPER LIST ;

END-CLASS
```

The set of important words looks a lot better.

```
31 Word-Counts HOT

Forth 740   Word 732   Stack 443   set 429   Core 406
Data 394   Standard 355   words 350   system 348
addr 334   file 333   Address 323   Floating 305
definition 303   implementation 289   Input 279
String 269   EXT 265   Name 263   cell 256
Character 245   defined 238   space 229   execution 224
characters 221   Program 217   Semantics 204   zero 198
Systems 197   Block 191   Code 188
```

Iterated interpretation is used to put words into the ordered list **STOPWORD**.

The following code is equivalent to:

```
STOPWORD ADD: a   STOPWORD ADD: about  STOPWORD ADD: above
STOPWORD ADD: across   STOPWORD ADD: after   . . .
```

```
{ ==========================================================
These stop words are taken from Christopher Fox,
"Lexical Analysis and Stoplists" in Frakes and
Baeza-Yates, _Information Retrieval_, ISBN 0-13-463837-9.
The list is based on the Brown Corpus.
========================================================== }

(: STOPWORD ADD: ||
    a   about   above   across   after   again   against
    all   almost   alone   along   already   also
    although   always   among   an   and   another   any
    anybody   anyone   anything   anywhere   are   area
    areas   around   as   ask   asked   asking   asks
    at   away   b   back   backed   backing   backs   be
    became   because   become   becomes   been   before
    began   behind   being   beings   best   better
    between   big   both   but   by   c   came   can
    cannot   case   cases   certain   certainly   clear
    clearly   come   could   d   did   differ   different
    differently   do   does   done   down   downed
    downing   downs   during   e   each   early   either
    end   ended   ending   ends   enough   even   evenly
    ever   every   everybody   everyone   everything
    everywhere   f   face   faces   fact   facts   far
    felt   few   find   finds   first   for   four   from
    full   fully   further   furthered   furthering
    furthers   g   gave   general   generally   get
    gets   give   given   gives   go   going   good
    goods   got   great   greater   greatest   group
```

```
   grouped   grouping   groups   h   had   has   have
   having   he   her   here   herself   high   higher
   highest   him   himself   his   how   however   i
   if   important   in   interest   interested
   interesting   interests   into   is   it   its
   itself   j   just   k   keep   keeps   kind   knew
   know   known   knows   l   large   largely   last
   later   latest   least   less   let   lets   like
   likely   long   longer   longest   m   made   make
   making   man   many   may   me   member   members
   men   might   more   most   mostly   mr   mrs   much
   must   my   myself   n   necessary   need   needed
   needing   needs   never   new   newer   newest   next
   no   nobody   non   noone   not   nothing   now
   nowhere   number   numbers   o   of   off   often
   old   older   oldest   on   once   one   only   open
   opened   opening   opens   or   order   ordered
   ordering   orders   other   others   our   out   over
   p   part   parted   parting   parts   per   perhaps
   place   places   point   pointed   pointing   points
   possible   present   presented   presenting
   presents   problem   problems   put   puts   q
   quite   r   rather   really   right   room   rooms
   s   said   same   saw   say   says   second   seconds
   see   seem   seemed   seeming   seems   sees
   several   shall   she   should   show   showed
   showing   shows   side   sides   since   small
   smaller   smallest   so   some   somebody   someone
   something   somewhere   state   states   still   such
   sure   t   take   taken   than   that   the   their
   them   then   there   therefore   these   they
   thing   things   think   thinks   this   those
   though   thought   thoughts   three   through   thus
   to   today   together   too   took   toward   turn
   turned   turning   turns   two   u   under   until
   up   upon   us   use   used   uses   v   very   w
   want   wanted   wanting   wants   was   way   ways
   we   well   wells   went   were   what   when   where
   whether   which   while   who   whole   whose   why
   will   with   within   without   work   worked
   working   works   would   x   y   year   years   yet
   you   young   younger   youngest your   yours   z
:)
```

Accumulate and Display Top Values

The class object here is not a linked list or ordered list. However, most of the operations have the same purpose and the same name. Subclasses can re-define **.LINE** to get other forms for listings.

```
{ ------------------------------------------------------------
CLASS TOPPERS -- Accumulate and Display Top Values

<object> INIT  clears accumulators for top values.
<n> <object> !  sets number of top values to <n>, and
    initializes.

<addr> <quantity> <object> ADD  inserts into top values.
    <addr> is a counted string.

<object> |TOPPER|  is the maximum number of toppers.

<object> @  is the current number of toppers.

<object> REWIND  sets the list to the beginning.

<object> READ  gets the next ( _addr qty_).

<object> ITEMS  displays the strings.

<object> .LINE  displays the string at ( _addr_) and
( _qty_).  Deferred.

<object> LIST  uses .LINE for every ( _addr qty_).

---------------------------------------------------------- }

CLASS TOPPERS

150 CONSTANT |TOPPER|                ( Allow for so many. )

PROTECTED

VARIABLE N

VARIABLE NEXT

|TOPPER| 1+ CELLS BUFFER: ADDR      ( Ptr to Counted String )
|TOPPER| 1+ CELLS BUFFER: RANK      ( Quantity for Rank )

PUBLIC

: @   ( -- n )  N COMMON @ ?DUP 0= ?? |TOPPER| ;

: INIT                                           ( -- )
    N COMMON @ 0= IF  |TOPPER| N COMMON !  THEN
    ADDR N COMMON @ CELLS ERASE
    RANK N COMMON @ CELLS ERASE ;

: !                                         ( n -- )
    DUP 1- |TOPPER| U< NOT ABORT" Illegal Number of Toppers."
    N COMMON !                                         ( )
    INIT ;

: ADD                                ( item quantity -- )
    0 N COMMON @ 1- CELLS DO                ( item quantity)
        DUP  RANK I + COMMON @ > NOT ?? LEAVE
        DUP  RANK I + DUP COMMON @ OVER CELL+ COMMON ! COMMON !
        OVER ADDR I + DUP COMMON @ OVER CELL+ COMMON ! COMMON !
    -1 CELLS +LOOP 2DROP ;

: REWIND   NEXT OFF ;

: READ      ( -- false | addr qty true )
          NEXT COMMON @ DUP N COMMON @ < IF
               CELLS DUP ADDR + COMMON @
               SWAP RANK + COMMON @  TRUE
               NEXT ++
          ELSE
               DROP  NEXT OFF  FALSE
          THEN ;

: ITEMS    CR  REWIND
          BEGIN  READ  WHILE
                 DROP  COUNT ?TYPE  3 SPACES
          REPEAT
          CR ;

DEFER: .LINE     ( addr qty -- )
          SWAP COUNT ?TYPE SPACE . 2 SPACES ;

: LIST   CR REWIND  BEGIN  READ WHILE  .LINE  ?REPEAT  CR ;

: ?   ( -- )  @ . ;

: OFF  0 ! ;

: ON   -1 ! ;

: +!   @ 1+ ! ;

: ++  1 +! ;

END-CLASS

TOPPERS BUILDS TOPPER
```

Part Two

Reed Solomon Error Correction

In part one of this article (*FD* XX.4), an introduction to finite field arithmetic was given. This is the math system used in many computer algorithms such as error correction, data encryption, and pseudo-random number generation. It is used because it is efficiently implemented in both computer hardware and software. In this article, Reed Solomon Error Correction design and use is discussed.

Reed Solomon Design

To design a Reed Solomon corrector, various parameters such as symbol size, generator polynomial, and number of redundancy symbols must be chosen. A thumbnail sketch of how to do this is presented here.

Reed Solomon Error Correction Codes (ECC) always work on a block of data, usually of a fixed size determined by the designer. If you have a continuous data stream to protect, it must be divided into blocks, or chunks of fixed size. Sometimes, the block size to choose is obvious: if you are sending data in 128-byte packets, for example, that would be a natural choice for the block size. The protected data is not changed by the ECC. ECC symbols are simply added to the end of the data block and the new, larger block is sent as a single block of data. The maximum block size that can be protected is limited by the symbol size chosen, as follows:

Maximum block size = $2^{(\text{symbol\#bits})}$– error correction symbols added – 2

For example, if we choose an eight-bit symbol (byte), and decide to protect the data block with 16 bytes of ECC, the maximum data block we could handle would be 256 – 16 – 2, or 238 bytes. Any smaller block size will also work. The data symbols must be the same size or smaller than the ECC symbols, so eight-bit symbols are often chosen if the data block is byte-oriented. *Interleaving* techniques, discussed later, can be used if larger block sizes are needed.

The number of correction symbols to add to each data block depends on the correction capability desired. It takes two symbols of redundancy (*ECC symbols*, or *check symbols*) to correct any data symbol in error. You can think of it as one symbol to figure out where the error is, and one to figure out what the error needs to be corrected to. If we only needed to correct a single byte in error, we would append two bytes to a byte-oriented datablock. It is usual practice to use at least six to eight or more symbols of protection, and more symbols may be needed to give good error detection performance, as discussed later.

In the example above, sixteen bytes of ECC have been appended to the data block. The ECC can therefore correct up to eight bytes in error. The bytes in error can be anywhere in the received block, including the ECC bytes themselves. However, the nature of Reed-Solomon ECC is such that, if the received data block has more errors than can be corrected, no errors at all can be corrected. So if, in the example above, nine bytes are in error, the correction algorithm will fail, and not a single byte can be repaired.

Furthermore, if all the check symbols are used to correct errors, we cannot tell with confidence whether or not the capability of the ECC was overrun. This can result in a miscorrection in which the ECC seemed to correct all errors but, in reality, there were still more errors and we didn't know it. We can reduce the probability of miscorrection either by giving up some error correction capability, by adding more check symbols, or by further protecting the data with an error detector such as CRC (cyclic redundancy check). Notes in the ECC software package discuss this further.

The choice of which of the available irreducible polynomials and which offsets to use is subtle and not important in most applications. Some polynomials and offsets allow hardware savings if parts of the algorithms are to be implemented in hardware. I don't recommend one over another, as all valid polynomials will give the same correction capability and the same probability of miscorrection over the full set of data patterns.

An *encoder* (Figure One-a) is used to generate the ECC symbols on the sending end. These symbols are then sent along with the original data. A *decoder* (Figure One-b) is used at the receiving end to detect received errors, and to provide the information necessary to correct errors if any are present. The encoder is implemented as a polynomial divider that generates check symbols. When the check symbols are appended to data symbols, the entire data set is evenly divisible by a polynomial called the *generator polynomial*.

Because the number of data sets that are evenly divisible by the generator polynomial is miniscule compared to all possible data sets, nearly any error combination will corrupt the data set to something not evenly divisible by the generator polynomial. We can, therefore, detect the presence of an error in the received data set by testing to see if a remainder exists when the data set is divided by the generator polynomial. This is what the decoder does.

If errors are detected, the decoder block's registers contain *syndrome* symbols. These symbols contain the information necessary to locate and correct the errors. For each error in the data set, two parameters are required: the location of the error (given as the number of symbols from the end of the data set) and the error value, which is given as the symbol to add (XOR) to the received symbol to make it correct. The algorithms to find these parameters involve solutions to simultaneous equations.

The more errors there are in the received data set, the larger

Glenn Dixon • Roy, Utah
Dixong@iomega.com

Glenn Dixon works at Iomega Corporation. He served as channel engineer for the Zip 100 and Zip 250, and uses Forth for anything the company will allow. When not Forthing, he writes murder mysteries.

the set of simultaneous equations that must be solved. If there was only one error, the correction chore is trivial. It is not too bad when only two errors have been detected. For more errors, the task of correction gets exponentially harder. Furthermore, in most situations, single-symbol errors are far more common than multi-symbol errors. For this reason, single, double, and sometimes triple and quadruple error correctors are encoded directly and tried first. If they fail, a general corrector that is much slower, but that can correct any number of errors up to the code's capability, is invoked. This gives better correction times.

The general error corrector is not very efficient, but does have some optimizations. For example, instead of attempting to solve a large set of simultaneous equations to find the error locations, it uses a *Chien search*, which is simply a loop that tests every symbol location, one-by-one, to see if there was an error at that spot. To find error values, the Belekamp-Massey algorithm is used, and is explained in the references.

Suppose you wish to protect a byte-oriented block that is 512 bytes long. If you use byte-sized symbols, you cannot cover a 512-byte span. You must, therefore, break the block into smaller blocks. In many systems, errors tend to come in bursts, so that multiple-symbol errors are adjacent, or at least close to each other. If this is the case, a set of smaller data blocks, each with its own protection, are often interleaved. Symbols are sent in a round-robin fashion, one from each data set in turn. This way, if a burst error occurs, the multiple data sets share the burden of correction. For example, we might divide the 512-byte data set into three smaller data sets of roughly 171 bytes each, then interleave them. This solves the block size problem and also aids correction efficiency. If a six-byte error burst occurs, it will distribute evenly across the three data sets, and it is more efficient to correct two bytes in three data sets than it is to correct six bytes in

Figure One-a. Encoder

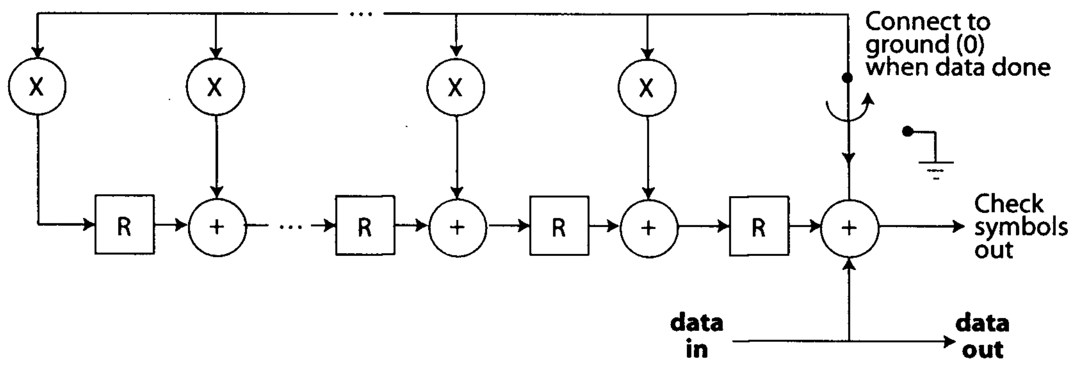
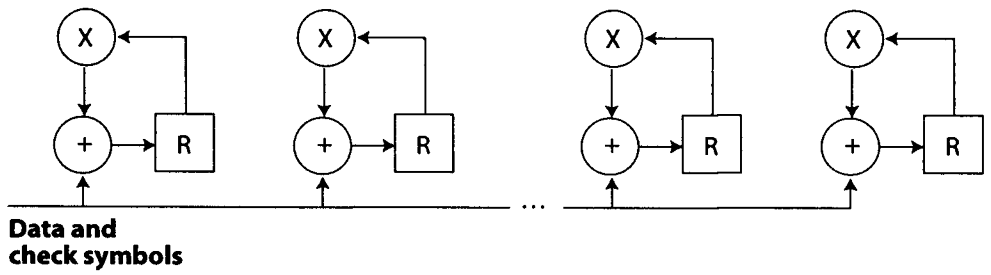
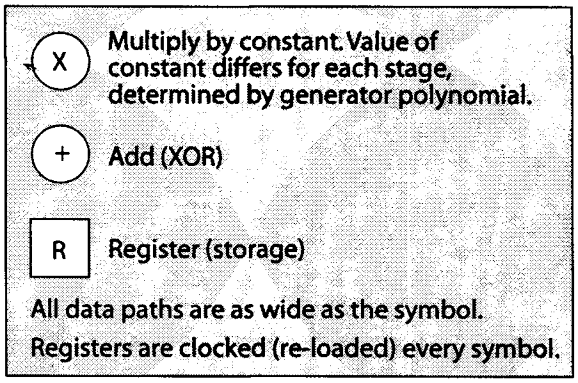


Figure One-b. Decoder





Multiply by constant. Value of constant differs for each stage, determined by generator polynomial.

Add (XOR)

Register (storage)

All data paths are as wide as the symbol.
Registers are clocked (re-loaded) every symbol.

one data set.

The code package that implements Reed-Solomon in Forth is too large to be included with this article. It is available for downloading at ftp://ftp.forth.org/pub/Forth/FD/1999/ReedSol.zip. The zipped file contains five files:

rsencode.txt	The finite field generator and the encoder. Also verifies the finite field.
rsdecode.txt	The decoder, and single- and double- error correctors. Some disk utilities for reading and writing data sets are also included.
rscorrec.txt	The full error corrector.
rsverify.txt	A simple code verifier that ensures things are basically working.
rsload.txt	Loads the above files.

Some words of note

ENCODE takes a symbol on the stack and encodes it. Results are in the array called REGISTERS. DECODE takes a symbol from the stack and decodes it. Results are in the SYNDROMES array.

RESET-ENCODER and RESET-DECODER must be invoked before starting a new data set.

DATABUF is an array you can use for the data set you're working on.

ERROR? returns a true flag if the syndrome array indicates an error has occurred. You must decode the entire data set, including the ECC check symbols, before the syndrome array has valid data.

CORRECT implements the correction algorithm. It checks for an error, then attempts single and double error correction, in order. If that fails, it invokes the full correction algorithm. It returns a true flag if correction was done successfully. The data in DATABUF is changed to a corrected version by the CORRECT algorithm. Note that if ECC is overrun, CORRECT may return a true flag when the data has actually been miscorrected. As noted above, you can compensate for this with CRC protection, or by limiting the number of errors the full corrector is allowed to correct. If, for example, the ECC has the capability to correct eight symbol errors (16 check symbols), and you limit the corrector to error lengths of four or less and fail any data set with five or more errors, the miscorrection probability will be very low.

In this implementation, all symbols are stored as CELLs, and the check symbols are always stored separately from the data, in the REGISTERS array. In practice, the check symbols are concatenated to the data, and then sent. This implementation does not have support for interleaving multiple data sets.

As a final comment, Forth's interactivity was very valuable in helping me understand this difficult subject. The ability to do finite field arithmetic from the keyboard as if it were a calculator, and to simply play with the algorithms, was very nice.

References

1. *Theory and Practice of Error Control Codes*. Richard Blayhut. ISBN 0-201-10102-5 (1983)
2. *Practical Error Correction Design for Engineers*. 2nd ed. Neal Glover and Trent Dudley. Available from Cirrus Logic (303-466-5228).
3. "Error Recovery Codes," *Dr. Dobbs Journal* (Dec 1994). Bart de Cann. Good summary of ECC.

Three-Stack Machine, continued from page 14.

in the world. I suspect that a viable H3sm engine would be somewhere in the vicinity of twice the silicon of a similar Forth or p21-type machine. Maybe more, but not ten times more. Most of the horridness of H3sm-in-C should vanish as hardware. While coding the interpreter in macros, I got the sense of the thickness of the language. I had hoped stack-dancing would all but vanish. It didn't, not nearly; but with three legs, you kick twice as many shins with the same pirouettes.

Parts of the H3sm interpreter are reusable in ways that may not be the case in a Forth, because things could be done with just pointers. Pointers seem almost inviting at times in H3sm. That surprised me. They were certainly a nightmare in C in the H3sm primitives, such as doHNC, H3sm's EXECUTE. doHNC is where everything that is out of phase with C comes to a head. Pointers are still sort of an unsolved problem in programming, it seems, and remain so. H3sm hides pointers somewhat, though, without Java-like silliness like pretending there's no address bus.

Also of note, stack manipulation challenges are perhaps at their worst when writing an interpreter, because the interpreter must be immune to any possible stack or Size effects of EXECUTE (doHNC). That's one of the things you learn when writing a TIL. Hairy stuff. The H3sm user shouldn't have to resort to elaborate stack notations like the ones in the interpreter source, which I found quite necessary at the time because that wasn't H3sm yet. Similarly, I hope to have done the ugliest pointer abuse for you already.

Proper factoring can allow you to make maintainable C programs that violate every written and unwritten rule of C programming. Factoring is the Light and the Way. "Structured Programming" is a historically unfortunate obfuscation of factoring.

The "operator-typing" of Forth words like 2+ and so on tends to persist in threads using these operators. I see the elimination of size-typed operators like 2DUP and so on as a win, and the win persists. The attendant added instructions, such as TOsize, have other uses and synergies as well. That is, they are not purely namespace baggage. And once again, H3sm has the equivalent of 8DUP, 64DUP, 128XOR, 11+ and so on, innately. There is no semantic cost for them, although bigger is still slower, in most cases. I suspect that some of these oddities would be quite useful in, e.g., graphics, and in conjunction with floating-point hardware. The possibilities are myriad.

Charles Moore mentioned recently that there is some suspicion that the p21's address register may become a stack. He doesn't see it going that way. The part the p21 doesn't have that justifies the extra stack in H3sm is the sizeability of pytes. This has been my interest since before hearing about p21. On the other hand, I settled on "flagpytes" after hearing about the p21's nifty little portable flag bits.

Look Ma, no interrupts! Real-Time Forth

This paper was first presented at the 1998 FORML Conference.

Introduction

Forth systems have been used for real-time applications from its earliest days. These real-time systems have typically been monolithic Forth systems (that is, they were pure Forth systems where Forth provided both the operating environment as well as the application itself).

It is increasingly common for Forth to be used in systems where Forth implements the application but not the operating environment. In the past, I have shown how Forth can still be used practically in these environments. This includes network applications [1] for scripting and to implement WWW CGI [2] and multithreading [3]. Forth can also be used to advantage in real-time systems that are designed in this way. We discuss the kinds of environments likely to be seen when using Forth in this way and present a small example under real-time Linux.

"External" real-time environments

There are many "external" real-time environments that a Forth application may find itself running in. The smallest of these are the real-time microkernels that provide only the real-time scheduler and the functions necessary to support applications that use it. RTKernel [5] is an example of this type of environment. Using this kind of system from Forth simply requires the use of a new API from Forth to reach the real-time functions; otherwise, it is pretty much like a monolithic Forth system.

There are more-sophisticated systems that provide a complete operating system environment, specifically designed for real-time use. An example of this is the VxWorks [6] system which was successfully used in the recent Mars Pathfinder mission. In this type of environment, a proprietary operating system sits underneath the applications, which must use the operating system to provide all the real-time resources and the API to use them. Technically, these types of systems are ideal, since the real-time operating system is specifically designed to handle that part of the tasks.

Proprietary real-time operating systems have the disadvantage that the systems developers must learn a new operating system in order to develop, implement, and test their application. As a consequence of this, many vendors of proprietary real-time operating systems have made an effort, as much as possible, to present a "look and feel" that is similar to more familiar operating systems (most typically, this means Unix). These operating systems also are necessarily rather expensive, because they must provide the complete operating system and support tools, which can make it onerous to use in an R&D environment. In response to these disadvantages, another category of real-time systems has been created. These systems retrofit real-time into an existing general purpose operating system. This has the advantage that the expense of obtaining this type of operating system is limited to the "kernel patch" that makes real-time possible and to the real-time API libraries. This type of retrofit is available for Linux [7] and for Windows NT.

Using Real-Time Linux from Forth

The Real-Time Linux kernel works by running a real-time microkernel underneath Linux. Linux itself is run as a low-priority microkernel task. To run a real-time application, one writes a small module that implements the portion of the application that is the real-time component. This module must conform to the Linux specification for a run-time loadable module [4], using the real-time API. The rest of the application is then a normal Linux program that communicates the control information to the module. The communication mechanism between the two parts of the application are up to the implementor to choose; typically shared memory or a FIFO is used. Because the module only needs to implement the actual real-time code, the module portion is typically quite small. The scale of the main control portion of the code depends upon the demands of the application.

An example

As a simple example, we present an application which must control two Pulse-Width-Modulated (PWM) controlled servomotors, which are commonly used in radio-controlled aircraft using the parallel port of the PC. The real-time code (Listing One) is a simple C program that generates two different square waves on the first two output pins of the parallel port. The controlling Forth program (Listing Two) accepts pairs of numbers from the user (which correspond to the desired position of the two motors) and sends them to the real-time module.

The communication between the real-time module and Forth is accomplished using a FIFO. This communication is accomplished with extensions to the Forth system that allow it to do binary I/O through a Unix file descriptor (which is more general than the ANS Forth File wordset); see Appendix One for details. If a shared memory design was used, then the two processes would use a programmatic interface similar to that described in Appendix Two.

Summary

A system where there is no real distinction between the Forth operating environment and the Forth application is considered ideal by most Forth programmers. However systems where Forth is used to implement the application and

Dr. Everett F. Carter Jr. • skip@taygeta.com
Monterey, California

Everett ("Skip") Carter is the President of Taygeta Scientific, Inc., and is also the President of the Forth Interest Group.

not the environment are increasingly common and are quite practical. We have presented here a case where real-time applications can be implemented in such an environment. In order to be able to use Forth in these kinds of a system, then it must be able to communicate with the real-time environment. This means that, depending upon the type of real-time system being used, that Forth must either have access to the real-time API or that it must have a means of communicating with the real-time tasks. Both of these are fairly easy to achieve in modern Forths.

References

1. Carter, E.F., 1994; "Internetworking with Forth," FORML 1994.
2. Carter, E.F., 1996; "Forth as a scripting language," Rochester Forth Conference, 1996.
3. Carter, E.F., 1996; "Can POSIX threads be used as a Standard Forth Multitasker?" FORML 1996.
4. Rubini, Allessandro, 1998; *Linux Device Drivers*, O'Reilley & Associates, ISBN 1-56592-292-1.
5. RTKernel, On Time Informatik GMbH, Hamburg Germany
6. VxWorks, Wind River Systems.
7. http://luz.cs.nmt.edu/~rtlinux/

Appendix One. Forth System Calls for Multitasking support

The following glossary describes the words added to the standard PFE V0.9.14 compiler in order to support multitasking on Linux/Unix systems.

`close-fd ( fd — flag )`
This word closes a file specified by the file descriptor, *fd*. The *flag* value is non-zero if it fails and zero if it succeeds.

`fork ( — x )`
This word causes the current process to clone itself, including copies of all variables as they are currently set and of all open file descriptor handles. The value of *x* is -1 if the fork fails. If the fork is successful, the value of *x* is 0 for the copied version (the child process), it is the process id of the child process for the original version (the parent process).

`open-fd ( c-addr u fam — fd flag )`
This word opens a file specified by the string *c-addr, u* with the given file-access method, *fam*. The *flag* value is non-zero if it failed and zero if it succeeds. The *fd* is an integer file descriptor of the file.

`pipe ( — rd wr flag )`
This word causes the creation of an anonymous (or un-named) pipe for use with inter-process communication. If it succeeds, *flag* is zero and *rd* is the file descriptor of the read end of the pipe, and *wr* is the fie descriptor of the write end of the pipe. If `pipe` fails, *flag* is -1 and the values of *rd* and *wr* are undefined.

`read-fd ( addr u fd — u' )`
This word reads up to *u* bytes from the filedescriptor, *fd*, and places the data at *addr*. The returned value *u'* is the number of bytes actually read.

`wait ( — p s )`
This word is used by parent processes to wait for the exit of a child process. On return, *p* is the process id of the exiting child process, and *s* is the exit status of the child.

`write-fd ( addr u fd — u' )`
This word writes up to *u* bytes from the data starting at the address, *addr*, to the file descriptor, *fd*. The returned value *u'* is the number of bytes actually written.

Example application

```
\ pipe_f.fth  Example of using pipes for two-way communication
\             between parent and child processes
\
\ This program runs on PFE or GForth
\ with special extensions to handle binary I/O to a
\ file descriptor: open-fd, write-fd, close-fd
\ (see accompanying documentation for details)
\
\ (c) Copyright 1998, Everett F. Carter
\ This program may be used for any purpose provided the above
\ copyright notice is preserved.
\ ==============================================================

VARIABLE wfd
VARIABLE rfd
VARIABLE cnt

128 CONSTANT bufsize

CREATE iobuf   bufsize ALLOT

HEX

: tolower ( cu -- cl )            \ trivial, for demo only
   DUP 40 > IF 20 OR THEN
;
```

```
DECIMAL

: parent_process ( -- )    \ sends (upper case) 'HELLO WORLD' to child

       s" HELLO WORLD"  wfd @ write-fd DROP

       iobuf bufsize rfd @ read-fd iobuf SWAP TYPE CR
;

: child_process ( -- ) \ returns received data converted to lower case
     iobuf bufsize rfd @ read-fd cnt !

     cnt @  0 DO iobuf I + DUP C@ tolower SWAP C! LOOP

     iobuf cnt @  wfd @ write-fd DROP
;

: pipe_test ( -- )

            \ open two pipes
            pipe  0< ABORT" unable to open first pipe "
          pipe  0< ABORT" unable to open second pipe "


            \ now split in two
            fork DUP 0< ABORT" unable to fork "

          0 = IF     \ child
                     \ the child uses the second write pipe and the
                   \ first read pipe, closing the others
                wfd !
              close-fd DROP
              close-fd DROP
              rfd !

             \ close standard I/O, since the child does not use
           \ them.  If they are not explicitly closed in the child
           \ then Forth starts double echoing everything typed.

              0 close-fd DROP
              1 close-fd DROP
              2 close-fd DROP

              child_process

              wfd @ close-fd DROP
              rfd @ close-fd DROP

              BYE
            ELSE   \ parent
                   \ the parent uses the second read pipe and the
                 \ first write pipe, closing the others
               close-fd DROP
               rfd !
               wfd !
               close-fd DROP

               parent_process

               wait   ." wait status " . . CR

               wfd @ close-fd DROP
               rfd @ close-fd DROP

            THEN
;
```

Appendix Two. PFE Forth system interface to shared memory and semaphores

The following glossary describes the words added to the standard PFE v0.9.14 compiler in order to support shared memory and semaphores on Linux/Unix systems.

Shared Memory Words

`shm_alloc ( size -- addr id )`
This word allocates a block of shared memory of `size` bytes. The `id` value is -1 if it failed, otherwise it is an identifying integer that can be used to reference that block by other words. The `addr` value is the memory location of the block.

`shm_attach ( id -- addr flag )`
This word is for attaching to a previously allocated shared memory block. The `id` value is the identification returned by the word `shm_alloc` when the block was first created. The value of `flag` will be -1 on a failure, and 0 if this word succeeds. `shm_attach` can be invoked by a separate process from the one that created the block with `shm_alloc`, thereby providing a mechanism for the two processes to communicate through a memory window.

`shm_detach ( addr -- )`
This word causes the shared memory block to no longer be associated with the address, `addr`. After this word is invoked, references to `addr` are no longer valid, *but the shared memory block still exists* (the block can still be accessed through other attached addresses).

`shm_dealloc ( id -- )`
This word causes the shared memory block referenced by `id` to be released from the system.

Semaphore words

`sem_create ( key val -- id )`
This word creates a semaphore that will be referred to by the returned `id`. The initial value of the semaphore is passed in as `val` on the top of the stack. The semaphore `key` is an arbitrary value that must be unique of for each created semaphore.

`sem_open ( key -- id )`
This word opens a reference to an existing semaphore, identified by `key`. This reference uses the returned `id`.

`sem_close ( id -- )`
This word removes a reference to an existing semaphore, identified by the `id`. The semaphore will still exist on the system, but `id` is no longer a valid reference to it. Typically this word is used by a task that no longer needs to use the semaphore, but there are still other tasks that are using it.

`sem_rm ( id -- )`
This word removes semaphore identified by the `id` from the system. Typically this word is used by the last process that will require the use of the semaphore. Note: the semaphore might not actually leave the system active semaphore list until the process that created it exits.

`sem_wait ( id -- )`
The task that calls this word will block until the semaphore with the given `id` becomes non-zero. When the semaphore has been signaled, then `sem_wait` will decrement it and return. If there are multiple tasks waiting on the same signal, it is undetermined which task will unblock first. Typically `sem_wait` is called immediately before entering a region of restricted or controlled access.

`sem_signal ( id -- )`
This word increments the internal value of the semaphore `id`. This causes tasks that are waiting for a semaphore signal to unblock. Typically `sem_signal` is called after a task has left a region of controlled access.

General considerations

Shared memory and semaphores are limited system-wide resources. The number of shared memory blocks allowed depends upon the configuration of the system, but it is typically a number like 128. This means that it it is generally more efficient to request a just few large blocks of shared memory and to manage them within the application, rather than requesting many small blocks. The maximum size of an individual block is also limited; again the actual value varies, but it is typically four kilobytes. The Unix command `ipcs -l` will show what the limits for the system are.

Example application

```
\ sharmem.fth   Example of shared memory
\
\ This program runs on PFE or GForth
\ with special extensions to handle shared memory on Unix
\ (see accompanying documentation for details)
\
\ (c) Copyright 1998, Everett F. Carter
\ This program may be used for any purpose provided the above
\ copyright notice is preserved
\
\ ===============================================================

 0 VALUE mem1         \ place holders for memory references
 0 VALUE mem2

-1 VALUE id           \ to hold the shared memory block id

   32 shm_alloc        \ allocate some shared memory, 32 bytes
```

```
    TO id
    TO mem1

 \ id would be -1 if the allocation failed

 \ now we can use mem1 like any other address, i.e.
 1234 mem1 !
 mem1 @ . CR

 \ we can get to it through another handle,
 \ this can even be done in a different process
 id shm_attach ABORT" unable to attach"
 TO mem2

 ." note that mem1 and mem2 are different values " CR
 mem1 . CR
 mem2 . CR

 \ but that mem2 has the same data as mem1
 mem2 @ . CR

 \ removing a reference
 mem2 shm_detach

 \ mem2 no longer refers to a valid address

 \ actually removing the shared memory block:
 \ the block will stay around until the system reboots
 \ unless someone eventually does this!

 id shm_dealloc
```

Listing One. The real-time module

```
/* rt_process.c    The Real-Time process that creates TWO PWM signals suitable for
                   RC servos on the first two data bits of the parallel port.
                   Communicates with the Linux side via a FIFO.

   Requires modules  rt_prio_sched.o and rt_fifo_new.o  to be loaded

  (c) Copyright 1998, Everett F. Carter
   This program may be used for any purpose provided the above
   copyright notice is preserved

*/

#define MODULE
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/version.h>
#include <linux/cons.h>
#include <asm/io.h>

#include <linux/rt_sched.h>
#include <linux/rtf.h>
#include <math.h>

/* the address of the parallel port */

#define LPT 0x378

RT_TASK mytask[ 2] ;
int hitime[ 2] ;
int mask[ 2] ;
```

```
float scale = RT_TICKS_PER_SEC / 1000000.0;
int msgsize = 2*sizeof(int);

/* a very simple PWM signal function */
void fun1(int t)
{
      int val;
        RTIME now, when;


        for(;;)
        {
                val = inb( LPT );
                val ^= mask[ t] ;
             outb(val, LPT);        /* write on the parallel port */

              now = rt_get_time();
                when = now + hitime[ t] ;
                while ( when > now )
                          now = rt_get_time();

                val = inb( LPT );
                val |= mask[ t] ;
                outb( val, LPT );

                rt_task_wait();
      }
}

/* receives the duty cycle data from the controlling application */
int iohandler(unsigned int fifo)
{
      int dat[ 2] ;

      if (rtf_get(fifo,dat,msgsize) > 0)
      {
          if ( dat[ 0] < 1 )
        {
             rt_task_suspend( &mytask[ 0] );
             rt_task_suspend( &mytask[ 1] );
        }
          else
        {
            hitime[ 0] = (int)(dat[ 0] * scale);
            hitime[ 1] = (int)(dat[ 1] * scale);
        }
      }

      return 0;
}

/* called when the module is loaded */
int init_module(void)
{
      RTIME now = rt_get_time();

        int period = (RT_TICKS_PER_SEC * 16667) / 1000000;
        int hi = (RT_TICKS_PER_SEC * 1000) / 1000000;

        hitime[ 0] = hitime[ 1] = hi;
        mask[ 0] = 1;
        mask[ 1] = 2;

        rtf_create(1, 100);

        /* create two tasks, one for each PWM signal */
      rt_task_init(&mytask[ 0] , fun1, 0, 3000, 4);
      rt_task_init(&mytask[ 1] , fun1, 1, 3000, 4);

      rt_task_make_periodic(&mytask[ 0] , now + 3000, period);
      rt_task_make_periodic(&mytask[ 1] , now + 4000, period);

        /* create another task to handle control input data */
        rtf_create_handler( 1, &iohandler );

      return 0;

}

/* called when the module is removed */
void cleanup_module(void)
{
        rtf_destroy(1);
      rt_task_delete(&mytask[ 0] );
      rt_task_delete(&mytask[ 1] );
}
```

Listing Two. The controlling Forth program

```
#! /usr/local/bin/forth -q
\ pwmtest.fth    excercises the PWM controls
\
\ This program runs on Real-Time Linux using PFE or GForth with special extensions to handle
\ binary I/O to a file descriptor:  open-fd, write-fd, close-fd
\ (see accompanying documentation for details).
\
\ (c) Copyright 1998, Everett F. Carter
\ This program may be used for any purpose provided the above copyright notice is preserved.
\ ===========================================================================================

-1 VALUE fifo

CREATE sbuf    12 ALLOT


: init-port ( -- n )
        S" /dev/rtf1" 1 open-fd
       ABORT" Unable to open real-time FIFO at /dev/rtf1"
;

: pwm_init
   init-port TO fifo
;

: pwm_close
   fifo close-fd drop
;

: fifo-write ( x y -- )
    sbuf !
    sbuf 4 + !
    sbuf 8 fifo write-fd DROP
;

pwm_init

\ to control the motors,
\ give two positions (in the range 400 to 2400)
\ then fifo-write:
\   400 2000 fifo-write

\ when finished type:
\ pwm_close
```

EVALUATE Macros

This article repeats some old material, but has lots of new stuff. It has been made agreeable with SwiftForth as well as other systems I use. It's a survey of my macro facilities. It shows why I generally prefer EVALUATE to POSTPONE for macros. Not all my uses of EVALUATE are given, but the examples are representative.

All definitions are in Standard Forth. Some definitions have an environmental dependency on 1 CHARS is 1.

Martin Tracy introduced EVALUATE to Forth. (It was called EVAL by him.) John Hayes taught us how to use it.

To me, EVALUATE is an INCLUDED literal or a BLOCK literal. My first implementation of EVALUATE 15 years ago was:

```
: EVAL  ( str len -- ??? )  0 LOAD ;
```

where LOAD had been tweaked.

John Hayes showed that

```
: <name>  S" <text>" EVALUATE ; IMMEDIATE
```

could be used for simple macros.

An important feature of EVALUATE macros is that they can be defined immediately in Standard Forth.

MACRO uses SLITERAL to make the text and expansion easier to see: MACRO <name> " <text>"

```
: MACRO ( "<name> <char> <text><char>" -- )
  : CHAR PARSE
  POSTPONE SLITERAL  POSTPONE EVALUATE
  POSTPONE ; IMMEDIATE
  ;
```

Simple Macros are severely limited—they cannot parse and they cannot have arguments.

Most of the Simple Macros I use could easily be written with POSTPONE. But not all.

```
MACRO NOT  " 0= "
```

I think 0< 0=, 0= 0=, and 0> 0= are ugly, and I would rather write 0< NOT, 0= NOT, and 0> NOT.

I do not want the invocation of 0= to be within a call, particularly since 0= will be optimized before IF with even the simplest peephole optimizer.

I also want to use NOT when interpreting, which I cannot do with POSTPONE.

Check a POSTPONE version.

```
: NOT  STATE @ IF  POSTPONE
  0=  ELSE  0=  THEN ; IMMEDIATE
```

The version using EVALUATE is not STATE-smart.
Optimization is also the reason for the following.

```
MACRO S=  " COMPARE 0= "
MACRO S<  " COMPARE 0< "
```

0= and 0< will combine with a following IF with peephole optimization.

S= NOT IF becomes COMPARE 0= 0= IF becomes COMPARE IF.

The optimization in one of the systems I use does not handle CELLS CELL+ CHAR+ 1+ 1-.

It turns them into subroutine calls rather than extending a literal. It works fine with literals and operators.

```
MACRO CELLS  " 4 * "
MACRO CELL+  " 4 + "
MACRO CHAR+  " 1 + "
MACRO 1+     " 1 + "
MACRO 1-     " 1 - "
```

Now 4 CELLS + CHAR+ becomes 17 +

How could POSTPONE compile a definition with a given name? With EVALUATE there's no problem, *[See Figure One]* where

```
: HI  S" ELECTIVES" INCLUDED ;
```

I do not have MACRO for macros with one or more arguments. Instead I define them explicitly with EVALUATE.

Figure One.

```
MACRO :GO    " ANEW NONCE  : (GO) "
MACRO GO     " (GO)  NONCE "
MACRO GOSEE  " SEE (GO)  NONCE "

MACRO EMPTY  " JOB  ANEW --EMPTY--  DECIMAL "
MACRO JOB    " [UNDEFINED] --EMPTY-- [IF]  HI  [THEN] "
MACRO PANIC  " JOB  ANEW --PANIC--  DECIMAL "
```

Wil Baden • wilbaden@netcom.com
Costa Mesa, California

Wil Baden, after many years of profane languages, has retired to Standard Forth. For the source for this article, send e-mail requesting Standard Forth Tool Belt #9 EVALUATE Macros.

```
: TOKEN   ( -- str len )
    BL WORD COUNT
    DUP ABORT" Unexpected End of Line. "
    ;

: ??   ( "word" -- ??? )
    S" IF "    EVALUATE
    TOKEN      EVALUATE
    S" THEN " EVALUATE
    ; IMMEDIATE
```

How can you write **+TO** in Standard Forth? Temporarily ignoring the problem of **PAD** —

```
: +TO   ( n <value word> -- )
    >IN @  BL WORD COUNT PAD PLACE
    S"  + TO " PAD APPEND
    >IN !  BL WORD COUNT PAD APPEND
    PAD COUNT EVALUATE
    ; IMMEDIATE
```

I don't think the following can be written without **EVALUATE**.

```
(:  before  |  after  |  fill-ins  :)
```

It needs private or general work areas. Here it is using a ring of **PAD** replacements.

```
CREATE QBUF  0 ,  1024 ALLOT
: >QPAD   ( str len -- addr )
  QBUF DUP @ 128 + 1023 AND
                          ( str len addr i)
  TUCK OVER !             ( str len i addr)
    + CELL+ DUP >R PLACE R> ( addr) ;

\ NEXT-WORD  gets next word across line
\ breaks as a char string.  Length of
\ string is 0 at end of file.

: NEXT-WORD                    ( -- str len )
   BEGIN   BL WORD COUNT          ( str len)
           DUP ?? EXIT
           REFILL
   WHILE   2DROP                  ( )
   REPEAT ;                       ( str len)
```

Iterated Interpretation

```
CREATE FILL-IN  128 ALLOT
: .FILL-IN ( -- ) FILL-IN COUNT TYPE ;
```

```
\  (: before|after| word-or-^ phrase^ … :)
: (:
   [CHAR] | PARSE >QPAD
   [CHAR] | PARSE >QPAD  SWAP
                               ( end start)
   BEGIN NEXT-WORD   ( end start str len)
         DUP
   WHILE 2DUP S" :)" S= NOT WHILE
         2DUP S" ^" S= IF
             2DROP  [CHAR] ^ PARSE
         THEN
         2DUP FILL-IN PLACE
         2OVER
           ( end start str len end start)
             COUNT >QPAD >R
             COUNT >QPAD >R
                            ( R: start end)
             ROT DUP >R
             APPEND  COUNT R@ APPEND
             R> COUNT EVALUATE          ( )
             R> R>               ( end start)
   REPEAT THEN ( end start str len)
           2DROP 2DROP
   ; IMMEDIATE
```

Examples

```
1 (: DUP CONSTANT | 1+ |
    JAN FEB MAR APR MAY JUN
    JUL AUG SEP OCT NOV DEC
    :) DROP
```

Polynomial Evaluation

```
( F: x) 0E0 (: FOVER F*   |  F+   |
      a4 a3 a2 a1 a0 :) FNIP

: ISVOWEL   ( char — flag )
  0 (: OVER [CHAR]   |   = OR   |
  A E I O U  :) NIP ;
```

I have several definitions I want to test: **PEASEBLOSSOM**, **COBWEB**, **MOTH**, **MUSTARDSEED**.

```
MACRO [TIME  " :GO COUNTER 1000000 0 DO "
MACRO TIME]  " LOOP TIMER CR ; GO "
(: [TIME | TIME] |
   PEASEBLOSSOM COBWEB MOTH MUSTARDSEED :)
```

This becomes the following.

```
:GO COUNTER 1000000 0 DO
   PEASEBLOSSOM LOOP TIMER CR ; GO
:GO COUNTER 1000000 0 DO
   COBWEB LOOP TIMER CR ; GO
:GO COUNTER 1000000 0 DO
   MOTH LOOP TIMER CR ; GO "
:GO COUNTER 1000000 0 DO
   MUSTARDSEED LOOP TIMER CR ; GO
```

:GO and GO are macros. The first line becomes:

```
ANEW NONCE : (GO) COUNTER 1000000 0 DO
     PEASEBLOSSOM LOOP TIMER CR ;
     (GO) NONCE
```

Macros help with repetitious coding. I found while using classes for input files and ordered lists that I was frequently writing:

```
SRC REWIND  BEGIN  SRC READ WHILE ...
CORPUS REWIND  BEGIN  CORPUS READ WHILE ...
LOANER REWIND  BEGIN  LOANER READ WHILE ...
```

This requires constructing the phrase before evaluating it.

```
\  TRAVERSE <obj>  becomes
\  <obj> REWIND BEGIN <obj> READ WHILE
: TRAVERSE  ( "object" -- )
    TOKEN                      ( str len )
    2DUP >QPAD >R
    S"  REWIND BEGIN "  R@ APPEND
    R@ APPEND                          ( )
    S"  READ WHILE " R@ APPEND
    R> COUNT EVALUATE ; IMMEDIATE
```

Example.

```
INPUTFILE BUILDS SRC
ORDERED-LIST BUILDS LOANER

: SORT-FILE
  HERE >R
    LOANER INIT
    TRAVERSE SRC  LOANER ADD-NEW  REPEAT
    TRAVERSE LOANER  COUNT TYPE CR  REPEAT
  R> HERE - ALLOT ;

  S" xxxxxxxxxx" SRC OPEN
  SORT-FILE
```

Around 1982 Klaus Schleisiek introduced SKIP and SCAN. Their stack-effect is (*str len char — str' len'*). A common use is BL SKIP and BL SCAN.

To extend SKIP and SCAN to check for more than a single value, break apart the two sections of logic.

```
MACRO SKIP[  " BEGIN DUP WHILE OVER C@ "
MACRO ]SKIP  " WHILE 1 /STRING REPEAT THEN "

MACRO SCAN[   " SKIP[ "
MACRO ]SCAN   " 0= ]SKIP "
```

Examples.

```
: JUSTIFY  SKIP[ IS-SPACE ]SKIP ;
: SCAN  >R  SCAN[ R@ = ]SCAN  R> DROP ;
```

where

```
: IS-SPACE  ( char -- flag )  33 - 0< ;
```

As written in Tool Belt #8, PRESWOOP.

ANS Forth specifies word list identifiers as "implementation-dependent single-cell values that identify word lists", which is the weakest possible specification, meaning you know nothing about them. ANS Forth also ignores saving the system after compiling new definitions, and then reloading the system with a possible relocation of addresses.

Some systems, such as PowerMacForth, define a word list identifier (*wid*) so that it is valid only in the session where it's defined. To provide maintenance and transition, WORDLIST: should provide in such systems named word list identifiers that can be used across sessions. The definition of WORDLIST: here is for implementations without a problem with word list identifiers.

```
: WORDLIST:   ( "name" -- )
  WORDLIST CONSTANT ;
```

Here's how it's to be done in PowerMacForth for word list identifier MEMBERS, using implementation dependent words.

```
0 VALUE MEMBERS
VOCABULARY MEMBERS-VOCABULARY
: PMF-RESTORE-MEMBERS
    ALSO MEMBERS-VOCABULARY
    GET-FIRST-WORDLIST TO MEMBERS PREVIOUS
  ;
' PMF-RESTORE-MEMBERS
  RESTORER LINKTOKEN
  PMF-RESTORE-MEMBERS
```

This is an atrocious solution that has to be done for word list identifier CC-WORDS as well.

The following extravagant macro is to provide for these and future word list identifiers. It is very implementation dependent.

```
: WORDLIST:   ( "name" -- )
    TOKEN >QPAD >R
    S" 0 VALUE " >QPAD >R    ( R: arg pad)
( 0 VALUE arg )
        2R@ DROP COUNT R@ APPEND
```

```
   R@ COUNT EVALUATE
( VOCABULARY arg-VOCABULARY )
   S" VOCABULARY " R@ PLACE
   2R@ DROP COUNT R@ APPEND
   S" -VOCABULARY " R@ APPEND
   R@ COUNT EVALUATE
( : PMF-RESTORE-arg )
   S" : PMF-RESTORE-" R@ PLACE
   2R@ DROP COUNT R@ APPEND
   R@ COUNT EVALUATE
( ALSO arg-VOCABULARY )
   S" ALSO " R@ PLACE
   2R@ DROP COUNT R@ APPEND
   S" -VOCABULARY " R@ APPEND
   R@ COUNT EVALUATE
( GET-FIRST-WORDLIST TO arg PREVIOUS ; )
   S" GET-FIRST-WORDLIST TO " R@ PLACE
   2R@ DROP COUNT R@ APPEND
   S"  PREVIOUS ; " R@ APPEND
   R@ COUNT EVALUATE
( ' PMF-RESTORE-arg RESTORER LINKTOKEN )
   S" ' PMF-RESTORE-" R@ PLACE
   2R@ DROP COUNT R@ APPEND
   S"  RESTORER LINKTOKEN" R@ APPEND
   R@ COUNT EVALUATE
( PMF-RESTORE-arg )
   S" PMF-RESTORE-" R@ PLACE
   2R@ DROP COUNT R@ APPEND
   R@ COUNT EVALUATE
2R> 2DROP ;
```

In the source for SWOOP:

```
WORDLIST: CC-WORDS
WORDLIST: MEMBERS
```

[In the Real World], chars have eight bits. They just do. All integral arithmetic is done in twos-complement binary. It just is. Several simple "get real" assumptions like these make our work possible.

Newsbrief

Forth in Space—again

A significant new application of Forth in space was recently launched with the help of several members of the Forth Interest Group.

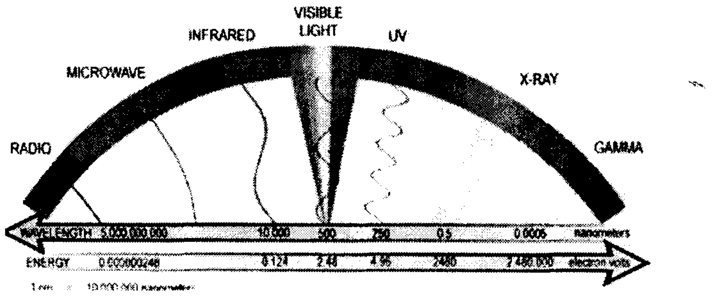
The Chandra x-ray telescope was launched on the space shuttle Columbia on 23 July 1999. This telescope is the x-ray equivalent of the Hubble Space Telescope.

Among the FIG members who worked on the satellite are: Fred Smith, Skip Carter, and Tom Zimmer (Tom may be surprised to learn that he had done anything. His TCOM cross-compiler was used to create the Forth software).

For news and further details about the satellite check: http://chandra.harvard.edu/chronicle/index.html

From the Chandra web site, we extracted the followed illustration and announcement just before press time.

Image courtesy of Chandra Center, Smithsonian Astrophysical Observatory.


VISIBLE LIGHT
INFRARED
UV
MICROWAVE
X-RAY
RADIO
GAMMA
WAVELENGTH 5,000,000,000 10,000 500 250 0.5 0.0005 nanometers
ENERGY 0.000000248 0.124 2.48 4.96 2480 2,480,000 electron volts
1 cm = 10,000,000 nanometers

125 miles: Altitude of early cosmic x-ray observations.
367–381 miles: Altitude range of Hubble Space Telescope's orbit.
6,000–86,000 miles: Altitude range of Chandra X-ray Observatory's orbit.

August 4, 1999 - One of Chandra's sensitive x-ray cameras detected x-rays from a cosmic event even before the door to the observatory has been opened. A solar flare occurred on the afternoon of August 2, and at 5:25 p.m. EDT the High Resolution Camera (HRC) aboard Chandra recorded an increase in the count rate.

FD Code Downloading

Code published in *Forth Dimensions* generally is available to be used without restriction unless otherwise indicated in the code itself or in the text that accompanies it. The general copyright notice for this magazine provides important and more-specific information. Applicable export laws apply.

If no URL is given from which the code you want can be downloaded, contact the author by e-mail.

"SWOOP: Object-Oriented Programming in SwiftForth" — A link to the code can be found in the members-only section of the FIG web site (www.forth.org). Have your FIG member number available when logging on for members-only features.

At ftp.forth.org/pub/Forth/FD/1999 you can find, from this issue:
CrakPoly.zip
PICasmRM.zip
ReedSol.zip
UsrStack.zip

SPONSORS & BENEFACTORS

The following are corporate sponsors and individual benefactors whose generous donations are helping, beyond the basic membership levels, to further the work of *Forth Dimensions* and the Forth Interest Group. For information about participating in this program, please contact the FIG office (office@forth.org).

Corporate Sponsors

AM Research, Inc. specializes in Embedded Control applications using the language Forth. Over 75 microcontrollers are supported in three families, 8051, 6811 and 8xC16x with both hardware and software. We supply development packages, do applications and turnkey manufacturing.

Clarity Development, Inc. (http://www.clarity-dev.com) provides consulting, project management, systems integration, training, and seminars. We specialize in intranet applications of Object technologies, and also provide project auditing services aimed at venture capitalists who need to protect their investments. Many of our systems have employed compact Forth-like engines to implement run-time logic.

Computer Solutions, Ltd. (COMSOL to its friends) is Europe's premier supplier of embedded microprocessor development tools. Users and developers for 18 years, COMSOL pioneered Forth under operating systems, and developed the groundbreaking chipFORTH host/target environment. Our consultancy projects range from single chip to one system with 7000 linked processors. www.computer-solutions.co.uk.

Digalog Corp. (www.digalog.com) has supplied control and instrumentation hardware and software products, systems, and services for the automotive and aerospace testing industry for over 20 years. The real-time software for these products is Forth based. Digalog has offices in Ventura CA, Detroit MI, Chicago IL, Richmond VA, and Brighton UK.

Forth Engineering has collected Forth experience since 1980. We now concentrate on research and evolution of the Forth principle of programming and provide Holon, a new generation of Forth cross-development systems. Forth Engineering, Meggen/Lucerne, Switzerland - http://www.holonforth.com.

FORTH, Inc. has provided high-performance software and services for real-time applications since 1973. Today, companies in banking, aerospace, and embedded systems use our powerful Forth systems for Windows, DOS, Macs, and micro-controllers. Current developments include token-based architectures, (e.g., Open Firmware, Europay's Open Terminal Architecture), advanced cross-compilers, and industrial control systems.

The iTV Corporation is a vertically integrated computer company developing low-cost components and information appliances for the consumer marketplace. iTVc supports the Forth development community. The iTVc processor instruction set is based on Forth primitives, and most development tools, system, and application code are written in Forth.

Keycorp (www.keycorp.com.au) develops innovative hardware and software solutions for electronic transactions and banking systems, and smart cards including GSM Subscriber Identification Modules (SIMs). Keycorp is also a leading developer of multi-application smart card operating systems such as the Forth-based OSSCA and MULTOS.

www.kernelforth.com

An interactive programming environment for writing Windows NT and Windows 95 kernel mode device drivers in Forth.

MicroProcessor Engineering supplies development tools and consultancy for real-time programming on PCs and embedded systems. An emphasis on research has led to a range of modern Forth systems including ProForth for Windows, cross-compilers for a wide range of CPUs, and the portable binary system that is the basis of the Europay Open Terminal Architecture. http://www.mpeltd .demon.co.uk

RAM Technology Systems - Specialists in real-time embedded control. We develop hardware and software from initial idea to final production if required. We have developed the only commercial Forth for the PIC16Cxx range of microcontrollers and now for the AVR. If you need an embedded compiler for your new processor give us a callhttp://www.ram-tech.co.uk • irtc@ram-tech.co.uk

www.theforthsource.com

Silicon Composers (web site address www.silcomp.com) sells single-board computers using the 16-bit RXT 2000 and the 32-bit SC32 Forth chips for standalone, PC plug-in, and VME-based operation. Each SBC comes with Forth development software. Our SBCs are designed for use in embedded control, data acquisition, and computation-intense control applications.

T-Recursive Technology specializes in contract development of hardware and software for embedded microprocessor systems. From concept, through hardware design, prototyping, and software implementation, "doing more with less" is our goal. We also develop tools for the embedded marketplace and, on occasion, special-purpose software where "small" and "fast" are crucial.

Tateno Dennou, Inc. was founded in 1989, and is located in Ome-city Tokyo. Our business is consulting, developing, and reselling products by importing from the U.S.A. Our main field is DSP and high-speed digital.

ASO Bldg., 5-955 Baigo, Ome,Tokyo 198-0063 Japan
+81-428-77-7000 • Fax: +81-428-77-7002
http://www.dsp-tdi.com • E-mail: sales@dsp-tdi.com

Taygeta Scientific Incorporated specializes in scientific software: data analysis, distributed and parallel software design, and signal processing. TSI also has expertise in embedded systems, TCP/IP protocols and custom applications, WWW and FTP services, and robotics. Taygeta Scientific Incoporated • 1340 Munras Avenue, Suite 314 • Monterey, CA 93940 • 408-641-0645, fax 408-641-0647 • http://www.taygeta.com

Triangle Digital Services Ltd.—Manufacturer of Industrial Embedded Forth Computers, we offer solutions to low-power, portable data logging, CAN and control applications. Optimised performance, yet ever-increasing functionality of our 16-bit TDS2020 computer and add-on boards offer versatility. Exceptional hardware and software support to developers make us the choice of the professional.

Individual Benefactors

Makoto Akaishi	Andrew McKewan
Everett F. Carter, Jr.	Peter Midnight
Edward W. Falat	John Muller
Michael Frain	Gary S. Nemeth
Guy Grotke	Marlin Ouverson
Bjorn Gruenwald	John Phillips
John D. Hall	Thomas A. Scally
Guy Kelly	Martin Shann
Zvie Liberman	Werner Thie
Marty McGowan	Richard C. Wagner

21st FORML Conference

"Forth and the Internet"

November 19–21, 1999
Asilomar Conference Center
Pacific Grove, California

Call for Papers

FORML welcomes papers on any Forth-related topics, even those which do not adhere strictly to the published theme.

Please send the title and abstract of your paper to
abstracts@forth.org
Deadline for abstracts: August 31, 1999

For more information, connect to the FORML21 web site:
www.forth.org

Richard C. Wagner, Conference Director
director@forth.org

or contact:

Forth Interest Group
100 Dolores Street, Suite 183
Carmel CA 93923
voice 831.373.6784 • fax 831.373.2845
office@forth.org