$10 Volume XV, Number 4 November 1993 December

# FORTH DIMENSIONS

- Sparse Matrices
- Drawing BMP Files
- UN*X Tools Used on FSAT
- Forth & the Rest of the (DOS) World

# Contents

## Features



## Departments

# Editorial

## A Wholly Unlikely Alliance?

Is it time for the Forth Interest Group and Forth vendors/developers to link arms and walk into the future (or even the present) side-by-side, as a team?

There is something to be said for an alliance of forces within the global Forth community. Vendors could pool some resources to generate more collective clout and a larger market presence than they can achieve individually. The non-profit sector (of which FIG is the largest and oldest representative) could do the same, reducing redundant overhead by consolidating operations that support publishing and conferences, although retaining the autonomy and unique flavor of each organization.

More could be said in favor of collaboration between the non-profit and for-profit sectors of our industry. Profitable enterprises could underwrite development of Forth marketing tools, which have been sorely lacking. Non-profit groups using those tools to generate broad-spectrum interest in Forth could, in turn, refer new inquiries to the Forth companies that supported the marketing effort. And companies that routinely provide FIG literature and mail-order forms to their customers could receive preferred placement or special discounts on advertising in FIG publications.

There would be other less tangible, but equally beneficial, results from more cooperation. Understanding each other better, learning from each other's vision of Forth, and developing a consistent way of talking about Forth's advantages could create a stronger community and would provide leverage for making Forth's viability obvious to those we want to reach with our respective messages and products. But, traditionally, there have been obstacles to such cooperation.

Vendors sometimes have objected to FIG's distribution of "free" Forth systems. The best response I know is that FIG cannot provide the quality control, documentation, technical support, custom programming, and other services customers expect of commercial-grade systems. As any competitive business knows, success is based on much more than possession of a product, regardless of its quality. Seen this way, FIG provides entry-level systems to those who aren't convinced or who don't yet need to invest in a commercial product, and experimental ones that are otherwise unavailable at any price. Forth innovation often has come from the grass roots, like the language itself; public-domain systems sometimes introduce technology that actually benefits commercial enterprise by preventing stagnation and by pushing the state of the art. (One also must wonder how many successful vendors actually could be sustained by sales to the relatively small number of people who acquire only such inexpensive systems.)

But what if professional programmers use those freebies as the basis of programs developed for their employers and clients? They should realize that the overall health of the Forth industry will, sooner or later, affect their ability to sell Forth-based solutions. Over the long haul, we succeed or fail together. Clients should acquire, as part and parcel of any software they contract for, a legitimate license to an underlying commercial Forth system.

Any community can founder on the shoals of special interests, but it can also navigate around such hazards and find open water. As significant as our differences may be, there is more to gain by focusing on our common interests and concerns, and by concentrating on how our unique strengths can complement one another.

Related material from Don Kenney and from Charles H. Small (senior technical editor at *EDN)* is printed in this issue's "Letters." Please read them closely and respond to us with a letter of your own. Further food for thought was offered by Tyler Sperry in an editorial published in *Embedded Systems Programming*:

"...At the other end of the political spectrum, the anarchists who embrace Forth can expect an interesting year as well. The usage of Forth by *Embedded Systems Programming* readers has dropped significantly in the last few years. The reasons are many, but they include the Forth user community's sporadic support of vendors, its blind acceptance of substandard tools simply because they're free or shareware, and—most damning of all—its hesitation to adopt programming conventions that the rest of the programming world has long taken for granted. The coming year will probably bring Forth's last chance for respectability, as the appearance of the ANS Forth standard brings with it a new understanding of how Forth should work in embedded systems. If the Forth community decides not to adopt the standard, however, that's fine with me. By not working in embedded systems, they'll clear the way for more 'happening' languages, such as Smalltalk and APL."

*—Marlin Ouverson*

**Forth Dimensions**
Volume XV, Number 4
November 1993 December

Published by the
**Forth Interest Group**

*Editor*
Marlin Ouverson

*Circulation/Order Desk*
Frank Hall

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

Subscription to *Forth Dimensions* is included with membership in the Forth Interest Group at $40 per year ($52 overseas air). For membership, change of address, and to submit items for publication, the address is: Forth Interest Group, P.O. Box 2154, Oakland, California 94621. Administrative offices: 510-89-FORTH. Fax: 510-535-1295. Advertising sales: 805-946-2272.



*The Forth Interest Group*

The Forth Interest Group is the association of programmers, managers, and engineers who create practical, Forth-based solutions to real-world needs. Many research hardware and software designs that will advance the general state of the art. FIG provides a climate of intellectual exchange and benefits intended to assist each of its members. Publications, conferences, seminars, telecommunications, and area chapter meetings are among its activities.

"*Forth Dimensions* (ISSN 0884-0822) is published bimonthly for $40/46/52 per year by the Forth Interest Group, c/o TPI, 1293 Old Mt. View-Alviso Rd., Sunnyvale, CA 94089. Second-class postage paid at San Jose, CA. POSTMASTER: Send address changes to *Forth Dimensions*, P.O. Box 2154, Oakland, CA 94621."

# Letters

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

### Forth's *Real* Problem

Dear Mr. Ouverson,

I'm glad to see that my article "Forth in Search of a Job" (*FD* XV/1) drew some response. Regrettably, I seem not to have communicated all that well, because Elizabeth Rather spent a lot of time and effort (*FD* XV/2) responding to the wrong issue. The issue isn't that I'm not especially bright and didn't do my homework. The issue is that a very large portion of Forth's potential customer community behaves as if they are rather slow and don't do their homework.

Possibly corporate America is educable, and educating them is worth trying. But bear in mind, we're dealing with people who, for the most part, believed until very recently that PCs would never replace their godawful mainframes. They are not long on insight or foresight. Education, if it's possible, may take many years.

Ms. Rather's reply consists mostly of anecdotal evidence of Forth's great worth. I am very skeptical that anecdotal evidence of Forth's virtues is going to influence corporate American (but I've been wrong before). The problem isn't Forth, but the fact that every language monger claims the same virtues and has lots of anecdotal evidence to back up their claims. This is not to say that good stories about Forth's capabilities don't have their place. But I fear that place may be in quelling customer doubts, rather than in arousing initial interest. In working the customer rather than in getting them to bite.

***The community persists in trying to popularize Forth by using the absolutely least-effective means.***

So I think things are hopeless? No, actually I don't. There are niche markets where I think Forth could do well, were they systematically targeted. Some examples:

- memory-constrained applications
- dead-iron situations—new hardware with no software
- prototyping
- independent software developers

I think the real problem is how to orchestrate a systematic marketing effort for a language when there is no large organization with "unlimited" funds to structure the effort, prioritize targets, pay its employees to write articles, pay for advertising... Possibly the Forth Interest Group would help, at least by serving as a clearing house for identifying markets and thoughts on what is needed to penetrate them.

Sincerely,
Don Kenney
Essex Junction, Vermont

*On a related note, Maris Ambats forwarded this excerpt from* EDN*'s BBS...*

"I have been an editor at *EDN* for ten years. We have done extensive research on the best ways to reach engineers. The Forth community persists in trying to popularize Forth by using the absolutely least-effective means. Forth proponents have consistently tried to prove that Forth is an effective, compact, speedy program-development system with case histories and proof by repeated assertion. In a survey asking engineers to rank 24 kinds of things that could be said about a product, the engineers ranked case histories dead last. Proof by repeated assertion, a style of argumentation endemic to the software world, unfortunately, needs no comment.

"What do engineers want? For a new software system, engineers want self-taught tutorials that they can use to bring themselves up to speed, real-world examples that apply to their jobs, and libraries of functions, routines, and schemas that they can plug into their problems.

"I should mention that I have programmed professionally in polyFORTH and that I am quite aware that Forth is indeed an effective, compact, and speedy software development system. Further, good Forth programming is simply good programming. I find myself using Forth style even when I program in other languages. I am saddened and frustrated that Forth usage is in decline among *EDN* readers. *EDN* is an information provider, not creator. If the Forth community does not create the kind of material that engineers are looking for, then we cannot, obviously, pass it along."

—Charles H. Small
Senior Technical Editor, *EDN*

### Pipe Dreams

I very much enjoyed the FSAT Project article (*FD* XV/2). I'm not sure how compatible BLOCKs are with POSIX, but the idea of melding ideas from Forth, UN*X, and POSIX, and possibly the GNU Project, seems attractive.

Despite the fact that UNIX systems tend to involve "evil" things like preemptive multi-tasking, dynamic memory management, and stream files (note that my tongue is *firmly* in my cheek!), there are some qualities similar to those of Forth:

- Diverse sets of tools.
- Systems for getting the tools to communicate with one another in order to create applications. In Forth, one uses words, compiled into a dictionary, arranged in vocabularies such as FORTH, ASSEMBLER, EDITOR, ... that tend to communicate through two stacks. In UNIX, one uses

programs á la *bc, grep, awk, tr, myprog, a.out,* ... arranged in directories such as */bin, /usr/bin, /ucb/bin, /usr/local/bin, /u/cbbrowne/bin,* ... that tend to communicate via pipes.

I don't know of a precise parallel to CREATE ... DOES> other than the UNIX concept of "little languages," but then I also am not sure about what to do with *RCS* or *make*, which are things that I tend to want to use with Forth, and often can't.

- Opinionated hackers.
- Religious wars.

I think that, in order for Forth to survive, it needs to take advantage of those parallels and, moreover, take advantage of more ideas that were Not Invented Here. Dynamic allocation, streams, and (a logical extension to streams) pipes being three such.

Wouldn't it be neat to have a Forth that would automatically spawn a process for every word invoked from interpretation mode? It might not run happily on an 8051, but then I'm trying to figure out a way to get some form of RISC workstation onto my desk. I'm not thinking about running an 8051. With a suitable tasking model, this sort of system might be a real performer as compared to the *direct* competition, which is *sh* scripts running under UNIX. And it might even work on an 8051.

What I *really* would like to see is a Forth-like approach to pipes.

"On the Back Burner" is always interesting. The philosophical side is neat; the "educational part" on the strange origin of the term "dead reckoning" was illuminating.

We are, I suspect, of somewhat different opinions about some of the other issues; I'm one of those people who likes to claim that Forth is a language with *at least* two stacks, and probably more...

Thank you for some interesting reading.

Yours truly,
Christopher B. Browne
Toronto, Ontario
Canada

### Amended Attribution and a Snap-able Stack

To the Editor and C.H. Ting:

In a recent issue (*FD* XV/2, "More on Numbers"), Ting *hsien-sheng* was very kind in attributing an algorithm for square root to me. It was in fact invented by the ancient Greek mathematicians, as well as the Chinese of course.

It is the classical method for calculating square roots by taking the sum of odd integers. It works for all unsigned numbers from 0 to FFFFFFFF (or FFFF). Try:

```
-1 sqroot .
```

I discovered the Forth code when implementing the sum of odd integers algorithm. It's astonishing how well the Forth primitives work to implement this algorithm. So far it's the only good use for the special features of the Forth-83 +LOOP I have found.

I offer my original implementation as more elegant and efficient than the version that was published.

```
: sqroot            ( radicand -- root )
      -1 SWAP OVER
            DO      ( term) 2 +  DUP +LOOP
      2/
;
```

Although it performs very well for small values, this algorithm is grossly inefficient for large values. If you only need it occasionally for large values that's not important, but you should have an industrial strength definition in your library [see Figure One, below].

This a direct translation of the pencil-and-paper method taught in high school.

Except for +UNDER and NOT the code is ANS Forth. I hope that +UNDER is a primitive in your system. *It allows an element on the stack other than the top to be an accumulator without intermediate stack manipulation.* As UNDER+ it is recommended by Charles Moore in Leo Brody's *Thinking Forth*. I changed the name to be analogous with +! and other words.

```
: +UNDER            ( a b c -- a+c b )
      ROT + SWAP
;
```

A debugging device that ought to be better known is to redefine " (" to be a print instruction followed by a stack dump. This turns stack comments into snapshot traces.

This can be implemented in any Forth, but it is particularly easy to do with PLEASE in this Forth.

*(Continues on page 41.)*

**Figure One.** Baden's "industrial strength" square root.

```
: sqrt      ( radicand -- root )
      0                   ( radicand root)
      0 ADDRESS-UNIT-BITS CELLS 2 -
      DO                                    ( x y)
            2*
            OVER  I RSHIFT      ( . y x')
            OVER  2*  1+ ( . . x' y')
            < NOT IF                  ( x y)
                  DUP                 ( x . y')
                  2*  1+  I LSHIFT  NEGATE
                  +UNDER        ( x y)
                  1+
            THEN
      -2 +LOOP
      NIP                               ( root)
;
```

# Sparse Matrices

*Rick Grehan*
*Peterborough, New Hampshire*

The program presented here is a spin-off of some research I was doing for *BYTE* magazine's lab. The project involved linear programming, which had me dealing with potentially very large matrices. In most cases, the matrices in question were *sparse matrices* (described below), and I developed this program as a means of handling these data structures.

Simply put, a sparse matrix is a large 2D array that's mostly zeroes. If you store such a matrix in standard fashion—i.e., elements stored row-order in a large buffer of contiguous bytes—a substantial portion of memory will be filled with... well, nothing.

In his now-legendary *Fundamental Algorithms*, D. Knuth suggests an alternate storage structure for sparse matrices. I based my implementation loosely on Knuth's, which uses circularly linked lists. The technique shown here is composed of singly linked lists, but the spirit is close to that described in *Algorithms*. The fundamental component of the sparse-matrix storage structure is shown in Figure One. This sparse-matrix element (as I will refer to it for the remainder of this article) is itself a data structure consisting of:

- *Row* and *column* coordinates, which specify the element's position within the array.
- *Right* and *down* pointers, which are the means by which elements are chained to one another.
- The *payload*, which is the actual data.

You can see how this data structure can be used to build a sparse matrix if you examine Figure Two, which shows a portion of a sparse matrix. As mentioned above, the sparse matrix elements are placed into two singly linked lists: one connecting all elements in the same column, the other connecting all elements in the same row.

The heads of these linked lists are two one-dimensional arrays of integers I'll call "anchor" arrays (so named because they anchor the lists of rows and columns). Hence, the zero-th element of the row anchor array points to the first

**Figure One.** Sparse matrix element structure.

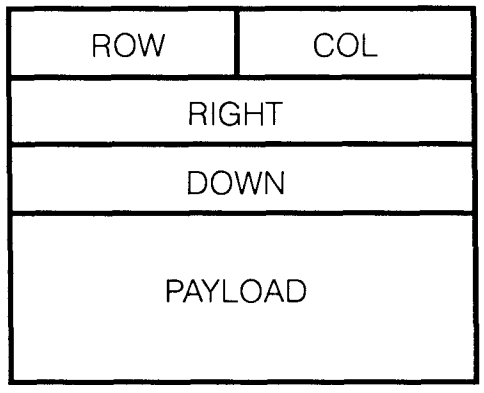
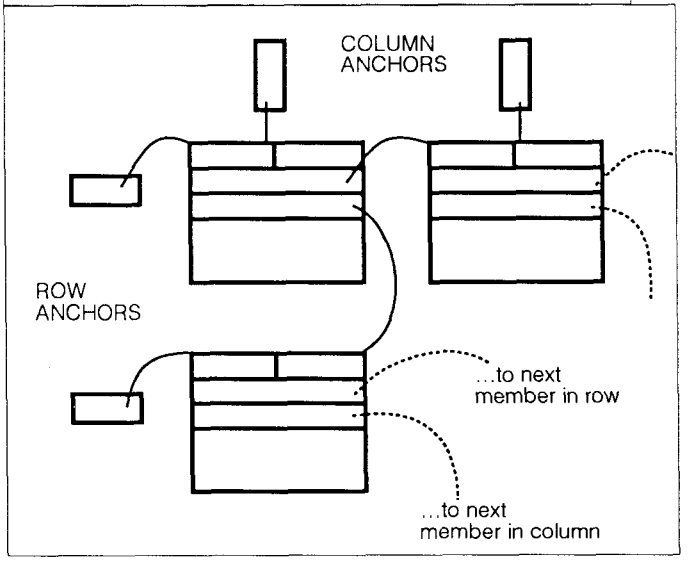



**Figure Two.** Inside a sparse matrix. Members of the "row anchors" array are the heads of the lists linking elements in the same row. Similarly, "column anchors" point to lists of elements in the same column.



element of the zero-th row in the sparse matrix; the zero-th element of the column anchor array points to the first element of the zero-th column in the sparse matrix; and so on.

The right pointer of each sparse-array element leads to the next element in the same row. Similarly, the down pointer of each sparse-array element points to the next element in the same column. A special constant (65535, defined as NIL) acts as an end-of-list indicator.

### Space Considerations

In terms of physical space, a sparse-matrix element (as I have defined it) requires 16 bytes. Each row and column entry is one byte, allowing matrices with logical dimensions up to 256 rows by 256 columns. You can easily extend this limit by changing the row and column entries to 16-bit words and modifying specific constants in the source code. The right and down pointers are 16-bits each, allowing for a total of 64K elements.

The payload in my implementation is a ten-byte, extended floating point (as defined in Apple's SANE documentation). Consequently, each sparse array element requires six bytes beyond an element in a standard array implementation.

When should you use a sparse matrix storage method like the one shown here? Let *a* equal the number of rows in a matrix, and *b* equal the number of columns in a matrix. If you're going to use a sparse matrix, then some of the ele-

**Listing One.** Sparse matrix code.

```
\ ***********************************************************
\ Each sparse matrix element consists of 5 components:
\ 1) A row value (byte)
\ 2) A column value (byte)
\ 3) A right-pointer...points to next member in same row (word)
\ 4) A down-pointer...points to next member in same column (word)
\ 5) A payload...the actual value
\
\ In this implementation, the payload is a 10-byte floating-
\ point number.  The total size of an element is 16 bytes.

\ *****************
\ **  CONSTANTS  **
\ *****************
65535 CONSTANT NIL            \ Indicates end of list

\ *****************
\ **  VARIABLES  **
\ *****************
\ SMATRIX_BASE holds the base address of the memory block
\ holding the sparse matrix elements
VARIABLE SMATRIX_BASE

\ Sparse matrix elements not in use are kept on an available
\ list.  SMATRIX_AVAIL_BASE is the root pointer of that list.
VARIABLE SMATRIX_AVAIL_BASE

\ #SMATRIX_ELEMS holds the number of elements in the
\ sparse matrix.
VARIABLE #SMATRIX_ELEMS

\ We need two 1-dimensional arrays to "anchor" the sparse
\ matrix...one anchors the rows, the other anchors the columns.
\ These are arrays of 16-bit words.
VARIABLE ROWS_ARRAY      \ Pointer to base of row-anchoring array
VARIABLE COLS_ARRAY      \ Pointer to base of column-anchoring array

\ Routines that add to and remove elements from the matrix need
\ to know the previous element in the current vector.
\ We'll keep that item here.
VARIABLE PREV_ELEM

\ Operations on the sparse array usually take place on row or
\ column vectors.  This variable points to the address of the
\ anchoring pointer to the current row or column vector we're
\ working in.
VARIABLE R/C_BASE

\ The sparse matrix has to know how big it is; these variables
\ hold the dimensions.
VARIABLE #SMATRIX_ROWS
VARIABLE #SMATRIX_COLS

\ Following variables are vectors to functions that indicate
\ whether we're searching down through a column or across
\ through a row.
VARIABLE NEXT_FUNC
VARIABLE IDX_FUNC

\ ***************************
\ ** LOW-LEVEL DEFINITIONS **
\ ***************************
: -ROT ROT ROT ;
: ENDIF [COMPILE] THEN ; IMMEDIATE

\ Following words calculate the address of various components
```

```
\ of a sparse matrix

\ Return address of ith sparse array element.
: SMATRIX_ELEM_ADDR ( i -- addr )
  16*                    \ Get byte offset into memory block
  SMATRIX_BASE @ +       \ Add offset to base
;
\ Return address of row member
: &SMATRIX.ROW ( i -- addr )
  SMATRIX_ELEM_ADDR
;
\ Return address of column member
: &SMATRIX.COL ( i -- addr )
  SMATRIX_ELEM_ADDR 1+
;
\ Return address of right-pointer member
: &SMATRIX.RIGHT
  SMATRIX_ELEM_ADDR 2+
;
\ Return address of down-pointer member
: &SMATRIX.DOWN
  SMATRIX_ELEM_ADDR 4 +
;
\ Return address of paylod
: &SMATRIX.VAL ( i -- addr )
  SMATRIX_ELEM_ADDR 6 +
;
\ Set the NEXT function to point to down
: NEXT_IS_DOWN ( -- )
  [COMPILE] LIT [ FIND &SMATRIX.DOWN , ] NEXT_FUNC !
;
\ Set the NEXT function to point to the right
: NEXT_IS_RIGHT     ( -- )
  [COMPILE] LIT [ FIND &SMATRIX.RIGHT , ] NEXT_FUNC !
;
\ Get the NEXT pointer for vector operations
: &SMATRIX.NEXT     ( i -- addr )
  NEXT_FUNC @ EXECUTE
;
\ Set the IDX function to point to ROW
: IDX_IS_ROW   ( -- )
  [COMPILE] LIT [ FIND &SMATRIX.ROW , ] IDX_FUNC !
;
\ Set the IDX function to point to COL
: IDX_IS_COL   ( -- )
  [COMPILE] LIT [ FIND &SMATRIX.COL , ] IDX_FUNC !
;
\ Get the IDX member
: &SMATRIX.IDX ( i -- addr )
  IDX_FUNC @ EXECUTE
;
\ Set the row base pointer
: SET_ROW_BASE ( n -- )
  2*                          \ 2 bytes per 16-bit word
  ROWS_ARRAY @ +              \ Add offset to base address
  R/C_BASE !                  \ Store
;
\ Set the column base pointer
: SET_COL_BASE ( n -- )
  2*                          \ 2 bytes per 16-bit word
  COLS_ARRAY @ +              \ Add offset to base address
  R/C_BASE !
;
\ ************************************
\ Build the sparse matrix available list
: BUILD_SMATRIX_AVAIL    ( -- )
```

ments are going to be empty; I'll represent the number of empty elements with *e*. The point at which it becomes beneficial to use a sparse matrix is when the following equation is satisfied:

$10ab > 2(a+b)+16(ab-e)$

That is, using standard techniques to store a matrix of extended reals requires $10(ab)$ bytes. A sparse array would require $2(a+b)$ bytes for the anchoring arrays plus $16(ab-e)$ bytes for the elements actually active in the array. This equation assumes a payload of ten bytes, and a sparse-array element size of 16 bytes.

You can determine how many empty elements would make using a sparse array "profitable" by re-arranging the above equation to:

$(a+b+3ab)/8 < e$

So, for example, if you are working with a 50 x 50 matrix, it becomes worthwhile to look into using the sparse matrix storage format if more than 950 elements are empty.

### The Code

The complete source code appears in Listing One. In practice, the first word your program must call is `INIT_SMATRIX`, which allocates space for the sparse-matrix elements as well as the anchor arrays (via `INIT_ANCHOR_ARRAY`). `INIT_SMATRIX` also places all sparse-matrix elements on an "available list" (the word `BUILD_SMATRIX_AVAIL` performs this task), yet another singly linked list that holds all unused sparse-matrix elements.

Initially, then, the two anchoring arrays point to empty lists; all sparse matrix elements are on the available list.

You build a sparse matrix by repeated calls to FROM_SMATRIX_AVAIL and INTO_SMATRIX. FROM_SMATRIX_AVAIL pulls an unused element from the available list and leaves that element's "identifier" on the stack. This identifier is a unique handle to the element, and is the means by which the program references an element whether it is in the array or on the available list. You load the returned element's payload with the appropriate value and execute INTO_SMATRIX. This word wires the element into the sparse matrix at the row and column coordinates specified on the stack. (If an element is already in the matrix at the given coordinates, the system exits with an error condition.)

The word &SMATRIX_VAL provides access to the payload. It expects an element identifier on the stack and returns the address of that element's payload component.

As you perform mathematics on the matrix members—pivoting operations, for example—some elements' payloads will be reduced to zero. (You'll have to decide what a zero is for your particular application. It might mean anything from an "honest-to-goodness zero" to "a very small number.") In that case, your code should call OUTOF_SARRAY. This word accepts on the stack a sparse-matrix element identifier; the element is presumed

*(Text and code continue on page 38.)*

```
  0 SMATRIX_AVAIL_BASE !       \ Anchor first element
  #SMATRIX_ELEMS @ 1- 0
  DO
    I 1+                       \ Each entry points to next higher
    I &SMATRIX.RIGHT W!        \ Store address in right pointer
  LOOP
  NIL                          \ Acts as terminator
  #SMATRIX_ELEMS @ 1-          \ Get address of last guy
  &SMATRIX.RIGHT W!            \ Attach terminator
;
\ ********************************************
\ Following are a couple of low-level definitions
\ that provide access to some Macintosh traps.
\ These traps allow us to allocate and release non-
\ relocatable chunks of memory.

\ _NEWPTR allocates n bytes and returns pointer to the memory
\ location. Returns NIL on failure.
<CODE _NEWPTR  ( n -- ptr/0 )
  POPD0,
  MAC NEWPTR W,
  PUSHA0,
  NEXT,

\ _DISPOSPTR releases the allocated memory
<CODE _DISPOSPTR    ( ptr -- )
  POPA0,
  MAC DISPOSPTR W,
  NEXT,

\ ********************************************************
\ Initialize an anchoring array [the row or column anchors
\ arrays].  addr is the address if the base variable, and n
\ is the number of elements
: INIT_ANCHOR_ARRAY ( addr n -- )
  DUP 2*                       \ # of bytes to allocate
  _NEWPTR                      \ Allocate memory
  ?DUP 0=                      \ Allocation failed?
    ABORT" Anchor array alloc. error"
  ROT OVER SWAP !              \ Save base address

  \ Now set all elements to 0xFFFF
  SWAP 0 DO
    NIL OVER I 2* + W!
  LOOP
  DROP                         \ Clean stack
;
\ ******************************************************
\ Initialize a sparse matrix.  Presumes #SMATRIX_ROWS and
\ #SMATRIX_COLS have been properly initialized
: INIT_SMATRIX ( -- )
  \ Initialize the rows array
  ROWS_ARRAY #SMATRIX_ROWS @ INIT_ANCHOR_ARRAY

  \ Initialize the columns array
  COLS_ARRAY #SMATRIX_COLS @ INIT_ANCHOR_ARRAY

  \ Initialize the sparse matrix memory
  #SMATRIX_ELEMS @ 16* _NEWPTR
  ?DUP 0= ABORT" Sparse array alloc error"
  SMATRIX_BASE !
  BUILD_SMATRIX_AVAIL
;
\ ************************************************************
\ Dispose of all the memory space taken up by the sparse matrix
```

# Forth and the Rest of the (DOS) World

*Richard Astle*
*La Jolla, California*

Most computer languages implemented for MS-DOS on the IBM-PC have a compile-link cycle which interferes with interactivity but allows them one advantage Forth rarely has: the ability seamlessly to include functions written in other languages. It has been said, with some justification, that this foreign code is often of poor quality, unsuitable and slow, and if we really want to do it right we should do it ourselves. But blind self-reliance can be expensive. Sometimes time is too tight or sources unavailable. And sometimes we actually *can't* do it better. In these cases, at least, it would be nice to be able to use other people's libraries and code, particulary if we can do so without losing the interactive, incremental, soul of Forth.

There are several ways to go about using the resources of other languages from within Forth. One is to put functions, procedures, or subroutines written in the other language(s) into a TSR and have Forth communicate with them through an interrupt or a jump table at a known location in memory. This method isn't difficult, at least on the Forth side, and it can be useful in some situations, but basically it's just a way around the fact that most Forths can't link.

***Nothing is portable except ideas, in C, Forth, or, for that matter, life.***

Another method is to write Forth in C or assembler or some other language, and link the desired external functions at the beginning or add them later by recompiling. Forth has, in fact, often been written in assembler, either as a way of bootstrapping or in the perhaps mistaken idea that assembler is easier or more transparent than meta-compilation. It would be easy to extend these Forths through the linker. One problem is that this method makes it impossible to add new external references without starting again from the assembler source; another is that, once you've appropriated external functions, it is difficult to save and reload the Forth image without going through the linker step all over again. As a consequence, you have to keep the .LIB or .OBJ files containing the object code for the appropriated functions in your working environment, and take them with you when you travel, with your system, to another machine.

A third method, the one I shall discuss, is to start, not with assembler or C, but with Forth itself. I have taken a straightforward, single-segment, indirect-threaded, 83-standard Forth, added a variety of external references to functions written in C, saved the Forth memory image containing these references as an object module (OBJ) file, and linked that module (using Borland's TLINK) with C libraries and object modules into an EXE which, when run, makes those functions available as normal Forth words. At the end of this process I have a FORTH.EXE which I can extend with further external functions via the linker, but also via normal Forth incremental compilation without the linker.

### 1. The Source Forth

Every Forth system is idiosyncratic, especially at the edges, and this topic is pretty edgy. The relationships to the operating system, to extended memory, and to the native machine language of the real machine (as opposed to the Forth pseudo-machine) are not, and cannot, be made standard. The particular Forth I begin with here conforms closely to the Forth-83 Standard, having been written as a teaching tool by Guy Kelly, the chair of the Forth Standards Team, but it of course has certain extensions. For memory access outside the 64K Forth segment, I assume the existence of the following semi-standard words, with these stack pictures:

```
@L   ( seg offset --- n )
C@L  ( seg offset --- c )
!L   ( n seg offset --- )
C!L  ( c seg offset --- )
CMOVEL ( source-seg source-off dest-seg
         dest-off len --- )
```

I also assume the existence of a few utility words like `CS@`, `DS@`, and `ES@`, trivial to define in assembler (or directly in machine code with `C,`), which push the contents of the segment registers onto the stack. Of course, in a single-segment model they'll all return the same value. I also assume the familiar F83 vectored execution mechanism using `DEFER`, `[']`, and `IS`.

For DOS file access I assume the existence of the

following words:

```
MAKE
( fbuf --- )

FREOPEN
( fbuf --- )

FCLOSE
( fbuf --- )

WRITE
( addr len fbuf --- )

READ
( addr len fbuf --- )

SEEK
( udOffset fbuf --- )

FILENAME
( fbuf --- )
```

These words call DOS int 21 functions 3C through 40 and 42. They assume the 16-bit addr is relative to DS and that the unsigned double offset for SEEK is from the beginning of the file. The parameter fbuf is a stand-in for the DOS file handle. In this Forth, FBUF is a defining word that creates an object which contains a space for a filename, a handle, and a few file statistics. The word FILENAME is used as in

```
FBUF FBUF1
FBUF1 FILENAME
FORTH.OBJ
```

to assign a filename to an FBUF since words like MAKE and FREOPEN need a name rather than a handle. The word FREOPEN is a safety word: first it closes a file that might have its handle in the FBUF in question, then it opens the file which has its name in the FBUF. This prevents a careless loss of handles. The word Z"—which handles the embedding of null-terminated C-like strings—uses the words ASCII and (") which aren't quite standard though they or their equivalents must exist in nearly all Forths. ASCII is sometimes called C' and just leaves the ASCII value of the following character on the

```
Screen 0
This file converts Guy Kelly's PC-FORTH 1.46, an 83-Standard
Forth implementation, from a COM to an EXE file, with links to
external functions written in C.  Since the program began as a
COM file when it becomes an EXE the area from below 100h is
available for scratch data.

Richard Astle
POBox 8023
La Jolla, CA 92038
619 456-2253

Screen 1
\ USEFUL THINGS                                  02JAN93 RA 29DEC92
HEX
ONLY FORTH ALSO DOS ALSO FORTH DEFINITIONS
.IFNDF OFBUF   FBUF OFBUF  .ENDIF    \ fbuf for this work
: WALL ;                             \ marker for FORGET
RE- : BYE 4C BDOS ;                  \ 0 BDOS is for .com files
-->

Screen 2
\ USEFUL THINGS: DOS READ WRITE SEEK    10FEB93 RA 29DEC92
: WRITE      ( addr len fbuf --- )
    -ROT  WRITE? 0= IF CR ." WRITE ERROR "  QUIT THEN ;
: READ       ( addr len fbuf --- )
    -ROT  READ?  0= IF CR ." READ ERROR " QUIT THEN ;
: SEEK       ( dOffset fbuf --- )
    SEEK?  0= IF CR ." SEEK ERROR "  QUIT THEN ;
: MAKE       ( fbuf --- )
    MAKE? 0= IF CR ." MAKE ERROR " QUIT THEN ;
-->

Screen 3
\ REDEFINITIONS                                    RA 16JAN93
\ redefinitions to correct parameter order
RE- : @L ( seg off --- n ) SWAP @L ;
RE- : C@L                  SWAP C@L ;
RE- : !L ( n seg off --- ) SWAP !L ;
RE- : C!L                  SWAP C!L ;
RE- : CMOVEL ( seg off seg off len --- )
         >R SWAP 2SWAP SWAP 2SWAP R> CMOVEL ;
-->

Screen 4
\ USEFUL THINGS  @L &c DUMPL                       RA 16JAN93
DECIMAL
: DUMPL ( seg adr cnt --- )
    BASE @ >R HEX
    0 DO CR OVER 5 U.R DUP 5 U.R 2 SPACES
        16 0 DO 2DUP C@L 3 U.R 1+ LOOP
        16 - 2 SPACES
        16 0 DO 2DUP C@L DUP BL < OVER ASCII ~ > OR
            IF DROP ASCII . EMIT ELSE EMIT THEN 1+
        LOOP
        KEY? ?LEAVE
    16 +LOOP 2DROP R> BASE ! ;
-->
```

```
Screen 5
\ BOOTZ to avoid default drive problem          RA 23JUL93
: BOOTZ
    FBUFS-INIT FYL0 FYL ! SET^BREAK
    SET-#DRIVES
    (BOOT) ;                                            -->

Screen 6
\ USEFUL THINGS     SET-BOOT            02JAN93 RA 29DEC92
DECIMAL
: SET-BOOT    \ sets boot-up variables
    ['] BOOTZ IS BOOT
default-drive# ON
    LATEST      8 +ORIGIN !   \ top nfa     boot
    HERE       22 +ORIGIN !   \ fence         up
    HERE       24 +ORIGIN !   \ dp            literals
    VOC-LINK @ 26 +ORIGIN ! ; \ voc list
HEX

: GET-MSGS  CS@ 1000 + 0  CS@ FIRST  200   CMOVEL ;
: @! ( addr1 addr 2 --- ) SWAP @ SWAP ! ;
\ : NIP SWAP DROP ; \ if you need it
: UMIN ( u1 u2 --- umin ) 2DUP U< IF DROP ELSE NIP THEN ;
-->

Screen 7
\ ZSTRING                                           RA 05FEB93
: $MOVE ( $adr dest --- ) \ moves string including count
    OVER C@ 1+ CMOVE ;
: Z,"
ASCII " WORD HERE $MOVE
1 HERE +C!
HERE C@ ALLOT 0 C, ;
: Z" COMPILE (") Z," COMPILE DROP ; IMMEDIATE COMPILE-ONLY
-->
Z" compiles an in-line string and appends a byte of 0.
At run-time it leaves the address of the first byte of
the string on the stack, suitable for passing as a character
string pointer to a C function.  It is thus comparable to "
which in this Forth leaves address and count and, of course,
doesn't append a 0 byte.

Screen 8
\ FINDING THE PSP AND ENVIRONMENT SEG   15FEB93 RA 16JAN93
HEX
CODE PSP@  ( --- psp-seg )
    B8 C, 00 C, 51 C,    \ mov ax,51h
    CD C, 21 C,          \ int 21h
    53 C,                \ push bx
NEXT,
END-CODE
: ENV@ PSP@ 2C @L ;
CODE DS@ 1E C, NEXT, END-CODE  CODE ES@ 06 C, NEXT, END-CODE
-->

Screen 9
\ EXTERN: LINKED LIST                               RA 03JAN93
```

stack. (") is embedded in a word before a compiled counted string. Its action is to leave the address and count of the string on the stack and to skip over it to the word following the string. Its definition is as follows:

```
: (") ( --- addr count)
  R> COUNT 2DUP + >R
; COMPILE-ONLY
```

I explain these details so that you will know how to write equivalent words in the Forth you use if you don't already have them, and/or understand my code well enough to adapt it. Nothing is portable except ideas, in C, Forth, or, for that matter, life.

### 2. EXE Files And Link

A COM file is just a memory image, with the restriction that it cannot be more than 65,178 bytes in length. An EXE file, by contrast, consists of two parts: a header and a "load module." The load module is approximately a memory image of the executable program represented by the EXE file. The reason the memory image is "approximate" has to do with the way the 80x86 processors address memory in real mode. Any reference (CALL, JUMP, etc.) to a memory address more than 64K away has to include an actual segment reference. These segment references cannot be known at compile time: on the contrary, they have to be "fixed up" when the program is loaded and run.

It is possible to have an EXE file without fix-ups. A Forth system I use daily has four 64K segments and sleeps in an EXE file, but since only one of these segments contains executable code, there is no need for so-called relocations: when the program loads, the DS, ES, and SS registers are set to appropriate offsets from CS and everything runs

smoothly. This EXE file was not created by LINK, however, and has no way to speak to anything it doesn't compile or metacompile for itself.

When the linker links OBJ and LIB files to create the EXE, it creates an almost executable image that could run if it could be loaded at the very bottom of memory, without a PSP or a memory allocation block below it. In other words, the linker makes all segment references in the load module relative to the beginning of the load module. When DOS loads the load module into memory it always does so on a segment boundary. All that is necessary, then, to "fix up" the file for execution is to take the load address as a segment value and add it to all segment references in the loaded image.

To facilitate this fixing up, the EXE header contains a "relocation pointer table." This table is poorly named: nothing is actually relocated. What it actually contains is just a list of pointers to segment references in the load module, represented as segment:offset pairs, relative to the beginning of the load module. This is exactly enough information to find the locations in the file that need to have their contents adjusted. In fact, this is all very simple: the hard work is done by the linker and, before that, by the language compiler/translator that creates the OBJ files.

### 3. External References

Functions written in C are accessed by calls. The basic format of these calls is described in the mixed-language programming sections of C and assembler manuals and books, where you can also find details of calling conventions for Pascal, Fortran, BASIC, etc. Parameters are passed on the stack and (unlike most

```
VARIABLE LAST-XLINK     \ pointer to last extern in linked list
VARIABLE 1ST-XLINK      \ pointer to first extern
0 DUP 1ST-XLINK ! LAST-XLINK ! \ mark list empty

: EXT-LINK,                 \ install the link
    LAST-XLINK @ ?DUP       \ is there already at least 1 extern?
    IF HERE SWAP !          \ store fwd link in previous extern
    ELSE HERE 1ST-XLINK ! \ or in linked list base pointer
    THEN HERE LAST-XLINK ! \ store current link in head pointer
    0 , ;                   \ terminate list
-->

Screen 10
\ EXTERN: LINKED LIST TRAVERSAL                    RA 03JAN93
: XLINK>FIXADR  ( ext-link --- patch-adr )  2- @ ;
: XLINK>XNAME   ( ext-name --- xname-adr )  2+ ;
VARIABLE CUR-XLINK   VARIABLE XREF#
: >1ST-XLINK 1ST-XLINK   CUR-XLINK @! 1 XREF#  ! ;
: >NXT-XLINK CUR-XLINK @ CUR-XLINK @! 1 XREF# +! ;
: NEXT,
    AD C,           \ LODSW
    93 C,           \ XCHG AX,BX
    FF C, 27 C, ;   \ JMP  [BX]
-->

Screen 11
\ EXTERN: SAVE-REGS & RESTORE-REGS      04FEB93 RA 03JAN93
HEX     CREATE RG-SV 8 ALLOT
HERE
    89 C, 2E C, RG-SV     , \ BP -> RG-SV       FORTH RP
    89 C, 1E C, RG-SV 2+  , \ BX -> RG-SV+2     FORTH W
    89 C, 36 C, RG-SV 4 + , \ SI -> RG-SV+4     FORTH IP
    8C C, 06 C, RG-SV 6 + , \ ES -> RG-SV+6
    C3 C,                   \ RET
CONSTANT SAVE-REGS
HERE
    8B C, 2E C, RG-SV     , \ BP
    8B C, 1E C, RG-SV 2+  , \ BX
    8B C, 36 C, RG-SV 4 + , \ SI
    8E C, 06 C, RG-SV 6 + , \ ES
    C3 C,                   \ RET
CONSTANT RESTORE-REGS                    -->

Screen 12
\ EXTERN: CODE-EXTERN                    15FEB93 RA 03JAN93
HEX
: CODE-EXTERN   ( #out #in --- cfa patch-address )
    HERE            \ -- #out #in cfa
    E8 C, SAVE-REGS HERE 2+ - ,             \ CALL SAVE-REGS
    9A C, HERE 0 , 0 , \ -- #out #in cfa patch \ CALL 0000:0000
    ROT ?DUP
    IF              \ -- #out cfa patch-addr #params
        \ clean up the stack in C fashion
        83 C, C4 C, 2* C,                   \ ADD SP,2*#params
    THEN            \ -- #out cfa patch-addr
    E8 C, RESTORE-REGS HERE 2+ - ,          \ CALL RESTORE-REGS
    ROT ?DUP        \ -- cfa patch 0 | cfa patch n n
    IF 50 C, 1-             \ PUSH AX
```

```
        IF 52 C, THEN           \ PUSH DX
    THEN  NEXT, ;  -->

Screen 13
\ EXTERN                                    15FEB93 RA 03JAN93
\ : HEADER CREATE -2 ALLOT ; \ if you need it
: CREATE-EXTERN ( cfa patch-addr --- )
    HEADER \ lays down Forth name and link for this word
    SWAP , ,  EXT-LINK, \ | cfa | patch | next-extern |
    BL WORD
    \ HERE $MOVE HERE \ include if WORD doesn't work at HERE
    C@ 1+ ALLOT ;  \ | linker-name |

: EXTERN ( #out #in --- )       ( ... --- ax dx | ax |  )
   OVER 0 2 WITHIN 0= IF ." INVALID # OUT PARAMS " ABORT THEN
    CODE-EXTERN          \ cfa patch-addr
    CREATE-EXTERN ;
-->      USAGE: #out #in EXTERN DOG _dog
creates Forth word DOG to execute external _dog
with #in params on stack before and #out after

Screen 14
\ SHOW-FIXUPS   UTILITY WORD                        RA 17JAN93
: 4H.R BASE @ SWAP HEX 4 .R BASE ! ;
: SHOW1FIXUP ( xlink --- )
    DUP              4H.R 2 SPACES     \ XLINK
    XLINK>FIXADR DUP 4H.R 2 SPACES     \ FIXADR
    2@ SWAP 4H.R ." :" 4H.R SPACE      \ FIXUP
    DUP 2+ COUNT TYPE ;
: SHOW-FIXUPS
    CR ." FIXUPS "
    CR ." XLINK FXADR  SEG:OFF  label "
    CR  1ST-XLINK
    BEGIN @ ?DUP
    WHILE DUP SHOW1FIXUP  CR
    REPEAT ;   -->

Screen 15
\ SAVE-OBJ: OBJ RECORD WRITING          03JAN93 RA 29DEC92
HEX FFFF CONSTANT TOP.ADDR  \ we save up to this addr = 64K
VARIABLE RECBUF          \ holds address of record buffer
VARIABLE RECBUFP         \ pointer into record fbuffer
: 0BUF RECBUF RECBUFP @! ;  \ sets pointer to beginning of buf
: REC.HERE RECBUFP @ ;       \ a minor convenience

: BUF$, ( $adr --- )     \ move and "comma-in" a string
    REC.HERE $MOVE  REC.HERE C@ 1+  RECBUFP +!  ;
: (BUF,")                \ based on (.")
    R> DUP COUNT + >R  BUF$, ;  COMPILE-ONLY
: BUF,"     COMPILE (BUF,")     ,"     ; IMMEDIATE COMPILE-ONLY
: BUFC, ( c --- )   REC.HERE C!   1 RECBUFP +!  ;
: BUF,  ( n --- )   REC.HERE  !   2 RECBUFP +!  ;
-->
: DMP.REC RECBUF @ REC.HERE OVER - DUMP ;

Screen 16
\ SAVE-EXE: WR.REC                                  RA 29DEC92
VARIABLE REC-FBUF  \ other Forth DOS interfaces use a handle
```

other languages) are left there for the caller to clean up. Functions that return values return them in registers: in Borland and Microsoft C, 16-bit values are returned in AX and 32-bit values in DX;AX. Since Forth tends to live in (single or multiple) 64K segments, the external C (or other) functions we use will have to be referenced by long calls (segment:offset) and should therefore be compiled in the medium, large, or huge models, or at least be declared as "far" or "huge." The huge model, though of course slowest, is the most straightforward, since huge functions save and set the DS register when they're entered and restore it when they leave, keeping Forth from having to know where C keeps its data, and preventing unexpected access to Forth's own. The only data we have to share is passed (directly or by reference) on the stack, which had better be big enough.

So in Forth we need a way to make long calls to places we don't know about, in such a way that the Microsoft or Borland linker can fix them up for us. I have written a defining word, EXTERN, which lays down code for long calls and links all the words it creates into a forward-linked list of external references so they can be found easily when we write out the OBJ file.

EXTERN takes two parameters on the stack and is followed by two names: the Forth word that will invoke the long call, and the external reference that the linker will use to resolve the address of the call with the appropriate C function. The usage is

```
EXTERN <Forth-name>
<external-reference>
( #out #in --- )
```

as in

```
1 2 EXTERN GETPIXEL
_getpixel
```

This expression defines a Forth word GETPIXEL, which has a stack picture:

```
GETPIXEL
( y x --- color )
```

The parameters to EXTERN indicate that GETPIXEL takes two 16-bit values from the stack and returns one. Note that these are not parameters in the C sense, but rather the number of 16-bit values that make up those parameters: an int is worth one 16-bit parameter, a double two, a far pointer also two. Since a C function returns at most one parameter and, in Borland C at any rate, the maximum size of that parameter is 32 bits, the first parameter to EXTERN can be only 0, 1, or 2. There can, of course, be any number of input parameters. The reason EXTERN wants to know how many there are is so that the word it defines can clean up the stack by adjusting SP before it pushes its return value, if any. This is the way C does it. Of course EXTERN could be written more simply, even more generally, to leave the input parameters on the stack and push both AX and DX on top of them. For those few functions, like `printf()`, which take a variable number of parameters, we'll have to clean up the stack in high level anyway, but we're Forth programmers, we can handle it.

Note that C, unlike Pascal for example, pushes its function parameters on the stack from right to left, so that, in this example, y x GETPIXEL corresponds to:
`getpixel(x,y)`

The final term in the EXTERN expression, _getpixel in the example, is the actual external refer-

```
: REC-SIGNATURE, ( b --- )
    0BUF            \ initialize buffer
    BUFC,           \ signature byte
    0 BUF,  ;       \ placeholder for length
: CHKSUM,       ( --- )
    0 BUFC, ; \ fake, irrelevant, checksum
: (WR.REC)  ( --- )
    RECBUF @   REC.HERE  OVER -   REC-FBUF @  \ addr len fbuf
    WRITE   ;
: !RECLEN   \ calculate and store record length in record
    REC.HERE RECBUF @ - 3 -     \ length of record after count
    RECBUF @ 1+ ! ;
: WR.REC  CHKSUM,  !RECLEN  (WR.REC) ; -->  \ write record

Screen 17
\ SAVE-OBJ: RECORD TYPES: THEADR LNAMES MODEND  RA 29DEC92
HEX
: WR.THEADR
    80 REC-SIGNATURE,    \ signature byte & space for length
    BUF," FORTH"         \ content of field, a counted string
    WR.REC ;             \ do count and fake checksum &
: WR.LNAMES
    96 REC-SIGNATURE,
    BUF," "     BUF," FORTH_TEXT"      BUF," CODE"
    WR.REC ;
: WR.MODEND
    8A REC-SIGNATURE,
    00 BUFC,
    WR.REC ;
-->

Screen 18
\ SAVE-OBJ: RECORD TYPES: SEGDEF        22JAN93 RA 29DEC92
HEX
: WR.SEGDEF ( class_index seg_index len ACBP --- )
    98 REC-SIGNATURE,
       BUFC,   \ ACBP 62 = relocatable, para aligned, no combine
       BUF,    \ 0000 = 64K segment
       BUFC,   \ segment name index    These indexes refer to
       BUFC,   \ class name index      names in the LNAMES
     1 BUFC,   \ overlay name index  - ignored
    WR.REC ;
-->

Screen 19
\ SAVE-OBJ: RECORD TYPES: EXTDEF                RA 02JAM93
HEX
: WR.EXTDEF
    8C REC-SIGNATURE,
    >1ST-XLINK                  \ point to the first extern
    BEGIN CUR-XLINK @ ?DUP      \ does the extern exist?
    WHILE XLINK>XNAME BUF$,     \ write the linker name
        0 BUFC,                 \ type = none
        >NXT-XLINK              \ get to the next extern
    REPEAT
    WR.REC ;
-->
An EXT-DEF contains list of names imported from other modules,
```

```
found by stepping through the EXTERN linked list.

Screen 20
\ SAVE-OBJ: RECORD TYPES: PUBDEF                    RA 29DEC92
HEX
: WR.PUBDEF
    90 REC-SIGNATURE,
     0 BUFC,     \ group index
     1 BUFC,     \ seg index
    BUF," _forth"
   100 BUF,      \ offset
     0 BUFC,     \ type = none
    WR.REC ;
-->
PUBDEF contains a list of names in this module to be exported
to other modules

Screen 21
\ SAVE-OBJ: RECORD TYPES: LEDATA                    RA 29DEC92
HEX
VARIABLE MEMP
: MEMP@   MEMP @ ;
VARIABLE SEG-INX
: WR.LEDATA ( length --- )
    0BUF A0 BUFC,    \ signature byte
    DUP 4 + BUF,     \ length
    SEG-INX @ BUFC,
    MEMP@    BUF,
    (WR.REC)         \ write header only
    MEMP@            \ -- len addr
    OVER 1+          \ -- len addr len+chk
    REC-FBUF @       \ -- len addr len+chk fbuf
    WRITE            \ -- len
    MEMP +! ;            -->

Screen 22
\ SAVE-OBJ: RECORD TYPES: LEDATA                    RA 03JAN93
HEX
: WR.PATCH.TARGET
    A0 REC-SIGNATURE,
    SEG-INX @ BUFC,     \ segment (by index)
    MEMP@     BUF,      \ offset in segment
    0 BUF, 0 BUF,           \ dummy target
WR.REC
    4 MEMP   +!  ;       -->
\ WR.LEDATA writes a 1K or smaller chunk of code without any
\ fixups in it.  WR.PATCH.TARGET is for those 4-byte sections
\ of code that require a fixup to a segment:offset of a long
\ call to another object module.  This is the simplest way to do
\ it.  The word that writes the code to the .OBJ file just steps
\ through the memory image writing the two kinds of records as
\ it comes to them.

Screen 23
\ SAVE-OBJ: RECORD TYPES: FIXUPP                    RA 02JAN93
HEX
: WR.FIXUPP
    9C REC-SIGNATURE,
```

ence, the linker name of the C function we want to call when we execute GETPIXEL. Note the leading underscore. The actual C function is `getpixel()`, but the C compiler adds the underscore when it creates the object module containing the code for the function, so it is as `_getpixel` that the linker recognizes it. Other languages may mangle their function, procedure, or subroutine names in different ways (C++ is particularly elaborate) which you'll want to know about if you plan to link with them.

According to my experiments, I can access most any C function with this technique. There are some restrictions. Some functions in the Borland library reference aren't really functions but macros. For example, you can't use Borland's `random()`, which is a macro, but you can use `rand()`, which is a function. Other functions require examination to use correctly. Some list parameters which have symbolic names. These are, of course, really numbers, usually listed in header files. Others are initially surprising or perhaps peculiar. For example, we can define a call to `printf()` as

```
0 0 EXTERN PRINTF
_printf
```

`printf()` doesn't return anything (which accounts for the first zero above) and it takes an indeterminate number of parameters (which accounts for the second zero), which causes all input parameters to be left on the stack for us to clean up afterwards. This is the best we can do. The parameters to `printf()` are a pointer to a formatting string and zero or more arguments. The formatting string is a standard C null-terminated string with embedded escape sequences (such as \n, which indicates a new

line) and format specifiers (such as %d, which says to print the top stack word as a decimal integer). When printf() executes, it prints the string, acting on the escape sequences and replacing the format specifiers with values from the argument list. If you're thinking of linking Forth and C you already know this, and know that the fragment

```
int x;
...
x = 1234;
printf("\nHere's an integer
value:\nx=%d.",x);
```

will print

```
Here's an integer value:
x=1234
```

We can go partway with this in Forth. The following sequence

```
: PRINT-INT
  Z" \nHere's an integer value:\nx=%d."
  PRINTF ;
1234 PRINT-INT
```

will yield

```
\nHere's an integer value:\nx=1234.
```

at least with Borland C. This shouldn't have been surprising: since the escape sequences are unambiguous at compile time the C compiler can process them before writing the OBJ file, but the format specifications can only be replaced by strings representing the appropriate values at run time.

For further examples of C function calls, see the sample code. One is particularly important: the call to the C function exit(), defined as

```
0 1 EXTERN C-EXIT
_exit.
```

Because a C program sets up certain parameters, captures various interrupts, etc., when it starts up, and we really don't want to know what those are, we need to exit from Forth in a way that will cause C to clean up after itself as well. The way to do this in C is either to exit through the bottom of the main() function or to call exit() explicitly with a parameter that tells DOS what the exit condition was (zero means "good," anything else means "bad"). So instead of BYE we can execute GOOD-BYE defined as

```
: GOOD-BYE 0 C-EXIT ;
```

and avoid having our machine lock up on us unexpectedly while we're in the middle of something else later.

Portability issues aside, the actual coding of EXTERN and, later, SAVE-OBJ and SAVE-EXE is also necessarily somewhat implementation (and personal preference) dependent. I chose an 83-standard, indirect-threaded Forth for this project because it is probably still the most widely available model, and I used Guy Kelly's implementation because it's the single-segment version I'm most familiar with. I could have used F83 with perhaps no changes not already mentioned except the need to use an unfamiliar editor. More serious adjustments would have to be made for direct-threading or subroutine-threading (one hesitates to contemplate token-threading), not to mention other standards, including the forthcoming ANS.

The code works and, working, should speak for itself, but here's a little more detail:

The defining word EXTERN has two parts. CODE-EXTERN creates Forth code words by compiling bytes in memory with C,. This is not a job for CREATE...DOES> since each external reference needs its own code body. The only unusual things

```
\ no thread field
\ locat
    CC BUFC,    \ M=1, loc = segment:offset
     0 BUFC,    \ offset in record, in 4Z.LEDATA always 0
\ fixdat
    56 BUFC,    \ F=0, frame det by target, T=0, ext index
    XREF# @ BUFC,
    WR.REC  ;
-->
\ For details of the FIXUPP record try the MS-DOS ENCYCLOPEDIA.

Screen 24
\ SAVE-OBJ: WR.CODE                          01JAN93 RA 29DEC92
HEX  DEFER 'WR.SEG   DEFER 'WR.FIX
: WR.FIX   WR.PATCH.TARGET WR.FIXUPP  ;
: WR1CODEREC            \ step through the memory image
    CUR-XLINK @ ?DUP    \ and the EXTERN linked list
    IF XLINK>FIXADR MEMP@ - ?DUP
        IF 400 UMIN 'WR.SEG ELSE 'WR.FIX >NXT-XLINK THEN
    ELSE TOP.ADDR MEMP@ - 400 UMIN 'WR.SEG
    THEN ;
: (WR.CODE)
    0 MEMP !  >1ST-XLINK
    BEGIN  WR1CODEREC  MEMP@ TOP.ADDR U< 0= UNTIL ;
: WR.CODE 1 SEG-INX !
    ['] WR.LEDATA IS 'WR.SEG
    ['] WR.FIX    IS 'WR.FIX
```

```
    (WR.CODE) ; -->


Screen 25
\ SAVE-OBJ: WR.OBJ.RECORDS                  03JAN93 RA 29DEC92
HEX
: WR.OBJ.RECORDS
    WR.THEADR
    WR.LNAMES
    3 2   0 62 WR.SEGDEF
    3 2 200 60 WR.SEGDEF     \ a little room for Forth msgs
    1ST-XLINK @ IF WR.EXTDEF THEN
    WR.PUBDEF
    WR.CODE
    WR.MODEND ; -->

Screen 26
\ SAVE-OBJ  SAVE FORTH AS .OBJ FILE                 RA 29DEC92
HEX
: SAVE-OBJ
    PAD  100 + RECBUF !              \ set buffer address
    OFBUF REC-FBUF !
    REC-FBUF @ FILENAME FORTH.OBJ
    REC-FBUF @ MAKE                  \ create file
    REC-FBUF @ DUP FREOPEN FYL !     \ open & make current
    EMPTY-BUFFERS FLUSH
    SET-BOOT GET-MSGS
    EMPTY-BUFFERS FLUSH
    WR.OBJ.RECORDS
    REC-FBUF @ FCLOSE                \ flush it to be sure
    CR ." OBJECT FILE WRITTEN "   ;
DECIMAL

Screen 27
\ SAVE-EXE: NAVIGATING THE ENVIRONMENT          RA 16JAN93

: 0SKIPL            ( seg addr --- seg addr' )
    \ skip past end of zstring at seg:addr
    BEGIN 2DUP C@L WHILE 1+ REPEAT 1+ ;

: 0LENGTHL          ( seg addr --- len )
    \ get length of zstring at seg:addr
    2DUP 0SKIPL         \ seg addr seg addr'
    NIP SWAP -  NIP ;   \ len

: 00SKIPL           ( seg addr --- seg addr' )
    \ skip past double-zero at end of environment variables
    BEGIN 2DUP @L WHILE 1+ REPEAT 2+ ;
-->

Screen 28
\ SAVE-EXE: FIND-PROGNAME !FORTH-OFFSET 18JAN93 RA 16JAN93
DECIMAL
2VARIABLE FORTH-IMAGE-OFFSET
: FIND-PROGNAME     ( --- addr zlen )
    ENV@ 0 00SKIPL      \ seg name-addr
    0SKIPL              \ get past sring count
    2DUP 0LENGTHL       \ seg name-addr zlen
```

about this code body are (1) that it is laid down in memory before, rather than after, its header (which makes finding other things in the header easier), and (2) that it contains a long call to segment:offset 0000:0000, which won't work very well if the word is executed before it's linked.

While it is laying down code, CODE-EXTERN consumes the two stack parameters already mentioned and saves two addresses on the stack: the address of the beginning of the code word and the address of the null long call. The second part of EXTERN, CREATE-EXTERN, creates an extended header for the word being defined. The word HEADER in Kelly's Forth is a factor of CREATE, and does everything CREATE does except lay down the CFA. It can be replaced by CREATE -2 ALLOT for portability. After the header is the CFA which, for a code word in an indirect-threaded Forth, contains the address of the executable code. Normally this code immediately follows the CFA, so the CFA can be compiled by the phrase HERE 2+ ,. For these EXTERNS, however, the executable code has already been laid down. Its address is on top of the stack at this point, and can be compiled by , (comma). After the CFA, there are two more fields: the patch address passed to CREATE-EXTERN by CODE-EXTERN, and the linker name for the EXTDEF record of the OBJ file, which is parsed out of the input stream by WORD.

The linking of the list of EXTERNs is straightforward. Variables point to the first and last elements in the list, each element in the list points to the next, and each time a new element is added LAST-XLINK and a pointer in the previously last EXTERN are adjusted by EXT-LINK,. What makes this list different from most others

in Forth (the dictionary, the list of vocabularies) is that the links point forward rather than backward, which is what we want, since this is the order we'll need them as we write the records of the OBJ file. The words XLINK>FIXADR and XLINK>XNAME, factored to make redesign easier, will be used to find the patch address field and the linker name field from the external link field in each header, and >1ST-XLINK and >NXT-XLINK manage stepping through the list.

In memory, EXTERN lays down something like this:

```
| long call code | next, |
| name | link | cfa | patch | fwd link | linker-name |
```

image of the linked EXE file. Strings in object records are the kind we're used to, beginning with a count byte rather than ending with a null.

Each record type does one thing well enough. We need eight different record types and, since they tend to refer to each other, order for the most part matters. I'll describe each as briefly as possible, in the order we need to use them. For further information, or to puzzle it out yourself, consult the *MS-DOS Encyclopedia*. For clarification, consult the code.

The THEADR or translator header record (I'm using names given from the *MS-DOS Encyclopedia*) just gives the module a name, and must be first. It's almost the simplest

### 4. Object File Records

After we create references to external functions, we have to save the running Forth image as an OBJ file. Most systems have a SAVE-FORTH word which saves the Forth image as an executable file, either a COM file or a workable facsimile of an EXE without fix-ups. What we need is a bit more complicated but, in the end, more tedious than difficult.

The record types for OBJ files are described completely, if not quite lucidly, in the *MS-DOS Encyclopedia*, though I must add that the formats of these records are twisted enough that they perhaps cannot be described clearly. The important thing for us is that there are things about them we do not need to know, and some that we do.

All OBJ records begin with a one-byte signature and a two-byte length, and end with a checksum byte. The Borland and Microsoft linkers, at least the versions I've used, ignore the checksum, so we don't have to care what's in it. The length is the length of the record after the length—in other words, three less than the length of the whole record in bytes. The contents of object records—between the count and the checksum—are strings, tightly packed bit-field bytes, index bytes (pointing at or into other records), and (for two record types) data destined to compose the memory

```
    CS@ HERE ROT DUP >R CMOVEL HERE R> ;

: HEADERPARS@    ( --- n )      0 BLOCK 8 + @ ;
: !FORTH-OFFSET  ( --- dOffset )
    HEADERPARS@ 16 UM*  \ offset of code portion in EXE file
    CS@ PSP@ -  16 UM*  \ offset of Forth from PSP in memory
    256. D-             \ subtract psp size
    D+ FORTH-IMAGE-OFFSET 2! ;
-->

Screen 29
\ SAVE-EXE: WRITE-FORTH-IMAGE   SAVE-EXE         RA 20JAN93
: WRITE-CHUNK ( len --- ) MEMP@ SWAP DUP MEMP +! OFBUF WRITE ;
: SKIP-FIXUP  ( --- )      \ +SEEK would be useful here
    4 MEMP +! FORTH-IMAGE-OFFSET 2@ MEMP@ 0 D+ OFBUF SEEK ;
: WRITE-FORTH-IMAGE
    !FORTH-OFFSET   FORTH-IMAGE-OFFSET 2@ OFBUF SEEK
    ['] WRITE-CHUNK IS 'WR.SEG
    ['] SKIP-FIXUP  IS 'WR.FIX
    (WR.CODE) ;
: OPEN-PROGFILE
    FIND-PROGNAME   OFBUF Z>BUF OFBUF FREOPEN OFBUF FYL ! ;
: SAVE-EXE
    OPEN-PROGFILE   SET-BOOT GET-MSGS
    WRITE-FORTH-IMAGE   OFBUF FCLOSE ;          -->

Screen 30
\ EXTERN TEST WORDS
0 1 EXTERN C-EXIT _exit      \ returns code to DOS
: GOOD-BYE 0 C-EXIT ;        \ use C's cleanup
2 0 EXTERN CORELEFT _coreleft ( --- dPars )
1 2 EXTERN SETBLOCK _setblock
1 1 EXTERN MALLOC   _malloc
0 1 EXTERN FREE   _free                          -->

Screen 31
\ EXTERN TEST WORDS
1 2 EXTERN GETPIXEL _getpixel   0 3 EXTERN PUTPIXEL _putpixel
0 6 EXTERN GR.INIT _initgraph   0 0 EXTERN GCLOSE _closegraph
VARIABLE GMODE
VARIABLE GDRIVER

: (GINIT)
    CS@ Z" \BORLANDC\BGI " CS@ GMODE CS@ GDRIVER GR.INIT ;
```

```
: (VGAHI) 9 GDRIVER ! 2 GMODE ! ;
: (CGAC3) 1 GDRIVER ! 3 GMODE ! ;
: CGA-INIT (CGAC3) (GINIT) ;
: VGA-INIT (VGAHI) (GINIT) ;
DECIMAL

: GTEST CGA-INIT
    100 30 DO
    100 30 DO I J + 8 MOD I J PUTPIXEL LOOP
    LOOP ;                                              -->

Screen 32
\ EXTERN TEST WORDS: PRINTF                          RA 05FEB93
0 0 EXTERN PRINTF _printf   \ we'll have to clean up stack

: TEST-STRING
    CS@
    Z" HERE'S A PRINTSTRING TEST " ; \ seg addr

: TEST-PRINTSTRING TEST-STRING PRINTF ;
: TS2   CS@ Z" \nHERE'S A NEWLINE OR NOT" ;
: TS3   CS@ Z" HERE'S AN INT %d OKAY? " ;
: TEST2       TS2 PRINTF ;
: TEST3 12345 TS3 PRINTF ;
-->

\n not handled but %d is

Screen 33
\ EXTERN TEST WORDS: CGA GRAPHICS
DECIMAL
0 2 EXTERN LINETO   _lineto
0 1 EXTERN SETCOLOR _setcolor
1 0 EXTERN RAND     _rand
: GTEST2
    CGA-INIT
    1000 0 DO
        RAND 200 MOD RAND 300 MOD LINETO
        RAND 4 MOD SETCOLOR
    LOOP
    KEY DROP GCLOSE ;
-->

Screen 34
\ MORE .OBJ GRAPHICS WORDS
DECIMAL
\ 0 0 EXTERN VGAINIT _vgainit

: GTEST3
    VGA-INIT
    1000 0 DO
    RAND 480 MOD RAND 640 MOD LINETO
    RAND 16 MOD SETCOLOR
    LOOP
    KEY DROP GCLOSE ;
```

possible: between the count and the checksum, it contains only a counted string, in our case "FORTH." The linker uses this name in error messages.

The LNAMES record contains a list of group and/or segment names other records will refer to by number, counting the first as one, not zero. For us, only SEGDEF records refer to this list. Imitating other object modules I've examined, I've included, as the first name in the list, a blank name of length zero. This isn't really necessary: using an index of zero seems to have the same effect as using an index of one to point to this zero-length string. The other two names will be used for the segment name and the class name of our single Forth code segment. (But see the "More Implementation Details" section, later.)

The SEGDEF record defines a memory segment. It contains a number of fields: an "ACBP byte," a two-byte segment length, and three name index bytes which contain references to names in the previous LNAMES record.

The ACPB byte encodes various attributes of the segment. Ours is 62H, or 011 000 1 0. The three highest bits indicate the segment alignment, which in this case is relocatable and paragraph aligned (i.e., it will begin on a 16-byte boundary). The paragraph alignment is important, since we want to set the segment registers to the beginning of the Forth address space. The next three bits are the combine type, which in this case indicates that the segment cannot be combined. This is the safest value for this field, though 010, which would allow concatenation with another segment of the same name, would work as well. The penultimate or "B" (for

"big") bit combines with the following two bytes to give the length of the segment. In our case, the segment is 10000 hex bytes, or 64K, supposedly the maximum segment length, and is what we want. The last bit is the "Page resident" flag, which the *MS-DOS Encyclopedia* says is unused and should always be zero.

```
Screen 35
\ PRINT FIXUP TABLE FROM EXE                          RA 15FEB93
HEX
VARIABLE TABLE-OFFSET
VARIABLE TABLE-ENTRIES
2VARIABLE TABLE-ENTRY
: PT-FIX-TABLE
    OPEN-PROGFILE
    0000.0006 OFBUF SEEK    TABLE-ENTRIES 2 OFBUF READ
    0000.0018 OFBUF SEEK    TABLE-OFFSET  2 OFBUF READ
    TABLE-OFFSET @ 0 OFBUF SEEK
    CR
    TABLE-ENTRIES @ 0 DO
        TABLE-ENTRY 4 OFBUF READ
        TABLE-ENTRY 2@ 4H.R SPACE 4H.R
        I 1+ 8 MOD 0= IF CR ELSE SPACE THEN
    LOOP ;
```

The three bytes after the length bytes are indexes into the list in the LNAMES record to assign segment, class, and overlay names to the segment. Actually, the overlay index is ignored, so pointing it at a blank string as I have done is superfluous. The other names appear in MAP files and help control segment ordering and combination. Unless its normal operation is overridden with command line switches, LINK will combine segments with the same segment and class name, and concatenate segments with the same class name, in the order in which the linker encounters them. This order is controlled, within an OBJ file, by the order of their declarations within it and, among object files, by the order in which they're listed as parameters to LINK on the command line or in the makefile. In our case, our CODE segment will be concatenated with other CODE segments from other object modules, but it won't combine with anything that is not also called "FORTH_TEXT".

WR.SEGDEF, which writes this record, takes its parameters (name indexes, length, and ACBP byte) from the stack, to make it easier to define multiple segments.

The EXTDEF record contains a list of external references, names of functions defined elsewhere that will be called from Forth: in other words, an *imports* list. We want to put the linker names here from the words we defined with EXTERN. The linker will use these names to resolve our patch addresses with the addresses of C functions with the same linker names. Each name in the EXTDEF record is followed by a type index byte but, since we have no TYPDEF records, these bytes are all just set to zero.

The PUBDEF record contains the names of symbols *exported* by the OBJ file. In this case we have only one, _forth, which will be used to resolve the calling address of a C function called forth(). Before the name field in the PUBDEF record are a group index and a segment index, and after the name are an offset into the indicated segment and a type index. We have no GRPDEF or TYPDEF records, and don't need them, so these bytes are set to zero. The segment index is set to one for the first segment and, since this Forth began life as a COM file, the offset into the segment is set to (hex) 100. Using this information the linker can link _forth to a C function called forth() so that calling that function will execute our familiar Forth image, almost as though it were still a COM file.

(You will notice that by making C call Forth we are, in some small way, giving precedence to C. We could, in fact, put the symbol _main rather than _forth in this PUBDEF record and let Forth start immediately when the EXE file is loaded. The problem with this is it doesn't allow C to perform its initalization correctly, and some C functions don't quite work. The fact that it makes the MODEND record more difficult to construct is only a minor inconvenience.)

The LEDATA ("logical enumerated data") and FIXUPP records occur together. The LEDATA records contain the actual code, the contents of the Forth image in memory at the time they are written out. (This is where our activity most differs from that of a C or other language compiler: we're writing code out of memory, code that for the most part is actually running while it's being written, rather that merely translating text from a source code file.) The maximum amount of object code one record can contain is 1K, so we could write it out as 64 1K blocks if it weren't for the patch addresses we want the linker to resolve. Simplicity is a practical virtue: the method I've adopted is to write two kinds of LEDATA records: pure object code records, which can be up to 1K in length and have nothing to resolve, and pure *fix-up target* records, which are always four bytes long and all zeros.

Each fix-up target LEDATA record needs to be followed immediately by a FIXUPP record. The FIXUPP record type has twelve pages in the *MS-DOS Encyclopedia*, with a fairly bewildering variety of options. We need only one.

FIXUPP records can contain thread fields and fix-up fields. We don't need a thread field. The four bytes between the signature and the checksum in our FIXUPP record make a single fix-up field. It's identified as such by having the high bit set in the first byte. The other bits in that byte and the next indicate that the fix-up field is segment-relative and refers to a segment:offset at offset zero of the LEDATA record it

follows. The other two bytes are called the "fixdat" field. The bits in the first indicate that the "frame" is determined by the target and that the target is specified by an external index. Trust me: this is what we want. What this comes to so far is the three bytes CC 00 56. The last byte before the fake checksum is an index into the list of external references in the previous EXTDEF record, indicating which external function is to be resolved to this patch address.

The MODEND record is the simplest of all: since the Forth object module isn't going to be the "main" program module (it won't contain the entry point for the EXE file), it only marks an end, and is just one byte of zero between the length and the checksum.

Most of these object records get constructed in memory before being written to disk. SAVE-OBJ sets up this buffer, and creates and opens a DOS file to put the records in. Almost everything in this word is implementation dependent, and already explained. (SET-BOOT and GET-MSGS are implementation peculiarites that will be discussed later.) I've defined a set of words to compile bytes, words, and strings into this buffer. In some implementations it might be possible to construct these records at HERE by saving and restoring the DP each time: in that case, you could just use , (comma), C, (c-comma), and (if you have it) ," (comma-quote) instead.

With these core words, I've defined words which compile the record signature, (fake) checksum, and length into the record and then actually write it out. These words make writing most records easy and uniform. The exceptions are the EXTDEF record, which contains the linker names of the external references defined by EXTERN, and the LEDATA and FIXUPP records, which contain the actual Forth-image machine code.

The variables CUR-XLINK (current external link), MEMP (memory pointer), and XREF# (external reference number) are used to keep track of what's been written from the Forth memory image to the OBJ file. The words >1ST-XLINK and >NXT-XLINK step CUR-XLINK through the linked list of words defined with EXTERN, and keep a count in XREF#. CUR-XLINK always points to the next external reference that needs to be written, or to zero to indicate that none are left. WR.EXTDEF uses these words to find the linker names, and WR.CODE uses them to find the patch addresses.

The EXTDEF record is composed in the same buffer as the other records. WR.CODE works a little differently. Except for the signature, length, and fake checksum of each LEDATA record WR.CODE writes memory directly to the file.

When WR.CODE begins, MEMP is set to zero, CUR-XLINK is set to the contents of 1ST-XLINK (the first EXTERN), and XREF# is set to one, the index of the linker name associated with the first external reference in the EXTDEF record. (The variable SEG-INX is also set to one, to point at the first SEGDEF record, since this is the one associated with the memory segment we're writing out to the OBJ file.) Then WR.CODE executes WR1CODEREC in a loop.

During each pass through the loop, CUR-XLINK either points at a link, or it does not (indicated by a value of zero). If not, WR1CODEREC writes a record containing a maximum 1K from memory (it may be less if it's the last record), starting with the address pointed to by MEMP. Before it writes the record, WR1CODEREC knows its length so it can write the first three bytes of the record, then the 1K or smaller body, then the fake checksum. (This allows the record buffer to be relatively small, even though many records will be 1028 bytes.)

If, on the other hand, there are EXTERNs left to write, MEMP either points to the patch address of the next one or it does not. If MEMP has reached a patch address, WR1CODEREC writes a four-byte LEDATA record filled with zeros followed by a FIXUPP record, then calls >NXT-XLINK to advance CUR-XLINK and increment XREF#; otherwise WR1CODEREC writes a record similar to the one described above, 1K or less from memory, starting from MEMP, and stopping at or before the next patch address.

The format of the FIXUPP record has already been described: the important thing here is that the value in XREF# is included, which tells the linker which name in the EXTDEF record refers to this particular fix-up so it can patch its address correctly. In all cases, WR1CODEREC advances MEMP to point to the first byte of memory not yet written and, so long is there is memory left to write, the loop repeats.

### 5. Linking Forth and C

It is possible to write out a Forth image, without any EXTERNs, as an OBJ file and use LINK to create an EXE. This is the simplest situation and, though interesting, not very useful (except as a step in the development of the method).

To link in C functions, we have to have a `main()` function and link in the initializaton module and libraries for the huge model. We can make Forth the main module by calling it `_main` in the THEADR object record but, as mentioned earlier, this doesn't quite work. In order for C to do its initialization correctly, `main()` has to be a C function, however trivial. The file FORTHC.C is just:

```
void far forth();
main()
{
     forth();
}
```

This is all the C we need. We declare `forth()` as a far function—that is, one that is called with a long call—and then make the call by invoking the function.

But C not only has some initialization to do before calling `main()`, it also wants to clean up after itself. Since Forth changes all the segment registers, including SS, to point to itself—losing, in the process, the return address left on the stack—the function `forth()` will never return; therefore, to be really safe, we need to redefine BYE as a call to the C function `exit()`. This is redundant, but it's important.

After we write the OBJ file, all that remains is to call LINK (or TLINK). I won't go into detail about this except to note

that our object file, FORTH.OBJ or whatever, should be put in the TLINK command line just like a C object file, after the initialization module (C0H.OBJ or C0FH.OBJ for TLINK). If you want to link Forth and C, you already know how to use LINK and probably MAKE, and if you don't there are far better explanations, even in the Borland and Microsoft compiler manuals, than I can give here; but just as an example, here's the makefile I used:

```
FORTH.EXE: FORTH.OBJ GINIT.OBJ FORTHC.OBJ
TLINK /d/m/s/Tde C0fH FORTHC FORTH CGA
EGAVGA GINIT,
   FORTHC,,CH.LIB GRAPHICS.LIB
FORTHC.OBJ: FORTHC.C
   BCC -mh -c FORTHC.C
GINIT.OBJ: GINIT.C
   BCC -mh -c GINIT.C
```

### 6. Saving Without Re-linking

Once we have C routines linked into FORTH.EXE, we might want to go away from the C environment and still be able to add words and save the image. It would be inconvenient to have to use LINK in these circumstances and, if we're careful, we don't have to. What we do have to do is figure out where in FORTH.EXE the Forth image is and write the new image right back there.

There are three steps to this: finding the EXE file on disk, finding the location of the Forth image in the EXE file, and writing the Forth image from memory back into the EXE file without destroying the fix-up records.

To find, on disk, the EXE file of the program that is actually running, we need to find what the DOS books call its "fully qualified filename." The string representing this filename, which includes the "canonical path" (drive and path from the root directory, with any re-directions imposed by APPEND resolved), can be found at the end of the running program's environment block.

We can find the program segment prefix (PSP) of the running program through a DOS call, int 21h function 51h (or 62h). At offset 2Ch of the PSP is the segment address of the environment. The environment consists of a series of null-terminated strings terminated (depending on how you look at it) by a zero-length string or an extra null. Right after the environment is another set of strings. First there is a count of these strings (which may always be one), followed by a null, then by the string representing the "fully-qualified filename" we're looking for. No matter what we've renamed it, no matter where it is on our path or how we've called it, this string can be used to locate the file and open it. FIND-PROGNAME finds this string, moves it—with its null-terminator—to HERE, and leaves its address and count on the stack.

The next thing we want to do is open this file for reading and writing. We can do this because, although we are running the program contained in the file, the file is not now open. (In a multi-user or multi-tasking system, someone *could* come along and delete, rename, or remove the file before we get to it.) The word OPEN-PROGFILE does all this. The phrase

```
OFBUF Z>BUF OFBUF FREOPEN OFBUF FYL !
```

which is highly implementation dependent, moves the filename to the FBUF OFBUF, opens the file, and makes it current, so that BLOCK will refer to blocks in this file. Your technique may vary. The important thing is to open the EXE file so we can read its header.

There's lots of possibly interesting information in the header, but really all we need to know is how big it is so we can find the rest of the file. The number of 16-byte paragraphs in the header is stored in the word at byte offset 8 in the header. We also know the address, in memory, of the PSP for our program and the segment address (from CS@) of the Forth segment. Since the PSP is 256 (decimal) bytes long, and resides just before the image of the EXE program in memory, and since we know and trust that the code portion of the EXE file is the same as the memory image less the fix-ups to specific addresses in it, finding the Forth portion of the EXE file is just arithmetic. !FORTH-OFFSET performs this arithmetic and stores the result in the variable FORTH-IMAGE-OFFSET.

After we've found this offset, we use SEEK to get to the right place in the file, and then write the memory image in much the same way we wrote LEDATA records. The operations are similar enough that I've used the same loop, vectoring the action words that handle the code and fix-up parts of the operation; the differences are that the pieces of memory we write go directly into the EXE file instead of into OBJ file records and, instead of writing four-byte fix-up targets, we just carefully skip their places in the file.

(The reason we have to be careful about these fix-ups is that they no longer contain zeros, either in memory or in the EXE file. Further, their contents in memory and in the file are different. In the EXE file, these locations contain segment:offset addresses relative to the beginning of the executable portion of the file, while in memory these addresses are fixed up to actual addresses.)

So now we have two ways to save Forth: when we've added new external references with EXTERN we save it as an OBJ file and use LINK, including re-linking all the things we've linked before; when we have no new references to link, we can just save the image into the current EXE file. (A simple extension would be a word which would check the fix-up list to see whether we have to use SAVE-OBJ or can get by with SAVE-EXE.) There are two small problems with SAVE-EXE: one is the checksum in the EXE header, which doesn't get updated by this technique. But, like the checksums in the OBJ records, this one doesn't seem to matter except, perhaps, to virus-checking programs. The second problem is that you can't use LZEXE or EXEPACK or any other EXE-file-compression programs on the EXE file and hope to save the Forth image back into it.

### 7. More Implementation Details

A couple of implementation peculiarities of Kelly's Forth

are encapsulated in the words SET-BOOT and GET-MSGS. The boot-up values of LATEST, FENCE, DP, and VOC-LINK are stored in low Forth memory in this implementation. If they are not set before the image is saved, nothing compiled since the previous save will be available when the file is reloaded. The words will be in memory but nothing will point to them, so they won't be found. This can be a useful feature: you can, for example, execute SET-BOOT before you compile the code to save the system, which will then effectively be discarded by the save. However, for what I like to think of as simplicty and clarity, I have chosen to include SET-BOOT in SAVE-OBJ and SAVE-EXE.

The other implementation peculiarity, GET-MSGS, is more interesting in the current discussion because of the problem it caused and the solution I found. In Kelly's FORTH.COM, the system messages are in the FIRST block buffer. As part of the boot-up process, the contents of this block buffer (actually, the first half of it) are moved to just beyond the 64K boundary, from where they are accessed by CMOVEL when needed. Consequently, before the Forth image is saved, whether by SAVE-FORTH, SAVE-OBJ, or SAVE-EXE, these messages have to be moved back into the FIRST block buffer or they will be lost. (With SAVE-EXE, they will be lost only indirectly, since they remain in the file but will be over-written by whatever happens to be in the FIRST block buffer when the system boots.) This is the function of GET-MSGS. A consequence of this location for the messages is that, in order to have them, I had to define a segment to keep them in. In WR.OBJ.RECORDS, the second invocation of WR.SEGDEF defines a 0.5K segment for the messages with—and this is important—the same class and segment names as the main Forth segment. The size is different, and the "B" bit of the ACBP byte is turned off, but since these two segments have the same pair of names, and since their ACPB bytes indicate that they cannot be combined, the linker will see that they are concatenated, in the order in which they are declared. This is just what I needed and it suggests that if you wanted, for example, a four-segment Forth, with separate 64K segments for code, lists, data, and headers, you could define their segments in this kind of way. Note that no LEDATA records (or LIDATA records, a type not discussed here) are written for this segment: it just reserves space for Forth to put its system messages.

### Conclusion

I hope this relatively simple recipe for accessing external functions and libraries from Forth will prove helpful to those who feel isolated in our high-performance ghetto, or who feel they have to work in some less attractive language just to get work done. Its value for some may be as a method for gluing together other things in a way that provides familiar control and semi-interactive development and testing. But it would be foolish for me to guess the uses of this tool: if it is a good one, it will find homes. My interest has, so far, mainly been in learning how to do it, rather than finding something to do with it. I feel confident that the techniques I've used—writing main() in C, using exit() for BYE, using huge functions to isolate data segment references—have produced a sound method of linking C and Forth in a way that is less annoying than might have been expected.

Extensions to this method might include (1) creating a segmented linkable Forth with, for example, data in a segment addressed by DS, code referenced from CS, lists from ES, and a stack segment big enough to handle the hungriest set of C functions; (2) object linking to other, more interesting, languages such as Prolog and Lisp; (3) linking to "extended" or "new exe" format files such as those used by Windows and OS/2. Have fun.

### Bibliography


Ray Duncan, ed., *The MS-DOS Encyclopedia*, Microsoft Press, 1988.

Ray Duncan, *Advanced MS-DOS Programming*, Microsoft Press, 1986.

Mark Ludwig, *The Little Black Book of Computer Viruses*, American Eagle Publications, 1991. I am indebted to this book for its discussion of the structure of EXE files, particulary the function of the "relocation index table."

*Turbo Assembler 3.0 User's Guide*, Borland International, 1991.


---


Richard Astle has been programming in Forth for about eight years, most of that time developing and maintaining a rather large database management set of programs. In the process, he has re-written the underlying Forth system more than once for speed and capacity. He has a bachelor's degree in mathematics from Stanford University, a master's in creative writing from San Francisco State, and a Ph.D. in English literature from the University of California (San Diego).

# Where Do You Go From Here?

*C.H. Ting*
*San Mateo, California*

This series of tutorials has only scratched the surface of Forth and F-PC. Its goal is to expose you to a very minimum set of Forth instructions so that you can start to use Forth to solve practical problems in the shortest possible time. What is intentionally neglected is why and how Forth works, as a language and as an operating system, in addition to a host of topics about various features of Forth and F-PC.

There are different directions you may proceed from here, depending upon your needs and interests:

1. For the practical engineer, the next logical step is to work on *The Forth Course* by Dr. Richard Haskell. In fact, this tutorial was developed to complement *The Forth Course*, which skims too fast over the elementary Forth instructions and dives too quickly into the advanced topics of an upper-level college microcomputer laboratory. *The Forth Course* is available from Offete Enterprises for $25.

2. If you are more interested in Forth as a programming language and want to learn more about the language aspects of Forth, you will enjoy Leo Brodie's books on Forth:
*Starting Forth*, Prentice Hall, 1982
*Thinking Forth*, Prentice Hall, 1984

These two books are classics in the Forth literature and are pleasant reading. The only problem is that some of the code examples might not work in F-PC without modification, because of the difference between the block-based source code in the books and the file-based source code in F-PC.

3. If you are interested in using F-PC and getting the most out of your PC for a specific project, you need to know more about the structures and the special features of F-PC, such as the DOS interface, file-access mechanism, color control on the display, hardware input and output, interrupts, etc. For this, you need a complete F-PC system for exploration. My following manuals can be of great help:
*F-PC User's Manual*, Offete Enterprises, 1989, $20
*F-PC Technical Reference Manual*, Offete Enterprises, 1989, $30
F-PC System Disk Set, Offete Enterprises, $25
F-PC User Contributions Disk Set, Offete Enterprises, $25

The F-PC materials are also available from:
Forth Interest Group
P.O. Box 2154
Oakland, CA 95621
510-89-FORTH

The address of Offete Enterprises is:
1306 South B Street
San Mateo CA 94402
415-574-8250

4. For commercial and professional applications, you may want to consider buying a commercial Forth system which is supported by a real software company. You will get documentation, and you can get help when in trouble. A few of the commercial Forth vendors are:

Forth, Inc.
111 N. Sepulveda Blvd.
Manhattan Beach, California 90266
213-372-8493

Laboratory Microsystems, Inc.
P.O. Box 10430
Marina del Rey, California 90295
213-306-7412

Miller Microcomputer Services
61 Lake Shore Road
Natick, Massachusetts 01760
617-653-6136

*[For easy access to many of the materials listed here, refer to the mail order form included as the centerfold in this issue of* Forth Dimensions. *Other vendors of Forth products may be located via their advertisements in our pages.—Ed.]*

---

Dr. C.H. Ting is a noted Forth authority who has made many significant contributions to Forth and the Forth Interest Group.

# Drawing BMP Files

*Hank Wilkinson*
*Greensboro, North Carolina*

Microsoft provides with Windows the useful drawing program Paintbrush. Paintbrush drawings may be directly printed, inserted, or otherwise linked to other documents. Forth pixel drawing performed in memory and stored in a file may be handled as if it were a Windows Paintbrush file. Simplifying assumptions make this process easy enough to conceive, do, and (perhaps) explain.

One use of computer-generated drawing is graphing mathematical results. Dr. J.V. Noble's *Scientific Forth: A Modern Language For Scientific Computing* (Mechum Banks Publishing, 1992) provides a college/graduate-level text in numerical methods using Forth. Though not up to the quality of Dr. Noble's text, rudimentary methods discussed below expose a foundation for graphing into Windows Paintbrush files.

Programming is the process of discovering and communicating details necessary for desired functionality. Forth routines used to discover the details presented here are not shown. My general goal is to use Forth "in" Windows. Black-and-white drawing into a file represents only a small increment toward this goal.

Black-and-white pixel drawing simplifies the general case of Paintbrush files, while allowing enough "drawing" to be of practical use. A blank drawing, created and saved using Paintbrush, serves as the starting point. When Paintbrush saves an image on disk, it tries to give the filename an extension of .BMP.

These .BMP files contain a 62-byte header consisting of attributes describing the image. Loading an image restores the attributes from its header. For this article we use black-and-white images that are 640 pixels wide and 880 pixels tall. Except to copy, we ignore the header.

The image directly follows the header, with its rows ordered sequentially from 879 to zero (i.e., the rows are in reverse order). Conceptually, the image is the set of (x,y) points ranging from (0,0) to (639,879). One bit represents one pixel for black-and-white images.

When the pixel value equals zero, the pixel is shown as black, on both the printed page and the screen. With the bit equal to one, the pixel is displayed as white on the screen and is not printed on the page. For completeness, we represent below a sheet of paper with a black-and-white "image," dimensioned 640 pixels wide and 880 pixels tall.

Rudimentary drawing into Windows Paintbrush files requires the following functions:
a) Loading file into memory
b) Drawing a pixel
c) Saving image to disk

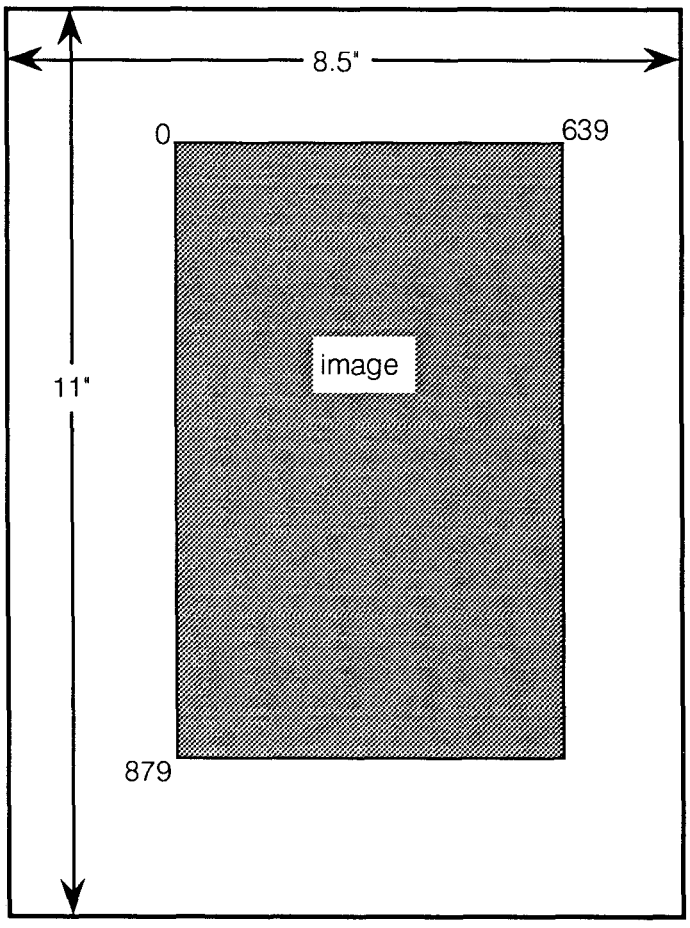
Knowing a pixel's state may increase utility, so we include the following:
d) Reading a pixel

To better follow, here is my system. My computer is a VSI PC 286 name-brand "compatible," with VGA, 40-meg. hard drive, both 5 1/4" and 3 1/2" floppy drives, a mouse, H-P DeskJet 500, a modem, and four megs. of memory. I have DOS 5, Windows 3.1, and HS-Forth 4.11 (regular—i.e., uses segmented memory).

Simplifying the drawing process to black-and-white drawings 640 pixels by 880 pixels results in an image space of 70,400 bytes (640*880/8=70400). I have no method of accessing this image space without using paragraph/offset memory addressing. Again I point to Dr. Noble's text, this time for a lucid explanation of paragraph/offset addressing.

Of the ways I have to perform paragraph/offset addressing, the simplest uses HS-Forth's SEGMENT command. SEGMENT expects the number of bytes (plus one) to allocate, and creates a named array holding a pointer and other variables. Fetching this pointer provides the memory paragraph segment.

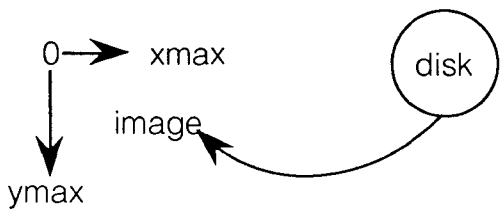
HS-Forth provides a large set of commands accessing segmented memory via paragraph and offset. However,



none were found for bit accessing, so we use C@L and C!L. C@L expects paragraph and offset on the stack, and returns a byte. C!L expects a byte, paragraph, and offset, and stores the byte.

For navigation (see code at the end of the article), we define three SEGMENTs: HEADER.SEG, IMAGE.BOT.SEG, and IMAGE.TOP.SEG. IMAGE.BOT.SEG holds image rows 879 through 440, while IMAGE.TOP.SEG holds rows 439 through zero. (Recall that Paintbrush reverses the order of the image rows.)

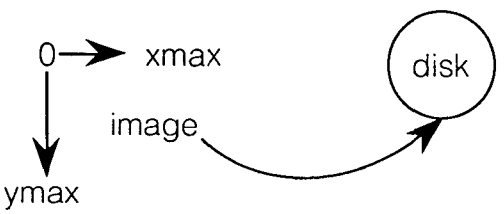
To get an image of the Paintbrush .BMP file into memory, we use GET-IMAGE. First, GET-IMAGE uses HS-Forth's OPEN-R to find the file whose file-spec address is passed, and OPEN-R obtains a file handle. The handle is stored on the return stack.




Using HS-Forth's READH command, GET-IMAGE then fills the respective memory areas with a file's contents. READH expects a memory paragraph segment, offset, file handle, and number of bytes to read. It returns on the stack the number of bytes it read into the memory location.

GET-IMAGE "anchors" the image at IMAGE.BOT.SEG. In other words, after the first half of the image is loaded into memory, the location of the second half is computed relative to the first. (Math for the last segment takes advantage of the fact that eighty divided by sixteen equals five.) CLOSEH consumes the file handle as it closes the file.

PUT-IMAGE works analogously to GET-IMAGE. Neither command tests the header or image, but simply moves the data. Both commands need rewriting for better error handling, if needed.

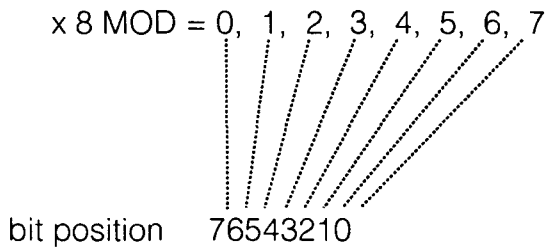



The image is accessed relative to its first row, which happens to be its last row in memory. The arcane method of calculating the paragraph's segment of the first row of the image is shown next in the code. Fetching the IMAGE.BOT.SEG paragraph starts the process. Indexing to the very first image row requires moving the pointer down 879 rows. One VAR receives the segment value, another receives the offset.

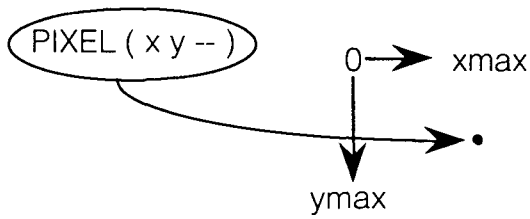
Mapping x and y coordinates to the image byte containing the pixel uses the command XY->ADR. Notice the use of high-level words to write a machine language command using the HS-Forth OPT" command. XY->ADR converts y into a paragraph segment by multiplying y by minus five and adding the result to the location of row zero (which is last in memory).

Dividing x by eight converts the pixel number into the byte number in the row. Adding that to the offset indexes into the image row. (For several reasons, the beginning offset of each of the image rows is zero, so some of these steps are superfluous for the images we are accessing.)

The remainder of x divided by eight points to the pixel, as shown below.

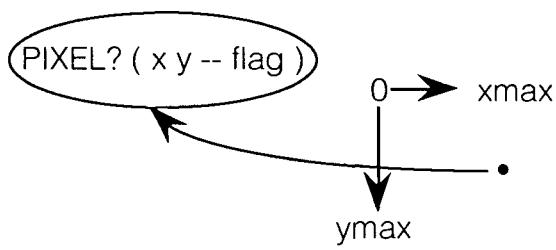



The above is performed inside the PIXEL command by 8 MOD (obtaining the bit position) and the data structure CLEAR.BIT. CLEAR.BIT provides a mask for ANDing the bit clear (i.e., to zero), thus forcing the pixel lit. In our case, Paintbrush thinks a zero pixel is lit (black) while a pixel equal to one is white. (PIXEL performs no range checking, so either make your own, or be careful.)




The PIXEL? command works like PIXEL except a flag is returned instead of the pixel being set. PIXEL? selects a bit, then tests it. When the pixel tested is zero, we say it is lit, or true. Otherwise, PIXEL? returns false.

Finally, we have filename buffers and (rather uninteresting) test code. TEST1 fills an entire Paintbrush page with black pixels in about twenty-five seconds on my system.




TEST2, TEST3, and TEST4 draw lines. T1 sets the filenames buffers and loads an image. GB$ is set to a blank page previously created in Paintbrush, while PB$ is set for keeping results.

These tests were used to verify PIXEL and, oddly enough, GET-IMAGE and PUT-IMAGE. Strangely, I still haven't completely figured out HS-Forth's SEGMENT com-

# HARVARD SOFTWORKS

*NUMBER ONE IN FORTH INNOVATION*

(513) 748-0390 P.O. Box 69, Springboro, OH 45066

You already know HS/FORTH gives more speed, power, flexibility and functionality than any other implementation. After all, the majority of the past several years of articles in Forth Dimensions has been on features first developed in HS/FORTH, and many major applications discussed had to be converted to HS/FORTH after their original dialects ran out of steam. Even public domain versions are adopting HS/FORTH like architectures. Isn't it time you tapped into the source as well? Why wait for second hand versions when the original inspiration is more complete and available sooner.

Of course, what you really want to hear about is our **SUMMER SALE!** Thru August 31 only, you can dive into Professional Level for $249. or Production Level for only $299. Also, for each utility purchased, you may select one of equal or lesser cost free.

Naturally, these versions include some recent improvements. Now you can run lots of copies of HS/FORTH from **Microsoft Windows** in text and/or graphics windows with various icons and pif files available for each. Talk about THE tool for hacking Windows! But, face it, what I really like is cranking up the font size so I can still see the characters no matter how late it is. Now that's useful. Of course, you can run bigger, faster programs under DOS just as before. Actually, there is no limit to program size in either case since large programs simply grow into additional segments or even out onto disk.

Good news, we've redone our **DOCUMENTATION!** The big new fonts look really nice and the reorganization, along with some much improved explanations, makes all that functionality so much easier to find. Thanks to excellent documentation, all this awesome power is now relatively easy to learn and to use.

And the Tools & Toys disk includes a complete mouse interface and very flexible menu support in both text and graphics modes. **Update to Revision 5.0**, including new documentation, from all 4.xx revisions is $99. and from older systems, $149. The Tools&Toys update is $15. (shipping $5.US, $10.Canada, $22.foreign)

**HS/FORTH** runs under MSDOS or PCDOS, or from ROM. Each level includes **all** features of lower ones. Level upgrades: $25. plus price difference between levels. Source code is in ordinary ASCII text files.

HS/FORTH supports megabyte and larger programs & data, and runs as fast as 64k limited Forths, even without automatic optimization -- which accelerates to near assembler language speed. Optimizer, assembler, and tools can load transiently. Resize segments, redefine words, eliminate headers without recompiling. Compile 79 and 83 Standard plus F83 programs.

**PERSONAL LEVEL $299.**
**NEW! Fast direct to video memory** text & scaled/clipped/windowed graphics in bit blit windows, mono, cga, ega, vga, all ellipsoids, splines, bezier curves, arcs, turtles; lightning fast pattern drawing even with irregular boundaries; powerful parsing, formatting, file and device I/O; DOS shells; interrupt handlers; call high level Forth from interrupts; single step trace, decompiler; music; compile 40,000 lines per minute, stacks; file search paths; format to strings. software floating point, trig, transcendental, 18 digit integer & scaled integer math; vars: A B * IS C compiles to 4 words, 1..4 dimension var arrays; **automatic optimizer delivers machine code speed.**

**PROFESSIONAL LEVEL $399.**
hardware floating point - data structures for all data types from simple thru complex 4D var arrays - operations complete thru complex hyperbolics; turnkey, seal; interactive dynamic linker for foreign subroutine libraries; round robin & interrupt driven multitaskers; dynamic string manager; file blocks, sector mapped blocks; x86&7 assemblers.

**PRODUCTION LEVEL $499.**
Metacompiler: DOS/ROM/direct/indirect; threaded systems start at 200 bytes, Forth cores from 2 kbytes; C data structures & struct+ compiler; MetaGraphics TurboWindow-C library, 200 graphic/window functions, PostScript style line attributes & fonts, viewports.

**ONLINE GLOSSARY $ 45.**

**PROFESSIONAL and PRODUCTION LEVEL EXTENSIONS:**

**FOOPS+** with multiple inheritance **$ 79.**
**TOOLS & TOYS DISK $ 79.**
**286FORTH** or **386FORTH $299.**
16 Megabyte physical address space or gigabyte virtual for programs and data; DOS & BIOS fully and freely available; 32 bit address/operand range with 386.
**ROMULUS** HS/FORTH from ROM **$ 99.**

Shipping/system: US: $9. Canada: $21. foreign: $49. We accept MC, VISA, & AmEx

mand. When I first wrote these routines, I used both IMAGE.BOT.SEG and IMAGE.TOP.SEG, but could never get the halves to align. Anchoring the image relative to IMAGE.BOT.SEG in both GET-IMAGE and PUT-IMAGE works around this confusion.

Much still is left to learn. HS-Forth already has graphics commands that, perhaps, may interface to a Paintbrush file. The .BMP file's header needs to be understood as well. Surely the desire to use color will become overwhelming. Additionally, different rules for drawing, like XOR, OR, NAND, etc., could be accommodated. Solving these requirements will open the way for error handling, range checking, and optimizing for speed. However, as they are, the routines shown here draw.

```
\ 11:19PM 1/29/93 by Hank Wilkinson
\ Code for drawing into Windows Paintbrush ".BMP" files
\ Reads and writes black and white 640 by 880 images
\ Draws via PIXEL ( x y -- )
\ Assumes .BMP image, 640 bits wide and 880 bits long
\ & page header (first 62 bytes of file)

DECIMAL
FIND SEQ-FND 0= ?(((
FLOAD C:\HSFORTH\FASTCOMP
FLOAD C:\HSFORTH\AUTOOPT
FLOAD C:\HSFORTH\TOOLBOX
\ FLOAD C:\HSFORTH\8088ASM
FLOAD C:\HSFORTH\DOSEXT
\ FLOAD C:\WINDOWS\DOCS\TOOLS\GETPUT.FTH
)))
TASK HANK

\ code below is for article
\ -------------------------
       62 1+ SEGMENT HEADER.SEG \ header
440 80 * VAR 440*80
440*80 1+ SEGMENT IMAGE.BOT.SEG \ top half
440*80 1+ SEGMENT IMAGE.TOP.SEG \ bottom half

\ Use: $" \path\filename" GET-IMAGE
: GET-IMAGE ( adr -- )
  OPEN-R >R
     HEADER.SEG @          0     62 R@ READH DROP
  IMAGE.BOT.SEG @            0 440*80 R@ READH DROP
  IMAGE.BOT.SEG @ 440 5 * + 0 440*80 R@ READH DROP
  R> CLOSEH ;

\ Use: $" \path\filename" PUT-IMAGE
: PUT-IMAGE ( adr -- )
  MKFILE >R
     HEADER.SEG @            0     62 R@ WRITEH DROP
  IMAGE.BOT.SEG @            0 440*80 R@ WRITEH DROP
  IMAGE.BOT.SEG @ 440 5 * + 0 440*80 R@ WRITEH DROP
  R> CLOSEH ;

\ set up pointer to memory location of first row of image
IMAGE.BOT.SEG @           \ segment of last row
 879 80 16 / *            \ 80 bytes/16 = segments per row
  + VAR FIRST.ROW.SEG
  0 VAR FIRST.ROW.OFF

-5 VAR -5
\ given X and Y, divine a paragraph and offset (address)
CODE XY->ADR ( X Y -- PARAGRAPH OFFSET )
   OPT"
    -5 * FIRST.ROW.SEG +
       SWAP
         8/ FIRST.ROW.OFF + " END-CODE

\ To convert x 8 MOD POWER into bit mask
\ x 8 MOD yields the bit# of the byte to bit
\ position of pixel ( BIT#->POSITION )

  8 1VAR CLEAR.BIT
127 0 IS     CLEAR.BIT
191 1 IS     CLEAR.BIT
223 2 IS     CLEAR.BIT
239 3 IS     CLEAR.BIT
247 4 IS     CLEAR.BIT
```

```
251 5 IS     CLEAR.BIT
253 6 IS     CLEAR.BIT
254 7 IS     CLEAR.BIT

\ given X and Y, light the pixel
CODE PIXEL ( x y -- )   \ set pixel at x, y
  OPT"
       OVER >R
        XY->ADR
         DDUP C@L
           R> 8 MOD CLEAR.BIT AND
            -ROT C!L " END-CODE

  8 1VAR SELECT.BIT
128 1 IS     SELECT.BIT
 64 2 IS     SELECT.BIT
 32 3 IS     SELECT.BIT
 16 4 IS     SELECT.BIT
  8 5 IS     SELECT.BIT
  4 6 IS     SELECT.BIT
  2 7 IS     SELECT.BIT
  1 0 IS     SELECT.BIT

\ given X and Y, see if pixel is lit
CODE PIXEL? ( x y -- f )
\ TRUE, x y "pixel lit"
\ FALSE, X Y "pixel dark"
  OPT"
   OVER >R
       XY->ADR C@L
          R> 8 MOD SELECT.BIT AND
           0= " END-CODE

\ to put the file names in
CREATE GB$ 128 ALLOT \ for GETB file
CREATE PB$ 128 ALLOT \ for PUTB file

: T1
$" C:\WINDOWS\DOCS\TOOLS\ARTICLE2\BLNKPAGE.BMP" GB$ $!
 $" C:\WINDOWS\DOCS\TOOLS\ARTICLE2\RESULTS.BMP" PB$ $!
 GB$ GET-IMAGE ;
```

```
: TEST1
 T1
 880 0 DO 640 0 DO I J PIXEL LOOP LOOP
 PB$ PUT-IMAGE ;

: TEST2 640 0 DO I I PIXEL LOOP ;

: TEST3 880 0 DO I 2/ I PIXEL LOOP ;

: TEST4
  880 0 DO
          I 3 / I PIXEL
          I 2/ I PIXEL
          0 I PIXEL
    I 639 < IF I 879 I - PIXEL THEN
   LOOP ;

: DO.TEST
  T1
   TEST2 TEST3 TEST4
  PB$ PUT-IMAGE ;

T1
```

## ADVERTISERS INDEX

## Part 2

# UN*X Tools Used on the FSAT Project

*Jim Schneider*
*San Jose, California*

As you will recall, I concluded the last article with a discussion of *m4(1)*. This is a continuation of that article.

While *m4* is very powerful (indeed, I use it instead of *masm(1)*'s macro processor), everything has to be spelled out explicitly. For example, there's no practicable way to match a string that starts with a letter, and is followed by any combination of letters or digits. UN*X provides many tools that allow you to do this kind of pattern matching. In fact, almost all of the stream-based text processors will do this. If you wanted to print out all the lines of a file that started with a letter followed by any combination of letters or digits, this is one of many commands that would work:

```
grep '^[a-zA-Z][a-zA-Z0-9]*$' filex
```

The group of symbols in the quotes is a regular expression (see the sidebar "Regular Expressions"). The RE (regular expression) in the example says: "At the beginning of a line, match a string composed of an instance of a character in the set lower-case 'a' to lower-case 'z' inclusive, or upper-case 'a' to upper-case 'z' inclusive, followed by any number of instances of characters in the set above plus the digits to the end of the line." Thus, if filex contained

```
This is a line
 in the file
filex
      which contains
a
lot of
(different types
#of lines
99xx9
xx99sdf
```

the command in the example would print:

```
filex
a
xx99sdf
```

Regular expressions are very important in the area of automated text processing. They are useful whenever you wish to find a list of possible strings that match a specific pattern. Because it is easy to construct a routine that matches RE's, most UN*X text utilities will allow you to specify RE's to be matched and manipulated. Additionally, UN*X has a utility to create programs that recognize and manipulate strings that match regular expressions. This utility is called *lex(1)*. It is normally used in conjunction with the *yacc(1)* parser generator, because the lexical elements of a programming language (the "words," things like identifiers or operators) can usually be easily represented as regular expressions.

*lex* is a programming language in its own right. It can be called a non-procedural programming language. Instead of a linear sequence of statements (or procedures), *lex* uses a list of independent pattern action pairs. These patterns are regular expressions, and the actions are C statements that, more often than not, operate on the strings that match the regular expression. A program for *lex* consists of three sections: a definitions section, a *lex* language section, and a 'literal' section. The definitions and 'literal' sections don't have to be present in the file. The definition section can be used to declare frequently used regular expressions and what are known as start states (which will be discussed later). The 'literal' section is called that because everything in it is copied verbatim to the output file. The language section consists of the set of pattern-action pairs. *lex* generates a scanner that will "look" for the patterns in the input stream, and when it "finds" one, it executes the associated action.

The definitions section of *lex* is an outgrowth of the fact that it is easier to remember (and easier to type) `{Ident}` rather than `[_a-zA-Z][_a-zA-Z0-9]*` (the regular expression that matches an identifier in most programming languages). The definitions section is also used to declare start states. These are used to specify what is known as left context. Although some left context (basically, what's happened to the left of the current position in the input) can be specified by the '^' metacharacter (which matches the beginning of a line), in general, a regular expression can't "remember" what happened before it got to the point where it is at now. This has important consequences. For instance, there is no regular expression that can match the set of all strings of balanced parentheses. (This is not a regular set.) *lex* lets you get around this by specifying a start state. For example, the *lex* file in Figure One will recognize balanced parentheses.

**Figure One.**

```
%Start AA
%Start BB
%{
int pnest=0;
#define MATCHED 1
#define ERROR 0
%}
%%
<BB>\) { pnest--; if(!pnest) {BEGIN 0; return MATCHED; }
            else yymore(); }
<AA>\) { pnest--; if(!pnest) {BEGIN 0; return MATCHED; }
            else {yymore(); BEGIN BB;} }
<BB>\({ fprintf(stderr,"improper nesting\n"); return ERROR; }
<AA>\({ pnest++; yymore(); }
\(        { BEGIN AA; pnest ++; yymore();}
.         { fprintf(stderr,"'%c' not a parenthesis\n",*yytext); return ERROR; }
```

The rules that are active in a start state are preceded by the start state name surrounded by the characters < and >. Notice that the characters ( and ) were preceded by a \ character. This is because the parentheses characters are used for grouping regular expressions. To match a ( or a ) character, they must be either quoted or escaped. The \ character is used as an escape character throughout the UN*X operating system. It should be interpreted as: ignore the special meaning of the following character (for metacharacters), or add a special meaning to the following character (for ordinary characters). To enable the rules with specific start states, the macro BEGIN is used, followed by the name of the start state. (BEGIN is a C preprocessor macro. In essence, it expands to "start_state = ".) To disable rules with start states, use the phrase BEGIN 0. Finally, the function yymore is used to tell *lex* not to discard the buffer it saves matched input in, but to append whatever is matched later to the buffer. This is necessary because, by default, *lex* will discard the buffer it is using at the end of an action.

At this point, I should probably point out a few things. Since *lex* was designed to be used with the UN*X operating system, and the only truly well supported language on UN*X is C, *lex* will produce a C language file, and the actions for the associated patterns must be in C. The definition section is separated from the *lex* language section by a line that contains two '%' characters (and only two '%' characters). Literal C code can be included in the definition section by including it between two lines of the form '%{' and '%}'. Although there is no literal section in this file, if one were present it would be delimited from the language section the same way the definition section is delimited from the language section. *lex* stores the string it is processing in the character array yytext, and the length of the string in the integer variable yylen.

Although start symbols are useful for extending *lex* to allow it to recognize things that are not regular expressions, the primary use is for simplifying regular expressions. For example, the regular expression to recognize a quoted string in C is quite complicated:

```
\"[^"\n\\]*(\\[.\n][^"\n\\]*)*\"
```

A somewhat easier to recognize way of recognizing quoted strings is shown in Figure Two-a.

**Figure Two-a.**

```
%Start QT1
%Start QT2
%{
#define STRING 1
#define ERRSTRNG -1 /* for incorrectly terminated strings */
%%
\"                  { BEGIN QT1; yymore(); } /* start of a quote */
<QT1>[^"\n\\]*      { BEGIN QT2; yymore(); } /* all but newline, \ or " */
<QT2>\\[.\n] { BEGIN QT1; yymore(); } /* a \ escape */
<QT2>\"             { BEGIN 0; return STRING; } /* a terminating " */
<QT2>\n             { BEGIN 0; return ERRSTRNG; } /* error, newline */
```

# Regular Expressions

Regular expressions are a powerful way of specifying groups of related strings. The lexical elements of a language (the "words" of the language) are usually most conveniently represented by regular expressions. (For this discussion, an "alphabet" should be understood to be an ordered set of symbols. These symbols are the "letters" of the alphabet. Unless otherwise specified, the alphabet under discussion will be a standard computer character set.) The set of strings specified by a regular expression is known as a regular set. A string in a regular set is said to match the regular expression. These are the basic properties of regular expressions:

1. Any symbol in the alphabet is a regular expression, whose corresponding regular set contains one element: a string consisting of the symbol.
2. Any two regular expressions may be combined under the operation of alternation. The alternation will be symbolized with the 'I' metacharacter. The regular set corresponding to the new regular expression is the union of the regular sets of the original expressions. For example, the expression:
   alb
   would match a string composed of either a single 'a' or a single 'b' character.
3. Any two regular expressions may be combined under the operation of concatenation. The concatenation will be symbolized by abutting the two regular expressions. The regular set of the new regular expression will be the set of strings that can be decomposed into two strings such that the first substring is in the regular set of the first expression, and the second substring is in the regular set of the second expression. For example, this:
   ab
   would match a string composed of the character 'a' followed by the character 'b'.
4. A regular expression may be extended under the operation of Kleene closure. This means zero or more concatenations of the original regular expression. Kleene closure will be symbolized by appending the '*' metacharacter to the original regular expression. The regular set of the new regular expression will consist of all strings that can be decomposed into one or more substrings—each a member of the original regular set—and the null string. For example, this:
   a*
   would match a string composed of zero or more instances of the character 'a'.
5. The null string may be added to any regular set. This is symbolized by appending the '?' metacharacter to the corresponding regular expression. For example:
   a?
   would match a string of zero or one 'a' characters.
6. A regular expression may be delimited by matching parentheses. These are used for grouping.
7. If the regular sets corresponding to two regular expressions are identical, the regular expressions are said to be equivalent. For example, these expressions are equivalent:
   (a*b)I(ab*)
   (ab*)I(a*b)
   and they both match strings composed of either zero or more 'a' characters followed by a single 'b' character, or a single 'a' character followed by zero or more 'b' characters.

From these basic definitions, several properties should also be obvious:

1. Any finite set of strings is a regular set. If the strings are symbolized s1, s2, etc., a corresponding regular expression for the set is: s1Is2I...
2. A regular expression may be extended under the operation of positive closure. This is similar to Kleene closure, except that the regular set does not contain the null string. This is equivalent to: RERE*, but is symbolized by appending a '+' metacharacter. In layman's terms, one or more instances of the preceding regular expression.
3. If we consider the '=' character to symbolize equality, the '0' character to symbolize the null string, and the characters 'R', 'S', and 'T' to sybolize arbitrary regular expressions, then:
   a. RIS = SIR
   b. (RIS)IT = RI(SIT)
   c. (RS)T = R(ST)
   d. R(SIT) = RSIRT, and (RIS)T = RTIST
   e. 0R = R0 = R
   f. R** = R*
   g. (R) = R

Although as many levels of parentheses as wanted are allowed, they tend to reduce the readablility of the regular expression. To remove the need for them, UN*X adopts certain conventions:

1. All operators and operations on regular expressions are taken to be left associative.
2. The closure operations (symbolized by '*' and '+') have the highest precedence. Although the operation symbolized by '?' is not properly a closure, it also has the highest precedence.
3. The concatenation operation has the next highest precedence.
4. The alternation operation has the lowest precedence.

Thus, (a)I((b)*(c)) is equivalent to alb*c, while ((a)I(b)*)(c) is equivalent to (alb*)c.

Usually, certain shorthands are available to simplify the construction of regular expressions:

1. If a regular set consists only of strings of single characters, an abbreviated regular expression may be constructed by concatenating the strings and enclosing the resulting string in square brackets. Thus, the set {a, b, c} may be symbolized by [abc], which is equivalent to the regular expression albIc. The resulting regular expression is called a character class.
2. If certain characters in a character class form a "run" in the alphabet (i.e., [abcedfg]), the class may be further condensed by substituting the '-' metacharacter for the intervening characters (i.e., [a-g] is equivalent to [abcedfg]).
3. Since the alphabet is finite, the complement of a character class (i.e., all strings of one character not in the regular set of the class) is also a regular expression, and a character class. This is symbolized by inserting the '^' metacharacter as the first character in the class. For example, [^abc] is a character class corresponding to all the characters in the alphabet, except a, b, or c.
4. The '^' metacharacter, when prepended to a regular expression, matches the beginning of a line. The '$' metacharacter, when appended to a regular expression, matches the end of a line.
5. The '.' metacharacter is a character class that consists of all characters in the alphabet, except for the newline character.
6. When a metacharacter must be used literally in a regular expression (for example, you want a regular expression to match '8*3'), it can be escaped with the '\' metacharacter.
7. To include non-graphic characters in the alphabet in a regular expression, these escape sequences are used:
   \t matches the tab character
   \n matches the newline character
   The space character may be included in a character class literally, or it may be escaped with the '\' metacharacter. Thus, [ ] is a character class that will only match the space character.

Another advantage to using start states is the ability to recognize an erroneous construct, without needing an explicit regular expression for it. If the above example had been written using only regular expressions, it would look like that in Figure Two-b.

Which would you rather debug?

Start states are also useful if a regular expression can be used in more than one way. For example, the C convention of enclosing comments in `/* */` pairs can cause confusion with arbitrary pathnames used in C preprocessor statements such as this:

```
#include <sys/*.h>
```

Although this does have the potential for causing problems, they can be avoided either by using start states, or by realizing that the scanner will have to return a token coresponding to the string `<sys/*.h>` rather than a comment starting with `/*.h`. However, if the '<' character were left out (a rare, but possible, error), the scanner will probably scan the string `sys`, return it, and then scan in `/*.h`, followed by the rest of the file, until either the end of the file, or until it finds a closing `*/`, which will have the effect of commenting out whatever is in the intervening code. Although this provides quite a bit of motivation for start states, this error syndrome is not quite as bad as it looks. (I'll take this up later in the discussion of *yacc*.)

Since any legal C statement can appear in the action part of a pattern-action pair, a common usage of *lex* is to generate lexical analyzers for compilers from a set of regular expressions describing the lexical elements of the language being compiled. The typical usage is:

1. The *lex* generated scanner, which is usually called by the parser, recognizes a lexical element of the language
2. The actions section for the particular regular expression does whatever miscellaneous setup and conversion is necessary.
3. The last action in the action section returns a small integer, or token, to the caller.

Typically, a *lex* generated lexical analyzer is used as the front end for a *yacc* generated parser. *yacc* is a utility that turns a grammar specification into a C function that will accept a "statement" that is "grammatically correct." The specification for the grammar is written in a modification of "Backus Naur Form" (usually called "modified BNF," see the sidebar called "Grammar"). The *yacc* file also consists of three sections: the declarations section, the rules section, and the last section, which is copied verbatim into the output file.

Although a set of productions written in BNF is a concise description of a grammar, BNF is not without some drawbacks. The two major ones are: It's hard to tell which tokens are terminals and which are nonterminals using ordinary character sets. It's also difficult to construct an unambiguous, efficient grammar that obeys the precedence rules of the language it defines.

*yacc* gets around these limitations by requiring that all terminal tokens be declared in the declarations section, and adding precedence declaratation rules. Additionally, the conventions of using all upper-case letters for terminals and all lower-case letters for nonterminals are often used. In *yacc*, a token is declared by using the keyword `%token` in the declarations section. More than one token can be declared on a line. This line would declare the tokens `bar` and `baz`:

```
%token bar baz
```

To determine the precedence of a terminal (or operator), the keywords `%left`, `%right`, and `%nonassoc` are used, depending on whether the operator is left, right, or nonassociative. The first declared operators in the file are the ones with the lowest precedence, and operators declared on the same line have the same precedence. If more than one production can be used to reduce the input stream, the one with the highest precedence operator will be used (and if there are two or more operators in the same production, the one the farthest to the right will set the precedence for the production). If the two productions have the same precedence, the associativity rules are used (right associative operators reduce the rightmost part first, and left associative operators reduce the leftmost part first). If the two productions have the same precedence, and the operator involved is nonassociative, or no explicit precedence or associativity rules are given, *yacc* will report a conflict.

There are two types of conflict: a "shift/reduce" conflict, or a "reduce/reduce" conflict. The conflicts are given their peculiar names because of the way the *yacc* generated parser operates. At any given point, the parser is in one of several states. When the parser gets a token from *lex*, it can do one of two things (depending on the token): It can use the token to reduce the production it is working on and go to another state, or it can shift the token and the current state onto a stack and go to another state. Without rules to the contrary, when *yacc* has a choice between shifting and reducing, it will choose the shift (which means it will attempt to find a longer string of tokens to convert to a nonterminal). If *yacc* has a choice between two reductions, it will take the one that corresponds to the earlier production in the file.

Generally, conflicts are best avoided. Although the parser will probably do what you want it to do with a shift/reduce

**Figure Two-b.**

```
%{
#define STRING 1
#define ERRSTRNG -1
%}
%%
\"[^\n\\]*(\\[.\n][^"\n\\]*)*\"{ return STRING; } /* RE for a string */
\"[^\n\\]*(\\[.\n][^"\n\\]*)*\n{ return ERRSTRNG; } /* RE for error */
```

conflict, and you do have (rather crude) control over which rule is reduced in a reduce/reduce conflict, it is generally a bad idea to rely on ambiguous grammars. Almost any grammar can be rewritten to remove these conflicts.

Sometimes, an operator may have more than one precedence, depending on the context. For example, in algebraic notation, the unary minus operator should be evaluated sooner than the binary minus. Indeed, it should be evaluated before any binary operator. Thus, the correct order of evaluation of `-a&b` (where `&` is some binary operator) should be `(-a) &b`, and not `-(a&b)`, which it would be if `&` had a higher precedence than -. *yacc* allows you to get around this by tagging a production with the precedence it should use with the keyword `%prec`. Thus, if we define the pseudo-terminal `UMINUS` (pseudo because it never appears in a production, except as used in the example) to have a higher precedence than any binary operator, the following fragment:

```
expr: '-' expr %prec UMINUS
```

will tell *yacc* that this production should use the precedence of `UMINUS`, instead of -.

The *yacc* parser generator can do more than just recognize grammatically correct statements: It can also operate on them. To do this, you can add actions to the productions. The actions are C language statements that are enclosed in curly braces. If the action is at the end of the production, the action is triggered when the rule it corresponds to is reduced. If the

## Grammar

BNF (Backus Naur Form) is a compact way of describing the syntactic structure of a language. It describes what is known as a "context-free grammar," or one where the current fragment under examination doesn't depend on what has come previously.

A grammar may be specified as follows:

1. A set of terminal symbols is given. These correspond to the lexical elements of the language and can either be called tokens or terminal symbols.
2. A set of nonterminal symbols is given. These correspond to the grammatical elements of the language. These can be called nonterminals or nonterminal symbols.
3. A set of rules is given that map a string of symbols (either terminal or non-terminal) to nonterminals. These are called productions. There may be more than one production for a nonterminal.
4. A special nonterminal symbol called the start symbol is given. All productions must eventually map to the start symbol.

The production is said to produce the (or reduce to the) corresponding nonterminal. A sequence of symbols containing terminals and nonterminals can be transformed into an equivalent sequence of symbols by replacing a nonterminal with a corresponding production. This is called a derivation. For example, if this was the grammar under discussion (using the *yacc* conventions that an upper-case string or a quoted single character is a nonterminal, lower-case strings are nonterminals, and nonterminals are separated from their corresponding productions by a colon):

```
s: g
e: ID
e: '-' e
e: '(' g ')'
f: e
f: f '*' e
g: f
g: g '+' f
```

the nonterminal s (which is the start symbol for this example) would derive g. A sequence of derivations for s might take the form:

s => g => g '+' f => g '+' e => g '+' ID => f '+' ID => e '+' ID => ID '+' ID

The set of all strings of terminal symbols that can be derived from the start symbol is the language corresponding to the grammar. Any string of the set is called a sentence of the language. A string containing both terminals and nonterminals that is derivable from the start symbol is called a sentential form of the grammar. A language that can be generated from such a grammar is called a context-free language. If the languages derived from two grammars are identical, then the grammars are said to be equivalent.

A sequence of derivations may replace any nonterminal in a sentential form with an equivalent production, but for convenience in classification, we may wish to restrict ourselves to derivations that always replace either the leftmost or the rightmost nonterminal. These derivations are known as, respectively left sentential form and right sentential form (the example was derived in right sentential form). A viable prefix is a string of symbols that coresponds to a proper prefix of a sentential form (i.e., it contains up to the first n-1 symbols, where n is the total number of symbols in the sentential form).

There are two general methods of constructing parsers from grammars. The first is a top-down technique called LL (which stands for left-to-right traverse, left sentinel form). The second is a bottom-up technique called LR (which stands for left-to-right traverse, right sentinel form). An important result of compiler theory states that any language that can be parsed by an LL parser can be rewritten so that it can be parsed by an LR parser. Additionally, parsing techniques can be characterized by the number of tokens they look ahead into the input stream. For example, a parser that is able to use everything it has seen so far (its left context) plus the next terminal token to be returned from the lexical analyzer, has one token of lookahead. This is symbolized by placing a numeral corresponding to the number of lookahead tokens the parser can use in parentheses after the name of the parsing technique. Thus, you will see references to LR(0) grammars, etc.

There are three major divisions of LR(k) parsing, where k is greater than zero: simple LR (or SLR), canonical LR (abbreviated as LR), and lookahead LR (or LALR). The chief difference between these methods is: SLR builds an LR(0) parser, and then adds states to the parser to take lookahead into account; LR constructs the parser taking lookahead into account in the first place, and LALR constructs the parser by using the lookahead, and trying to collapse similar productions into the same rule. The *yacc* parser generator uses the LALR approach. This will generate an efficient parser for most grammars for which an LR parser can be constructed. If a parser cannot be generated for a grammar, the grammar is said to be "not (*whichever parse technique is used*)." For example, a grammar that can't be converted to an LR parser is said to be "not LR."

action is embedded in the production (like this: `reduce: foo { misc. C verbiage } bar`), the action is triggered when the token string leading up to it is seen. Notice that this will cause a reduce/reduce conflict if two rules have the same viable prefix, even if the actions are identical. To get around this, split the two productions into three, and perform the action at the end of the production that is common to both of the originals. If the actions required are not identical, you either have to find some way to postpone the action to the end of the production, or you will have to use some parser generator besides *yacc*. For example, this:

```
foo: footoken1 { foofunc(); } footoken2
       | footoken1 { foofunc(); } footoken3
       ;
```

must be rewritten as:

```
fooprefix: footoken1 { foofunc(); }
foo: fooprefix footoken2
       | fooprefix footoken3
       ;
```

and this:

```
foo: footoken1 { foofunc1(); } footoken2
       | footoken1 { foofunc2(); } footoken3
       ;
```

must be rewritten to move the action to the end of the productions, or at least past the point where the productions share a viable prefix.

Each token in *yacc* can have an associated value. This value is set in the actions associated with productions for nonterminals, or the lexical analyzer for terminals. It can be accessed in a similar fashion to the arguments to macros in *m4*, by `$(some digit)`, where the digit corresponds to the token in the production. The value is set by assigning a value to `$$` in the action corresponding to the production. Thus, this:

```
expr: '-' expr %prec UMINUS { $$ = - $2; }
```

would set the value of the left-hand `expr` to minus the right-hand `expr`. The lexical analyzer can set a token value by assigning something to the global variable `yylval`. The type of the values `$$`, `$1`, etc. can either be set by defining the typename `YYSTYPE` to something, or by the command `%union` in the declarations section. Since the FSAT project's C-to-Forth compiler will need to operate on strings and an associated type, I'll be using the first method, kludged to allow me to store both a string and a type value. If neither `YYSTYPE` is defined nor the keyword `%union` is used, the type will default to `int`.

Of course, we'd be in a pretty sorry state if our compilers couldn't handle syntax errors. *yacc* does have a way to easily integrate error handling, however. It uses the psuedo-terminal `error`. `error` stands for any sequence of input that doesn't form a valid production in the current context. For example, since a C statement must end with a ';' character, this production would recognize an erroneous statement:

```
statement: error ';'
```

This works by getting tokens from the lexical analyzer until the token or tokens following the psuedo-terminal `error` are seen. These tokens are discarded. This is very handy for resynchronizing the input stream after an error condition. Although this won't correctly handle the case described above, where the opening '<' character is left off a file name, it will make it possible to pinpoint the source of the error. For example, the code in Figure Three would suffice.

One final note on *yacc*. In specifying a context-free grammar for a language, it is necessary to specify the "goal" of the language. The grammar needs a "start" symbol. (This should not be confused with a *lex* start state.) Then, any token string that can be reduced to the start symbol is a "sentence" in the generated language. In *yacc*, this symbol can be declared with the `%start` directive in the declarations section. If the start symbol is not declared, *yacc* will use the first nonterminal in the language section as the start symbol by default.

This brief introduction to a few UN*X tools should not be taken as definitive, as a definitive treatment would take at least an article of this length for each of the tools I've briefly described. Again, if you want to learn more, there are several good books on the UN*X operating system.

For those of you who went to sleep during this and the last article, please bear with me. My next two articles will be about extending Forth to support porting C programs. In the articles that follow that, I'll be discussing a compiler that turns C programs into Forth.

---

The introductory article describing the FSAT Project, which aims to provide the advantages of both Forth and UN*X, appears in *FD* XV/2; the first part of the discussion of UN*X tools appears in *FD* XV/3. The author's e-mail address is `jim@net.com.com`. He'd appreciate your comments about the project, and will reply to all messages sent to that address, provided they pertain to technical aspects of the project, and not motivational aspects.

**Figure Three.**

```
%token INCLUDE /* #include token */
%token FILENAME
%{
#include "lex.h" /* lex function definitions file */
%}
%%
include_file: INCLUDE FILENAME { do_include($2); }
       | INCLUDE error '>'
             { eprint("badly formed preprocessor filename"); }
       ;
```

*(Sparse matrices, continued from page 10.)*

to be currently active in the sparse matrix. OUTOF_SARRAY unwires the element from the sparse matrix and places it on the available list.

Of course, actually manipulating matrix members means your code has to locate a particular element within the matrix. You accomplish this with the word S[], which accepts row and column coordinates atop the stack. S[] will return the sparse-matrix identifier of the element at that location, or NIL if the coordinates are empty.

S[] is only as smart as it can be. Notice that the structure I've described allows you to locate a member of a sparse array by walking either the row-based linked list to that member or the column-based linked list. S[] would be really smart if it followed whatever list had the fewest number of elements between the list head and the target element. Of course, there's no way S[] can know this precisely without actually counting the intervening members—but to do that would mean S[] would have to walk both lists to the target and compare the results. So S[] simply chooses the path most likely to be shortest, which is the path identified by the smaller coordinate. (I.e., if the row coordinate specified is smaller than the column coordinate, S[] searches the row list.)

```
: ZAP_SMATRIX  ( -- )
  ROWS_ARRAY @ _DISPOSPTR
  COLS_ARRAY @ _DISPOSPTR
  SMATRIX_BASE @ _DISPOSPTR
;
\ ************************************************
\ Place a sparse matrix element on the available list.
: ON_SMATRIX_AVAIL  ( i -- )
  SMATRIX_AVAIL_BASE @        \ Fetch old head
  OVER &SMATRIX.RIGHT W!      \ Store it
  SMATRIX_AVAIL_BASE !        \ Element is new head
;
\ ************************************************
\ Fetch a matrix array element from the available list
: FROM_SMATRIX_AVAIL     ( -- i )
  SMATRIX_AVAIL_BASE @              \ Fetch head of list
  DUP NIL =                         \ List empty
    ABORT" No more array space"
  DUP &SMATRIX.RIGHT W@             \ Fetch pointer to next
  SMATRIX_AVAIL_BASE !
;
\ ************************************************************
\ Return the element number of an element at index idx in a
\ vector.
\ Returns i=element # if found, NIL otherwise.  This word
\ also sets the PREV_ELEM variable for attaching/detaching
\ elements from vectors.
: <S[]>   ( idx -- i )
  R/C_BASE @ W@                     \ Fetch base of vector
  DUP PREV_ELEM W!            \ Initial previous element
  BEGIN
    DUP NIL =                 \ At end?
    IF   SWAP DROP EXIT \ Return NIL if so
    ENDIF
    2DUP &SMATRIX.IDX C@      \ Fetch index
    2DUP =                    \ Did we find a match?
    IF   2DROP SWAP DROP      \ Clear stack if so
         EXIT
    ENDIF
    >                    \ Are we past where we should look?
  WHILE
    DUP PREV_ELEM W!          \ Save previous element
    &SMATRIX.NEXT W@          \ Fetch next element
  REPEAT
  2DROP NIL                   \ Didn't find it...return NIL
;
\ ************************************************************
\ Prepare to place a sparse matrix element in a vector.
\ i is the sparse matrix element.  We assume that &SMATRIX.NEXT
\ and &SMATRIX.IDX have been set properly.
\ Returns:
\ flag = 0 if i is NOT in the vector
\ flag = 1 if i is in the vector
\ flag = 2 if someone else is in the vector where i wants to be
\ n = predecessor of i in the vector, NIL if i is or should be
\     the head member of the vector.
: WHERE_IN_VECTOR   ( i -- n flag )
  DUP &SMATRIX.IDX C@ <S[]>     \ Search the vector
  PREV_ELEM W@ -ROT             \ Save previous element
  DUP NIL =                     \ Was i found?
  IF 2DROP 0                    \ Nope
  ELSE    =                     \ Yes..izzit me? Or someone else?
          IF 1                  \ It's me!
          ELSE 2                \ It's someone else!
          ENDIF
  ENDIF
```

## Conclusion

As mentioned above, there are plenty of opportunities for extending the code. Increasing the row and column compo-

```
;
\ ************************************************
\ Locate i in row.
\ See WHERE_IN_VECTOR for results on parameter stack.
: WHERE_IN_ROW ( i -- n flag )
  NEXT_IS_RIGHT                    \ Set NEXT function
  IDX_IS_COL                  \ Index to modify is COL
  WHERE_IN_VECTOR             \ Find him
;
\ ************************************************
\ Locate i in column.
\ See WHERE_IN_VECTOR for results on parameter stack.
: WHERE_IN_COL ( i -- n flag )
  NEXT_IS_DOWN                \ Set NEXT function
  IDX_IS_ROW                  \ Index to modify is ROW
  WHERE_IN_VECTOR             \ Find him
;

\ ***************************************
\ ** INSERTING/REMOVING MATRIX ELEMENTS **
\ ***************************************
\ ************************************************************
\ Wire a sparse matrix element into a row or column vector.
\ i is the sparse matrix element, n is the predecessor in the
\ vector. n = NIL if i is to be wired in at the head of the list.
: WIRE_IN ( n i -- )
  OVER NIL =                       \ Wiring in at head?
  IF SWAP DROP                     \ Discard NIL
         R/C_BASE @ W@             \ Get old head
         OVER &SMATRIX.NEXT W!     \ Old head is our next
         R/C_BASE @ W!             \ We are new head
  ELSE   OVER &SMATRIX.NEXT W@     \ Fix our next
         OVER &SMATRIX.NEXT W!
         SWAP &SMATRIX.NEXT W!     \ Fix predecessor next
  ENDIF
;
\ ************************************************************
\ Unwire a sparse matrix element from a row or column vector.
\ Stack acts same as WIRE_IN.
: UNWIRE  ( n i -- )
  OVER NIL =                       \ Are we head?
  IF SWAP DROP                     \ Discard NIL
         &SMATRIX.NEXT W@          \ Our next is new head
         R/C_BASE @ W!
  ELSE   &SMATRIX.NEXT W@          \ Our next...
         SWAP &SMATRIX.NEXT W!     \ ...is predecessor's next
  ENDIF
;
\ ********************************************************
\ Given i -- index to a sparse matrix element -- link this
\ element into the sparse matrix at row,col.
: INTO_SMATRIX ( row col i -- )
  >R                               \ Save i
  \ Set the row & column members
  OVER R@ &SMATRIX.ROW C!          \ Set row
  DUP R@ &SMATRIX.COL C!           \ Set column
  \ Wire element into row
  SWAP SET_ROW_BASE                \ Set the row base address
  R@ WHERE_IN_ROW                  \ Locate where to wire in
  0=                               \ Flag=0 means slot is free
  IF R@ WIRE_IN                    \ Wire him in
  ELSE    ." Error attaching to row" CR
          ABORT
  ENDIF
  \ Wire new element into column
  SET_COL_BASE                     \ Set the column base address
```

nents of an element would allow for truly gargantuan sparse matrices. I suppose if you needed sparse multi-dimensional arrays, you could add the necessary anchoring arrays, extend the element structure, and add the necessary words for access to element components. Off the top of my head, I can't imagine an application for such structures... but I'll bet there's one out there somewhere.

---

Rick Grehan is the Technical Director of BYTE Labs. He wrote the code for this article with Creative Solutions' MacForth.

```
  R@ WHERE_IN_COL                       \ Locate where to wire in
  0=                                    \ As before, slot must be free
  IF R> WIRE_IN                         \ Wire him in the column
  ELSE    ." Error attaching to col" CR
          ABORT
  ENDIF
;
\ ******************************************************
\ Given i -- index to sparse matrix element -- unlink him
\ from the sparse matrix.
: OUTOF_SMATRIX    ( i -- )
  >R                                \ Save i
  \ Look for this guy in the row
  R@ &SMATRIX.ROW C@ SET_ROW_BASE \ Set row base pointer
  R@ WHERE_IN_ROW                   \ Search for i in row vector
  1 =                               \ i has to be there
  IF R@ UNWIRE                      \ Disconnect
  ELSE    ." Error removing from row" CR
          ABORT
  ENDIF
  \ Look for this guy in column
  R@ &SMATRIX.COL C@ SET_COL_BASE \ Set column base pointer
  R@ WHERE_IN_COL                   \ Search for i in column vector
  1 =                               \ i has to be there
  IF R@ UNWIRE                      \ Disconnect
  ELSE    ." Error removing from column" CR
          ABORT
  ENDIF
  R> DROP                           \ Clean return stack
;

\ ************************************
\ **  SEARCHING FOR MATRIX ELEMENTS  **
\ ************************************
\ ***************************************************************
\ Look for a sparse matrix element at row, col by row.  That is,
\ the search begins at the base of the row and proceeds "across".
\ Return i or NIL (if nothing exists at that location)
: S[]_BYROW    ( row col -- i )
  IDX_IS_COL                        \ Watch column index
  NEXT_IS_RIGHT                     \ Scan along a row
  SWAP SET_ROW_BASE                 \ Set the row/column base
  <S[]>                             \ Go git it
;
\ ***************************************************************
\ Look for a sparse matrix element at row,col by col.  That is,
\ the search begins at the top of the column and proceeds "down".
\ Return i or NIL
: S[]_BYCOL    ( row col -- i )
  IDX_IS_ROW                \ Watch row index
  NEXT_IS_DOWN              \ Scan along column
  SET_COL_BASE              \ Set the row/column base
  <S[]>                     \ Go git it
;
\ ******************************************************
\ Find a sparse matrix element at row,col.
\ This routine selects a search by row or column depending
\ on which index is smaller.
: S[]     ( row col -- i )
  \ Search by row or by column?
  2DUP
  >
  IF      S[]_BYCOL         \ Find by column
  ELSE    S[]_BYROW         \ Find by row
  ENDIF
;
```

(Letters, continued from page 6.)

```
variable BUG FALSE bug !
variable TRACE   TRUE trace !

: (    [chr] ) parse
       bug @ if
             please '." ~ " .S CR '
       else  2drop   then
; immediate

: snap trace @ bug ! ; immediate
: unsnap FALSE bug ! ; immediate
```

The comparison of what you say is on the stack with what is actually there usually identifies the problem very quickly.

You can also trace just to see what's going on.

If you type snap and recompile SQROOT, typing 180 sqroot . gives you:

```
term -1
term 1
term 3
term 5
term 7
term 9
term 11
term 13
term 15
term 17
term 19
term 21
term 23
term 25
13
```

To avoid too much output in SQRT I bracket the stack comments I want printed with snap and unsnap.

```
UNSNAP

: sqrt        ( radicand -- root )
     0           ( radicand root)
     0 ADDRESS-UNIT-BITS CELLS 2 -
     DO                    ( x y)
         2*
         OVER I RSHIFT     ( . . x')
         OVER 2* 1+        ( . . x' y')
             snap ( . . x' y') unsnap
         < NOT IF                ( x y)
             DUP           ( x . y')
             2*  1+  I LSHIFT
                 snap ( x . y') unsnap
             NEGATE +UNDER       ( x y)
             1+
         THEN
     -2 +LOOP
     NIP          ( root)
;
```

Typing 180 sqrt . gives:

```
. . x' y' 180 0 0 1
. . x' y' 180 0 0 1
. . x' y' 180 0 0 1
. . x' y' 180 0 0 1
. . x' y' 180 0 0 1
. . x' y' 180 0 0 1
. . x' y' 180 0 0 1
. . x' y' 180 0 0 1
. . x' y' 180 0 0 1
. . x' y' 180 0 2 1
x . y' 180 0 64
. . x' y' 116 2 7 5
x . y' 116 2 80
. . x' y' 36 6 9 13
. . x' y' 36 12 36 25
x . y' 36 12 25
13
```

*Procedamus in pace,*
Wil Baden
Costa Mesa, California

### Another Vote for natOOF

Dear Marlin,

I really enjoyed the article from Markus Dahm about natOOF (*FD* XV/2). I second the comments that were made by Mark Martino ("Letters," *FD* XV/3). Mark is a good guy and his enthusiasm is contagious. I hope that Markus Dahm makes his natOOF and other developments available. As Mark said, "I am ready to pay money for natOOF now. When can I get it?"

Thanks to everyone who supports the Forth Interest Group. You folks have been a real contribution to my life.

Gus Calabrese, President
WFT
Denver, Colorado

# Preparing a Press Release

Mike Elola
San Jose, California

Press releases can be your most powerful tool to promote business. The cost effectiveness of a well-received press release is unsurpassed. Although advertising guarantees you coverage, it does so at a much higher initial cash outlay. Furthermore, cost recovery for an ad may not be possible even with a mildly successful response. You need to be able to pay for advertising with disposable income, particularly new advertising for which a response has never been measured. The same is true about the overhead for a press release, but that overhead is relatively minor.

The process of creating ads and press releases can be very enlightening. That's because it forces you to think like your customers do. It also forces you to think about how your product meshes with the industry that you serve.

Because of their lower overhead, press releases can be utilized with more impunity. Nevertheless, there are important guidelines to follow, some of which I will get to shortly.

A failed press release may indicate that you haven't yet been able to illuminate your product in words and ideas that capture its value. Failures may also lead to new insights. If your surefire appeal fails to motivate the prospects you have targeted, you may be able to glean from its failure a better understanding about the values of that particular market segment. Perhaps you will discover a way to reposition your product to appeal to those values.

**You need the equivalent of a bullet's dense packaging when you tell about the benefits of a product or service.**

A press release offers you a chance to haul out those carefully honed stories and refined messages that you believe best capture your product or your service. I like the metaphor of a rifle shot to describe highly prepared marketing messages.

You should be fashioning those messages at least as carefully as you do your products. You don't just write them, you "engineer" (design) them. If your product can offer significant benefits for your target customers, then you ought to be able to find a way to express that potential so that the words rise up from the page like a rifle shot whizzing by.

You need the equivalent of a bullet's dense packaging when you tell about the benefits of a product or service. Concise and direct expression can pick up the reading pace. Fast pacing gives your messages increased impact.

Before you reach for those favorite stories and messages, however, you need to ensure that other elements are present. An essential factor is newsworthiness (or timeliness). The element of newsworthiness might be a company milestone, such as reaching one million in sales revenue for the first time. It might be a promotional event, such as an opening celebration after moving into a new facility. Often it will be a new product or service that you offer.

Press releases should also create a trail of product milestones reached. A product milestone could be hitting a sales target, such as 5,000,000 cheeseburgers sold; it could be a market-share target, such as Apple exceeding IBM in terms of sales market share for one quarter; or it could be a performance breakthrough, such as the *San Francisco Chronicle*'s ability to publish an earthquake version of their newspaper using backup emergency power only.

As a reader of press releases, I look for a timely element first, discarding anything that doesn't have one. I have been befuddled by press releases that are little more than product brochures and price lists. Without any milestone event, a news release is not the news its name would imply.

If you include coverage of old products with new products, you probably reduce your chances of getting press coverage rather than increase them. Don't be greedy. If you can obtain a paragraph of coverage for a new product and one-line mention of existing products or services, you have fared well.

You help immensely if you separate your coverage of a new product from your coverage of established products. You should at least treat old products differently than new ones. I suggest that you limit their discussion to one paragraph.

Similarly, offer distinctly different treatments for a product upgrade and a new product introduction. If one follows on the heels of another, consider generating more than one press release—perhaps a press release for each product that is new or upgraded.

If a product is complex due to its having many optional components, be absolutely clear about what comes and what doesn't come with each product package. If one quick reading does not accurately convey the various forms the product can take, it's not ready for general consumption. Test your press releases (and product sheets and brochures) by letting someone who is not acquainted with your products read the release, then asking them to tell you about your product's configurations and options.

If your information is unclear, you can hardly lay claim to

its being "released" (in a form that can be distributed and still be a service to readers).

### Review: *Insider's Guide to Getting Your Press Releases Published*

This slim guide about press releases has been prepared by Win-Win Marketing, whose mission is to provide marketing advice to small businesses. (Although they are intended to illustrate various types of press releases, the sample press releases in this guide provide us with a glimpse of the broad campaign that Win-Win Marketing has undertaken to establish its credibility.)

Topics covered in the guide's introduction include when to write press releases, how to target the "presses" to which you send informational releases, and some journalistic guidelines that help you write a press release. The book by Strunk and White, *The Elements of Style* is referenced. Besides re-emphasizing various journalistic guidelines, guidelines are also offered that apply to press releases exclusively.

In any case, you can't afford to ignore basic advice that you've already heard over and over, such as: Use short sentences and short paragraphs of no more than 50 or 60 words.

(As an aside, here's how the press releases I read for Product Watch fared: Five out of six violated the 60 word maximum paragraph length in their lead paragraph. One started with a paragraph of 134 words. One had a short letter that referenced new products described in a catalog. *If they don't bother to give their new products a journalistic treatment, why should I?* For its professionalism and conformance to guidelines, the news release I liked best was from a small company announcing a holder for H-P DeskJet/Writer Ink-Jet cartridges so that they are less likely to dry up. It consisted of about four or five short and simple paragraphs that easily fit onto one sheet of paper.)

Win-Win Marketing suggests that you write a press release to target just one type of medium or a specific publication that you are interested in. The guide talks about when to use photos as well as how they should be prepared for the demands of the print medium.

Besides photographic advice, you are offered advice about distributing the press release, about how to find the names of editors, and about using reprints of articles for your own self-promotion.

That covers the first ten pages of this short guide of about 30 pages. You get the idea that a lot of ground is covered quickly.

The offerings of advice continue with tips from editors of publications. They offer an assortment of insights: Do you want your envelope to be opened? If so, hand-write it. Do you want timely coverage? If so, understand the lead times for each publication you target.

The many guidelines and tips offered in the beginning sections eventually make way to reference information in the second half of the guide. In between, there is an informative section about the various types of press releases you might prepare.

The last five sections of this guide (about a dozen pages) are in reference format, with an occasional sprinkling of narrative. These sections include a sequence of example press releases, a list of suggestions about how article reprints might be used promotionally, a one-page press-release checklist, a list of library resources (such as directories) which can help you survey the publications that are available, and a short bibliography.

Overall, this guide offers a quick schooling in the subject of press releases. It introduces you to all the essential writing elements for various types of press releases as well as all the essential format elements of all press releases, such as the date of release, the requested release date, the headline, the continuation header, the continuation footer, the story-end indication, and a possible dateline (which is the city and state where the story originates when it is being released to a broad geographic area).

On the down side, I had a jarring experience when I tried to jump from the table-of-contents to the examples section. I became confused because I never saw the "Examples" chapter headline that I expected to see.

In most respects, the publication has a professional appearance, with the possible exceptions of its binding and its use of one-sided printing. Even though the body text is in large-size fonts, the pages still have room for an inside margin containing occasional quotations. The guide is printed on standard (U. S.) letter-size stock that is three-hole punched to fit into a report folder. Although it doesn't lie flat, your arms should not tire even if you read it without a break.

The number to call for further information about either the guide or a subscription to the Win-Win Marketing Newsletter is 408-247-0122 or 1-800-292-8625; or write to Win-Win Marketing, 662 Crestview Drive, San Jose, California 95117.

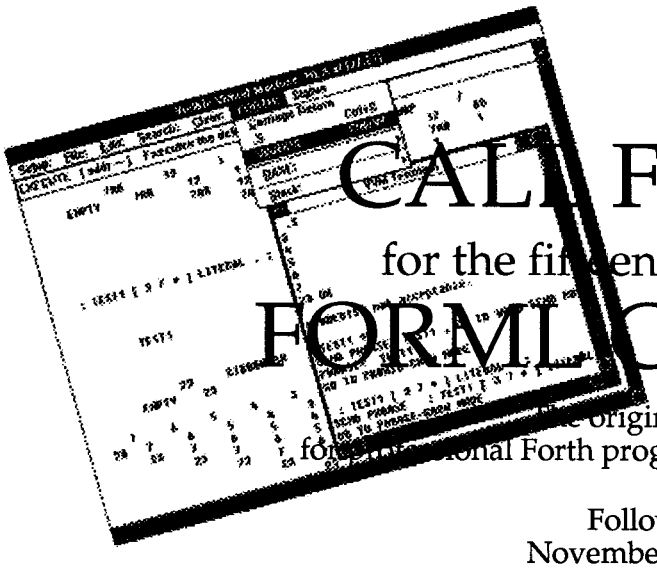
## Product Watch

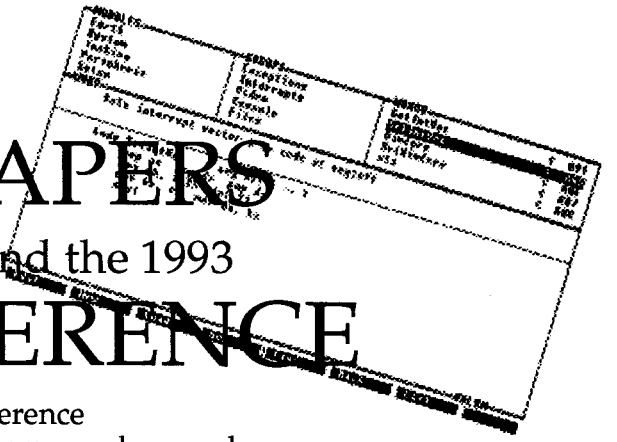
SEPTEMBER 1993

FORTH, Inc. announced a new release of its EXPRESS Event Management and Control System for process control and factory automation applications. It features high-speed I/O scan rates and improved connectivity. A new Modbus Plus driver helps achieve scan times under 10 ms. The graphical user interface in EXPRESS makes it a comprehensive development system without add-on products. Its appearance is similar to that of the OSF/Motif standards. Like other parts of the development system, the GUI is event-driven. "Rules" allow events to be generated and handled. Rules are similar to C switch statements. Constructs of this kind let you implement the state machines that serve most embedded applications.

### Companies Mentioned

FORTH, Inc.
111 N. Sepulveda Blvd.
Manhattan Beach, California 90266-6847
Fax: 213-372-8493
Phone: 800-55-FORTH





# CALL FOR PAPERS

## for the fifteenth annual and the 1993

# FORML CONFERENCE

original technical conference

for professional Forth programmers, managers, vendors, and users

Following Thanksgiving
November 26-November 28, 1993
Asilomar Conference Center
Monterey Peninsula overlooking the Pacific Ocean
Pacific Grove, California U.S.A.

## Theme: **Forth Development Environment**

Papers are invited that address relevant issues in the establishment and use of a Forth development environment. Some of the areas and issues that will be looked at consist of networked platform independence, machine independence, kernel independence, development system/application system independence, human-machine interface, source management and version control, help facilities, editor development interface, source and object libraries, source block and ASCII text independence, source browsers including editors, tree displays and source data-base, run-time browsers including debuggers and decompilers, networked development/target systems.

## Completed papers are due November 1, 1993.

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

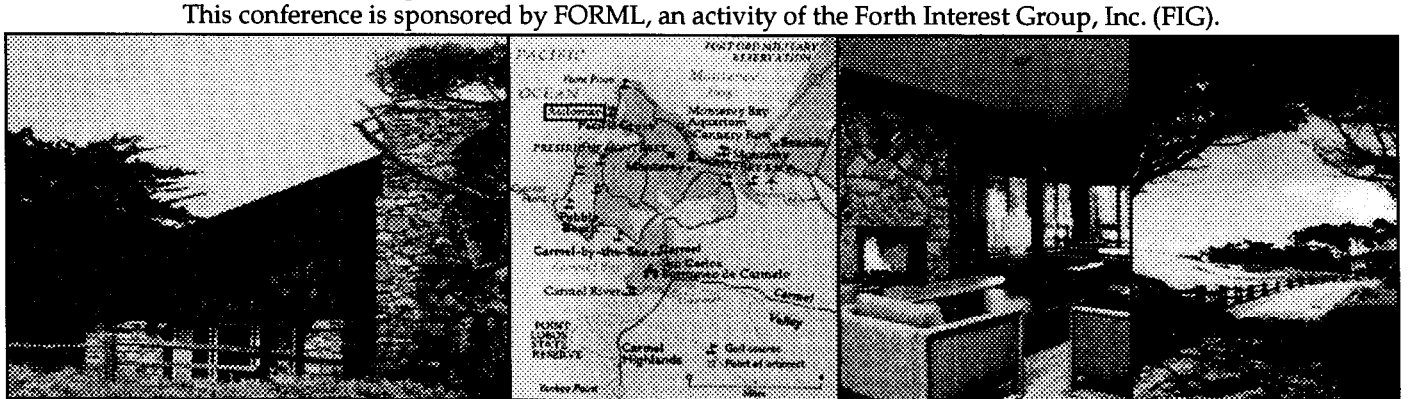
Conference attendee in double room - **$380** • Non-conference guest in same room - **$260** • Children under 18 years old in same room - **$160** • Infants under 2 years old in same room - **free** • Conference attendee in single room - **$490**

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

John Hall, Conference Chairman

Robert Reiling, Conference Director

Register by calling, fax or writing to:
Forth Interest Group, P.O. Box 2154, Oakland, CA 94621, (510) 893-6784, fax (510) 535-1295
This conference is sponsored by FORML, an activity of the Forth Interest Group, Inc. (FIG).



The Asilomar Conference Center combines excellent meeting and comfortable living accommodations with secluded forests on a Pacific ocean beach. Registration includes use of conference facilities, deluxe rooms, all meals, and nightly wine and cheese parties.