Implementing Data Structures in FORTH

James Basile

Department of Computer Science
C.W. Post Center
Long Island University, Greenvale, New York

Abstract

This paper focuses on the application of FORTH's extensibility to implementing powerful, versatile data structures. Use of CREATE ... DOES> is briefly explored. Tools are introduced to change the residency of a structure to disk, map memory, vector generic structures, and control multitasking.

This paper is intended as an exploration of some techniques that can be employed in FORTH to implement advanced data structures. The role of data structures in programming is discussed and its importance highlighted. Then we explore the use of FORTH's extensibility through the CREATE ... DOES> construct. Some simple applications of the construct are introduced and a variety of enhancements to the basic construct are detailed using the tools available in a FORTH programming environment. Several powerful, versatile, facile structures are created using the tools presented. (All code developed in the paper is FORTH 79-Standard with some well recognized extensions, e.g., FIG operators NFA , CFA).

The paper is not an introduction to the notion of a data structure itself. It is assumed that the reader knows what data structures are. The novice should first consult an introductory text on data structures such as [AHO83]. Many good references exist in the literature for actually designing particular structures. Perhaps the best is [KNU68], an encyclopaedic reference of structures and algorithms. The paper also assumes some familiarity with the use of the CREATE ... DOES> construct, although it is briefly reviewed. The tools and techniques presented herein are by no means exhaustive. The very nature of FORTH dictates that what one can conceive one can implement. So read further with an imagination open to the ways in which FORTH can be used to its fullest.

Data Structures

The novice programmer often is so engrossed in the details of simply solving a problem that he/she rarely gives any consideration to ways in which the solution can be arrived at more easily. Until the programmer has some real experience at solving problems, the underlying structure of data is often ignored. Nonetheless, as the problems one seeks to solve grow more and more complex, so does the need to realize just what relationships are present in the data being manipulated. The means chosen to represent these abstract relationships in a program comprise the data structures of the program.

Data structures are important in programming because the particular means chosen to represent an abstract relationship can have an effect upon the efficiency and ease with which elements may be stored, retrieved, and manipulated. Problems which seem insurmountable to a programmer equipped only with variables and arrays become quite routine to the programmer who has an arsenal of data structures at his/her disposal. Assuming that the reader appreciates the very

concept of a data structure espoused here, we will explore just how FORTH is so well suited to employing varied data structures.

CREATE ... DOES>

In most programming environments, the programmer is limited to those structures which are available in the language's compiler or interpreter. Any other structure has to be synthesized from these building blocks through use of some program structure such as a subroutine or function. For example, one can build a fine linked list in FORTRAN, but all you have to work with are variables and arrays. In PASCAL, the same linked list is easier to construct because of the pointer data type available. But in FORTH the ultimate is possible.

Perhaps one of the first things one learns about in FORTH is its extensibility. Yet, the novice FORTH programmer accepts this on the level of commands available as part of the interpreter. Each colon definition extends this command set, enriching the language and facilitating program design. One hears rumors of this fantastic ability — the DOES> word! — but rarely comprehends the versatility it provides. The defining structure CREATE ... DOES> is the key to implementing data structures in FORTH. It is through this feature of the FORTH system that one reaches the height of FORTH's extensibility — the ability to implement any data structure conceivable and then have it available as part of the language itself. This is the crucial difference; in FORTH the structure is not merely implemented, it becomes part of the language!

The CREATE ... DOES> structure consists of two elements: the CREATE clause and the DOES> clause. When a FORTH word has a CREATE clause in its body (or a word which uses CREATE such as CONSTANT), it becomes a defining word, i.e., when it executes it builds new FORTH words. In the absence of any other clauses, the new words created by a defining word behave as variables, returning their parameter field address when they are executed. However, when a DOES> clause is also part of the defining word, the code address of words defined by the defining word will branch to a routine which executes the code following DOES>, with the new word's parameter field address on the stack. This allows the programmer to attribute any action desired to the new words upon their execution. For a complete description of how one uses CREATE ... DOES> see [BRO81] or [LAX82].

While this paper confines discussion to the CREATE ... DOES> structure, FORTH also provides another way of associating actions with new data structures - CREATE ... ;CODE . This performs in essentially the same fashion, but allows the programmer to define the associated run-time action in assembler for added speed. Since this makes structures machine dependent we have chosen to ignore it within this paper.

Simple Data Structures Using CREATE ... DOES>

At this point perhaps some simple examples of data structures implemented using CREATE ... DOES> are in order to elucidate matters. Let's start by examining some common FORTH structures and then "create" some of our own. One can define CONSTANT to simply be

```
: CONSTANT  CREATE , DOES> @ ;
```

Thus, the defining word CONSTANT creates a variable and then associates with it the runtime behaviour of fetching from its parameter field. A bit more exciting is

```
: NOOP ;
: ACTION  CREATE  ' NOOP CFA ,  DOES> @ EXECUTE ;
```

which is used to define a class of words which execute the contents of their parameter field (cf. [MCC80]). Thus,

```
ACTION DEMO
```

defines one such word initialized to do nothing (NOOP). One may subsequently give varying

actions to DEMO by simply ticking them into place (or using more exotic measures if desired). If we define

```
: .DEMO  ' . CFA  ' DEMO ! ;
```

then executing .DEMO causes DEMO to print out the number on top of the stack, i.e.,

```
.DEMO
47 DEMO
```

causes 47 to be displayed. Changing the action assigned to DEMO to some other one is equally easy and may be compiled into other definitions; executing one word may then change the action of other words.

Thus, using CREATE ... DOES> whole new classes of words may be defined. This is distinct from most other programming environments and a welcome departure. Let us consider a bit more complicated example of the defining power of FORTH. Suppose we need the ability to distinguish if a particular item belongs to a certain class of items. We would like to have an easy, useful way of doing this. In most languages we could write a routine to scan a look-up table. In FORTH, we can actually build a structure called LOOK-UP as follows:

```
: (LOOK-UP)  ( table-addr,value -- found-addr/table-addr )
  OVER  ( copy start of table )  COUNT  ( # of entries )
  OVER +  SWAP  DO
       DUP I C@ = IF  ( found )
            SWAP DROP ( start of table )
            I SWAP  ( found location )  LEAVE
       THEN  LOOP   DROP ( value )  ;

: LOOK-UP  ( defining :  #entries -- )
                  ( new word :  value -- offset/0 )
  CREATE C,  ( store table size )
  DOES>  DUP ( copy table address )  ROT ( value )
       (LOOK-UP)  SWAP -  ;
```

The word (LOOK-UP) searches the table for the value in question and returns either the found address in the table or the address of the beginning of the table. Thus, by subtracting the beginning of the table from this returned address one arrives at a value which is false (0) when not in the table and the number of the item when in the table. The actual defining word is LOOK-UP which simply creates a byte variable (to be followed in practice by the table itself) and then sets the code address to branch to FORTH code which scans the table for a value. We can use LOOK-UP as follows

```
5 LOOK-UP ?VOWEL  65 C, 69 C, 73 C, 79 C, 85 C,
```

to build a look-up table for upper case ASCII vowels. Then,

```
: TEST  KEY  DUP EMIT  ." is a "
    ?VOWEL IF ." vowel"  ELSE ." consonant"  THEN ;
```

is a word which will take keyboard input and test if its a vowel, displaying either "vowel" or "consonant" accordingly. Note that LOOK-UP actually does more, telling which vowel (if that might be useful). Now, anytime we need to test if a value is in a given set we simply can define a look-up table for that set. The code we presented works for byte look-ups but can easily be modified by changing the table size parameter and the table values to any other precision desired. Then, just modify fetch operations and the loop increment to match. It can even work for strings!

These simple examples of using CREATE ... DOES> just serve as a warm up for the ability to use FORTH to build data structures that do what you need. If you still need more examples, try

looking at [LAX82]. We will now explore some other features we can use with this defining capability.

Disk Resident Data Structures

Another powerful feature often taken for granted in FORTH by the novice is its virtual memory allocation scheme. Recall that in FORTH the disk is separated into 1024 byte allocation units called blocks. Often, these are simply viewed as the place one writes programs. However, they form a powerful capability for creating data structures. In most programming languages you can either use your data structures in memory or you can write complicated routines to save structures onto disk and retrieve them into arrays as needed. But in FORTH the ability exists to put data structures on disk and use them by simply moving the block(s) into a virtual memory buffer by using the word BLOCK.

Suppose we were going to store sales figures for a company with a hundred sales regions scattered across the country. We want to record quarterly sales for each of fifty items as well as the number of units sold. The most straight-forward way to represent this data is a pair of three-dimensional arrays indexed by region, quarter, and item. But a little bit of quick calculation tells us that 100 regions * 50 items * 4 quarters * 4 bytes (double precision) is a bit more memory (about 160K) than we might want to spend! (In fact, under the 79 and 83 standards we can only directly address 64K.) If we were writing in most languages, we would have to come up with some routines to manage the data out on disk. In FORTH we can build it right into the structure.

We can store the information needed to address a FORTH data structure where it lives (in memory or on disk) right into the definition of the structure itself. Let's go ahead and implement the structures for our sales accounting problem. For simplicity, we will lay out the arrays without worrying about details like data compression. For each region we need 800 bytes for sales totals and 800 bytes for the number of units sold. Being generous, we'll use a whole block (1024 bytes) each for totals and volume for each region. We will define a contiguous range of blocks for any such array needed. A particular block will be selected by the region index, and then quarter and item number will specify the offset for that particular 4 byte value, yielding its address.

```
VARIABLE >REGION<          ( current region )
VARIABLE >ITEM<            ( current item # )
VARIABLE >QUARTER<         ( current quarter )

: REGION-ARRAY          ( start block -- )
    CREATE .          ( save first block of array )
    DOES>  @  >REGION< @ + BLOCK  ( fetch block for region )
        >ITEM< @ 16 * +  ( 16 bytes per item )
        >QUARTER< @ 4 * +  ( the one we want ) ;
```

We tacitly have assumed the region numbers are nice and will range from 0 to 99. (Of course, if they're not we can just use a look-up table to do the conversion!) We also have been somewhat non-FORTH-like by placing the indices in variables. This was done to simplify the stack manipulations somewhat but also because in many of our applications we'll only want to vary one of these indices while holding the others fixed. Index range checking is not included but easily could be added if desired. The magic of keeping the arrays on disk is all handled by saying BLOCK at just the right time (remember to UPDATE the block structure when storing into it, though). No mess, no fuss, no bother.

We can now use our new capabilities to write some simple applications. We define the arrays we need

```
100 REGIONAL-ARRAY SALES
200 REGIONAL-ARRAY VOLUME
```

and then we can define some subtotal routines

```
: REGIONAL-SALES  ( region -- sales total for current quarter )
    >REGION< !  0.  50 0 DO  ( for each product )
        I >ITEM< !  SALES 2@ D+  LOOP ;
: QUARTERLY-SALES  ( quarter -- sales total for current item )
    1- >QUARTER< !  0.  100 0 DO  ( for each region )
        I >REGION< !  SALES 2@ D+  LOOP ;
: ITEM-VOLUME  ( item -- volume for all regions, quarters )
    >ITEM< !  0.
    100 0 DO  ( for each region )  I >REGION< !
        4 0 DO  ( for each quarter )  I 1- >QUARTER< !
            VOLUME 2@ D+
        LOOP
    LOOP ;
```

Let's examine what we have. The first array definition (for SALES) tells us that the array starts at block 100. Note that again there is no range checking for a bad index. (If we wanted to, we could make the extent of the array a parameter for REGIONAL-ARRAYS and check.) Similarly, VOLUME is defined to start at block 200. REGIONAL-SALES assumes a quarter has been selected already and proceeds to set the current region and subtotal for each product. Only one block is accessed for the entire subtotal. QUARTERLY-SALES assumes an item has been selected and computes the subtotal for a given quarter over all regions. This subtotal uses 100 block accesses. All this points out the importance of thinking through the layout structure on blocks before hand —the most frequent summaries should direct how the indices select blocks. Finally, ITEM-VOLUME doesn't assume anything has been set. It takes an item number and sums the volume for that item across all regions and quarters. Again note the importance of the way the blocks are selected in writing the code — switching the loops would result in quadruple the block accesses!

This example illustrates how easy it is to take a data structure and make it reside on disk using FORTH. All we have to do is make BLOCK part of the DOES> action of the defining word in an appropriate way and keep the block number in the structure header; FORTH's virtual memory scheme does the rest. We can apply this same technique to any structure when we want it to reside on disk. We can even write a structure which may reside on disk or in memory, specifying its residency when it is defined. We can store either the structure's block number into the header or a 0 to indicate a memory resident structure. Then address computations can be performed by a word like

```
: 'START  ( a [structure residency value] -- start address )
    DUP @ ( residency )
    ?DUP IF ( on disk )  BLOCK SWAP DROP ( residency addr )
    ELSE  2+ ( structure follows residency word )
    THEN ;
```

This particular word assumes the disk residency value (block # or 0) immediately precedes the structure itself for memory resident versions of the structure. This feature allows us the added flexibility of putting certain occurrences of a structure on disk (e.g., very large occurrences) while letting others exist in memory. Yet the same operators will work on both transparently.

Some Memory Mapping Techniques

As indicated earlier the current standards for FORTH systems restrict addressable memory to 64K bytes. With the advent of the current generation of microprocessors it has become commonplace to have much more memory than this available for use. Leary and Winkler addressed three ways to use additional memory within this 64K restriction [LEA81]. However, the exact nature which such memory mapping takes on must to a great extent be hardware dependent. The address format and size, mapping registers, page size, etc. all will influence the optimal framework for a particular system.

One application discussed in [LEA81] was to use the extended memory for data storage. For

example, if we had enough memory, we might want to use it for those three-dimensional arrays we talked about earlier. While they clearly wouldn't fit within the 64K restriction, there really is no reason not to consider putting them in memory but outside the FORTH address space.

How can this be accomplished? [LEA81] uses the technique of windowing, i.e., reserving a portion of the 64K as a "window" that can be used to "look" at a section of the memory outside the 64K space. This section is selected by setting appropriate pointers in the hardware to address the desired section of memory.

It can also be feasible to build the mechanism for memory mapping a structure into the structure itself, much the way we did for disk resident structures. The structure can maintain a value for the physical beginning (base) of the structure and then use the address passed by FORTH (offset) to compute the physical address within the structure. For example, on an 8086 based system the 1024 maximum memory is addressed in 64K segments by use of appropriate segment registers [INT80]. These registers may be set to point to any 16 byte boundary as the beginning of a 64K address space. Then we can store in the structure the appropriate value so that when the structure is invoked a pointer is set to the base address for the structure. We then define special long fetch and store operators, L@ and L!, which use the pointer variable to change the segment base register, fetch or store a value, and then restore the segment base register. These operators are then used in place of @ and ! when accessing structures living in extended memory. This technique can be used in a similar fashion on other processors to build longer than 16 bit addresses from a 16 bit offset. Nonetheless, it requires the application to be cognizant of whether the structure is extended memory or not, so that the correct fetch and store operators are used.

Another approach, which alleviates the need for using special fetch and store operators, is to treat the extended memory as RAM disk. This factors all the dependencies down into a system's definition of block at the expense of some additional overhead. It essentially uses the block buffers as "windows" into the memory. On the 8086, we can reserve block numbers from 0 through 960 to refer to memory segments beyond 64K. This allows the complete complement of 1024K to be addressed. When the system is booted, it determines how much memory is available and allows "block" accesses into that memory range. With this scheme structures can be memory mapped by using the disk techniques discussed earlier in conjunction with block numbers specifying RAM blocks. This even could allow an application to run in RAM on one system while using disk on a smaller system with no change in the code (other than the base block of the structure). The overhead of the technique is not severe if one is using much of the structure at the same time and / or there are enough block buffers available.

This RAM block approach automatically provides FORTH applications with "spooling" capability by using block numbers correctly to minimize disk traffic. As memory allows, for example, blocks could be spooled to RAM disk before being flushed to actual disk. This could be useful in certain environments such as data collection or database manipulations. Some related techniques are available in [STA82].

Vectoring Generic Structures

Another capability that is available in FORTH is the ability to vector various data structures through variables. This allows us to write generic operators for a particular structure and then use that operator on particular occurrences of that structure without the need to change the code to match the new occurrence. Consider building and maintaining a linked test. Much of the operations to be performed — searching, inserting, deleting — is the same regardless of the specifics of the list; nonetheless, when it comes down to the level of actually manipulating the fields for the list, more information is needed. Is the list in ascending or descending order? Are the fields numeric or character? How is the data placed in the list?

One approach to answering these questions is to build a parameter list which contains the specifics for each occurrence of the structure into the header of that occurrence and then write operators which use this parameter list to decide how to process a structure. When the name of the

particular occurrence is executed, it sets a pointer to select its parameter list as the current one. This selection remains in effect until another occurrence is specified.

Thus, we can define a flexible linked list structure as follows:

```
CREATE B/E  2 C,  ( field size for the list )
: CHARACTER  ( n -- )  B E C! ;  ( select character field )
: DOUBLED  4 B E C! ;  ( select double precision field )

VARIABLE >LIST<  ( pointer to the current list )

: LINKED-LIST
( <field spec> LINKED-LIST <name> <fetch> <store> <comp> )
    CREATE  0 ,  ( head of list )  B E C@ C,  ( field size )
        2 B E !  ( reset size var )
        [COMPILE] ' CFA ,  ( list fetch operator )
        [COMPILE] ' CFA ,  ( list store operator )
        [COMPILE] ' CFA ,  ( list comparison operator )
        [COMPILE] ' CFA ,  ( list field display operator )
    DOES>  >LIST< ! ;
```

The word LINKED-LIST is the defining word for each linked list. It builds the parameter list associated with that particular occurrence of the structure >LIST< . The parameter list itself consists of the head pointer for the list, the field size of each entry in the list, and the code field addresses of operators used to store into a field, fetch from a field, compare two field values, and display a field. The head pointer is initialized to 0 to indicate an empty list. The field size is obtained from a variable B E. This variable is set for special kinds of lists by the words DOUBLE and CHARACTER to obtain fields of any desired size. B/E is reset automatically to 2 bytes, making single-precision fields the default size. The operator addresses are obtained by ' and stored. The runtime behaviour of a list word is simply to store a pointer to its parameter list in the generic linked list pointer >LIST< . We then write general operators for the list which are vectored through the parameter list.

```
: START        >LIST< @ @ ;
: SIZE         >LIST< @ 2+ C@ ;
: @LIST        >LIST< @ 3 + @ EXECUTE ;
: !LIST        >LIST< @ 5 + @ EXECUTE ;
: COMPARE      >LIST< @ 7 + @ EXECUTE ;
: DISPLAY      >LIST< @ 9 + @ EXECUTE ;
```

We are now in a position to write words to perform list operations using our general list words. The basic operations needed for a linked list are search, insertion, and deletion. To handle garbage collection for the lists we will also need an available space list to which deleted items will be linked and from which new fields will be assigned if possible.

```
: SEARCH  ( field value -- a[ptr to field], found/not found )
    PAD !LIST  ( buffer field value at PAD )
    >LIST< @  BEGIN  ( pick up a field )
        DUP @  DUP  ( next field's address )  IF
            @LIST  PAD @LIST COMPARE  ( field or bigger )
        THEN
    0< WHILE  ( more in list, still too small )  @ ( next! )
    REPEAT
    DUP @ DUP IF  @LIST  PAD @LIST COMPARE  0= ( found? )
    THEN ;
```

The word SEARCH accepts on the stack the value to be searched for (or a pointer to it for strings) and scans the list for either a match or a bigger item. All field fetches and compares are

vectored through the current parameter list so that the proper results are obtained. SEARCH returns the address of the pointer to the field which matches or exceeds the one we gave it, as well as a flag indicating whether a match was found. It is important to return the address of the pointer to the field and not the field itself so that subsequent operators can manipulate the pointers for insertion and deletion.

We assume without any loss of generality that the list is in ascending order for the sake of explanation. In fact, one could get descending order by simply reversing the sense of the compare operator. Since the routine is designed to expect COMPARE to return a negative value if field1 < field2 , zero if field1 = field2 , and positive if field1 > field2 , this can be accomplished by appending NEGATE to whatever the compare operator is. We can even gain an unordered list by following the compare operator by ABS NEGATE (think about it).

We create an available space list by defining a head pointer for it. Subsequent deletions will add freed fields to the head of this list, while subsequent insertions will scan the list for an appropriate size field. This list is not a vectored structure since all operations to this list manipulate only its pointers and the first byte of the field regardless of its actual size, thus no special processing is needed.

```
CREATE AVAILABLE 0 ,
```

Once we have found where we want to put a new item in the list (using SEARCH) we can add a spot in the list for the new entry:

```
: AFTER  ( a[ptr to field] -- a[new field] )
    ( get a spot from available space list or dictionary )
    AVAILABLE  BEGIN
        DUP @ 0=  ( end of list? )
        OVER @ 2+ C@  ( right size ? )  SIZE =
    OR NOT WHILE  @
    REPEAT
    DUP @  ?DUP IF  ( a spot, unlink it )  DUP @ ROT !
    ELSE  ( create a spot )  DROP HERE SIZE 2+ ALLOT
    THEN  ( link into current list )  OVER @ OVER !
    DUP ROT ! ;
```

The word AFTER presumes one already knows where an item is to be inserted (i.e., SEARCH has already executed). There are two functions it must perform: obtaining a memory buffer for the item and linking that memory buffer into the list. The buffer is obtained either by finding one of appropriate size in the available space list or by ALLOTing enough room in the dictionary. Then, the pointers are manipulated to make the buffer selected part of the current list.

Inserting a value into the list consists of writing code to obtain the value for the field, place the value (or its address) on the stack, use SEARCH to obtain its position in the list, create a buffer AFTER that position in the list, and store the value. Assuming that the value has already been input and it (or its address) is on the stack we define:

```
: INSERT ( value/addr -- )  DUP SEARCH
    0= IF ( not already there )  AFTER !LIST
    ELSE DROP ." item already in list"
    THEN ;
```

The principal work of deleting an item from a list is rearranging the pointers in an appropriate way:

```
: FREE  ( a[ptr to field] -- )
    ( delete field from list )
    DUP @ DUP @ ROT !  ( unlink from list )
    AVAILABLE @  OVER !  ( link into available space list )
    SIZE OVER 2+ C!  ( save size of field )
    AVAILABLE !  ( update head pointer ) ;
```

The word FREE performs the inverse of AFTER. FREE unlinks an item from the list, records its size, and chains it into the available space list. Then, assuming once again the value to be deleted (or its address) is on the stack, we define:

```
: DELETE ( value addr -- ) SEARCH
    IF FREE  ELSE ." item not found"  THEN ;
```

Displaying the list can also be done in a generic way if that is desirable.

```
: DISPLAY-LIST  ( display contents of current list )
    CR ." items in "  >LIST< @  NFA COUNT TYPE CR
    >LIST< @ BEGIN
    @ ?DUP WHILE  ( more in list )
        DUP @LIST  CR .LIST  ( display it )
    REPEAT ;
```

We can use the structure to construct a few examples. First, we will construct list operators and input and display routines for a single-precision list. Then, we will do the same for a character field list.

```
: L@  2+ @ ;    ( single-precision list fetch )
: L!  2+ ! ;    ( single-precision list store )
: LCMP  - ;     ( single-precision list comparison )

LINKED-LIST NUMBERS  L@ L! LCMP .
: READ-NUMBERS      NUMBERS ( make list active )
    BEGIN  CR QUERY INTERPRET  ( get a number )
        INSERT  ( put it in list )
        CR ." another? " KEY
    78 = UNTIL  ." ok" CR  QUIT ;
: SHOW-NUMBERS      NUMBERS  DISPLAY-LIST ;

: LS@  2+ ( skip over link ) ;
: LS!  2+ SIZE CMOVE ;
: LSCMP  SIZE CMPSTR ( CMPSTR performs string comparison ) ;
: LS."  SIZE TYPE ;

32 CHARACTER LINKED-LIST NAMES  LS@  LS!  LSCMP LS."
: READ-NAMES        NAMES
    BEGIN    CR QUERY  ( get a name )
    TIB @ C@ WHILE  ( NULL at CR position )
    TIB @ INSERT REPEAT  ." ok" CR QUIT ;
: SHOW-NAMES        NAMES DISPLAY-LIST ;
```

This idea of vectoring a structure is certainly compatible with the previous technique of making a structure resident on disk. In fact, with a few minor modifications and caveats the transition is straightforward. The layout of the structures on disk and their interaction should be carefully thought out; otherwise unacceptable levels of overhead can occur. For example, linked lists on disk can cause quite a bit of thrashing and can only be successfully used in a limited manner. Perhaps giving the structure residency on disk will mean using a different generic structure instead to achieve the desired results. Another consideration is that the fetch, store, and compare operators should say BLOCK to ensure the given block is memory resident. Finally, AFTER and FREE must be modified to allot and free space on disk instead of memory. Multitasking environments also provide no special problems for this vectoring, providing the structure pointer is defined as a user-variable rather than an ordinary variable. A detailed application of vectored structures can be found elsewhere in this issue [JOO83]. The reader is also invited to explore applying the recursive defining word techniques described in [FOR82] to accomplish some of these same objectives.

The notion of vectoring operations can be applied in other ways also. Some of the more prominent of these in the FORTH environment are based on the concept of using variables as pointers to one of several code addresses for an operator to select either at compile time or run time. The TO concept for alleviating the use of @ and ! operates at run time [BAR79] while the QUAN concept performs at compile time [ROS82]. Another use of compile time vectoring of code addresses is the scheme for selecting arithmetic operators in [BAS81]. All these vectoring methods reflect the powerful mechanisms available in FORTH.

Task Control Through Structures

Multitasking can be simplified and speeded-up through effective use of data structures. For example, a good way to pass parameters back and forth among various tasks is through use of a queue structure. But once data structures have been selected to build various tasks around, there is still a further optimization that can be employed. During multi-tasking, each task that is active gets a chance to do its job. However, some of these tasks may need input from other tasks before they can go to work. There are essentially two approaches to handling this synchronization (which parallel the situation encountered in writing device drivers for hardware). Certainly, the task itself could monitor a system flag of some sort which would indicate when necessary information were ready (just like a polling driver). Or else, we can approach the problem as does an interrupt driven driver.

By coupling FORTH's multi-tasking capabilities with the tools already presented in CREATE ... DOES> we can actually build a data structure which generates FORTH "interrupts" when it is accessed or when certain conditions within the structure are met (e.g., a full queue). Specifically, we can make the DOES> part of a structure decide if another task should be apprised of an access to the structure, and if so it might activate or kill other tasks. An example of a structure control mechanism for tasks is presented in this issue in [LEA83].

Suppose we have a task monitoring the keyboard. Characters are placed in a keyboard buffer from which they are to be processed. Another task is awaiting input from the keyboard. Either we can have both tasks active with the task awaiting input polling the keyboard or we can save overhead by building into the buffer structure a mechanism to activate the other task when it is stored into. This can minimize unnecessary multitasking overhead by keeping the number of active tasks to a minimum. Similarly, we can build mechanisms into structures to kill or suspend service tasks when they are no longer needed. If we have a printer task servicing a print queue, for example, the queue structure could both activate the task when it is passed a block number and kill it when the last block is finished.

Conclusions

We have explored some of the ways that the extensibility of FORTH can be employed to define complex, versatile data structures in a very natural manner. We focused not only on the ability to create these structures using the CREATE ... DOES> construct, but also on the capacity to have the structures inherently perform other useful roles such as determining where their residency is, vectoring among several structures of the same generic class, and serving as "controllers" for multitasking. Hopefully, the paper will inspire further activity in the areas in which FORTH can employ its extensibility in powerful ways to make FORTH programming even more enjoyable and efficient.

References

[AHO83] A. Aho, J. Hopcroft, and J. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983.

[BAR79] P. Bartholdi, "The 'TO' Solution", *Forth Dimensions*, Vol. 1, Nos. 4, 5, FIG, 1979.

[BAS81] J. Basile, "Vectored Data Structures and Arithmetic Operators", *Proc. 1981 FORML Conference*, FIG, 1981.

[BRO81] L. Brodie, *Starting FORTH*, Prentice-Hall, 1981.

[FOR82] L. Forsley, "Recursive Data Structures," *Proc. 1982 FORML Conf., FIG, 1982.*

[INT80] *The 8086 Family User's Manual*, Intel Corp., 1980.

[JOO83] R. Joosten & H. Nieuwenhuÿzen, "Vectoring Arrays of Structures", *Journal of Forth Application and Research*, Vol. 1, No. 2, Institute for Applied Forth Research, 1983.

[KNU68] D. Knuth, *The Art of Computer Programming*, Vol. 1, Addison-Wesley, 1968.

[LAX82] H. Laxen, "Defining Words", *Forth Dimensions*, Vol. 4, Nos. 1, 2, 3, FIG. 1982.

[LEA81] R. Leary and C. Winkler, "Mapped Memory Management Techniques in Forth", *Forth Dimensions*, Vol. 3, No. 4, FIG, 1981.

[LEA83] R. Leary and D.McClimans, "Message Passing with Queues", *Journal of Forth Application and Research*, Vol. 1, No. 2, Institute for Applied Forth Research, 1983.

[MCC80] M. McCourt, "The Execution Variable and Array", *Forth Dimensions*, Vol. 2, No. 4, FIG. 1980.

[ROS82] E. Rosen, "High Speed, Low Memory Consumption Structures", *Proc. 1982 FORML Conference*, FIG, 1982.

[STA82] J. R. Stapleton, "Memory Mapping in FORTH", *1982 Rochester Forth Conference on Data Bases and Process Control*, Institute for Applied Forth Research, 1982.

Dr. Basile holds a B.A. in Mathematics from SUNY College at New Paltz and M.A. and Ph.D. in Mathematics from SUNY at Stony Brook. He is an Assistant Professor in the Department of Computer Science at the C.W. Post Center of Long Island University. He is interested in database management and real-time system applications, particularly using FORTH. He is currently involved in the development of FORTH systems for 8086 and VAX11 processor.