FORTH DATA STRUCTURES: A working proposal

by Robert James Chapman
IDACOM Electronics LTD.

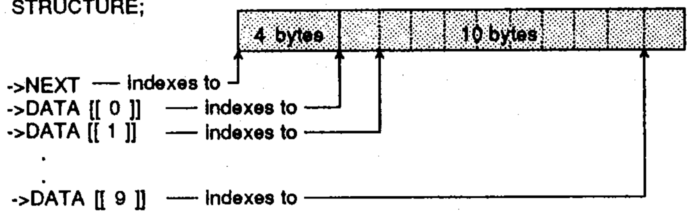
Abstract: Standard FORTH contains very little in the way of data structures. The only permanent storage structure is a variable. However, due to FORTH's extensibility, it is not limited to variables. A FORTH programmer may create any sort of data structure that may be required, but the FORTH tools for creating these structures are elementary and require a lot of effort to work with complex structures. Most of the time, no formal structure is used and the programmer must remember throughout their program whether they are accessing a word, a byte or even which bits in a bit field. Also if the structure is changed at some point, then all accesses to that structure must be changed as well. As FORTH makes more inroads into large programming environments (particularily 32-bit environments), extensions will be needed to make FORTH a more acceptable group programming language. An extension to FORTH is proposed that will provide the programmer with a small but powerful set of tools for the creation and manipulation of data structures. This data structure extension consists of four basic data types, with modifiers. These data type primitives may be used by themselves or combined to create new data types. The programmer can create data types appropriate for their environment and combine them to manipulate complex data structures. This paper is meant as a vehicle for introducing this data structure extension to FORTH and for that reason it will not go into detail about the implementation. Hopefully it will generate interest and lay groundwork for further discussion in this area.

An Example

A data structure is a template. When this template is placed over an area of memory, it provides meaning to certain portions of the memory by defining the size and access. The first part of utilizing a data structure is to define the template. The specification of a template is analogous to defining a FORTH word.

In figure 1, two structures[1] and six FORTH words have been defined. In the first structure, BUFFER is being defined as a memory template and it consists of two elements. The first element, ->NEXT[2], is used to contain a pointer to the next item in a link list and any access to it will be as a long word. The second element of the structure, ->DATA , is an array of bytes. Any access to this will be on a byte basis. Also, it is indexed on a

```
:STRUCTURE BUFFER                  ( this will be an item of a link list )
    LONG ->NEXT                    ( pointer to the next buffer )
    10 ARRAY_OF BYTE ->DATA        ( array of 10 bytes for data )
STRUCTURE;
```


4 bytes
10 bytes
->NEXT — indexes to
->DATA [[0]] — indexes to
->DATA [[1]] — indexes to
->DATA [[9]] — indexes to

```
:STRUCTURE BUFFER_POOL               ( pool of buffers )
    LONG ->FREE_LIST                 ( pointer to first buffer in free list )
    5 ARRAY_OF BUFFER ->BUFFER       ( 5 buffers as defined previously )
STRUCTURE;
```

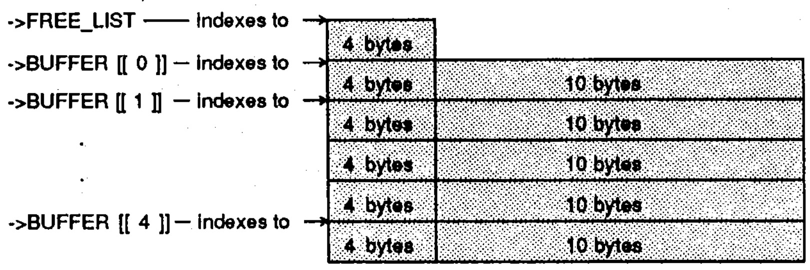

->FREE_LIST —— indexes to
->BUFFER [[0]] — indexes to
->BUFFER [[1]] — indexes to
->BUFFER [[4]] — indexes to
4 bytes
10 bytes

fig. 1 This is how a data structure for a link list buffer pool might look. The first structure defines the template for a buffer which consists of a pointer and a data portion. The second structure then defines the buffer pool which consists of a pointer for the beginning of the list and an array of buffers using the first structure. Now that the templates have been defined, they may placed over memory and the '->' words may be used to index to the specific locations.

[1] The word :STRUCTURE and STRUCTURE; are similar to : and ;. They delineate the definition and create the template word. LONG, CHAR and ARRAY_OF are simply elements of a structure. They make the following word into a FORTH definition and assign certain characteristics to it.

[2] The use of '->' is familiar to C programmers and it is used to indicate that that word is indexing into a structure.

byte basis. The second structure consists of 2 elements. The first element, ->FREE_LIST, is to be used as a pointer to a list. The second element uses the first structure to define an array of structures of type BUFFER which may be indexed with ->BUFFER to access the individual buffers. The following code illustrates the use of these structures in defining an initialization word for the buffer pool and words to obtain and return buffers in an organized manner.

Figure 3 illustrates the use of the structure words. The first statement 'BUFFER_POOL BUFFERS' reserves an area of memory that can be overlayed with the memory templates (data structures). The FORTH word BUFFERS acts just like a variable and in fact the statement 'BUFFER_POOL BUFFERS' is the equivalent of 'VARIABLE[3] BUFFERS 70 ALLOT'.

The words [[and]] are used to index into the arrays (if they are not used, then the address of the beginning of the array is left on the stack). The word PUT is used to store data into a structure and it knows what size the structure is (ie. long, word, byte, etc.). Similarily, GET is used to fetch the contents of the data structure. A walk-through example with a stack picture illustrates how the data structures affect the stack in figure 2.

Figure 4 and 5 illustrate how the words defined in figure 3 affect the data structures.

```
32            ( 32 )
GET_BUFFER    ( 32 \ address)
->DATA        ( 32 \ address+4 )
[[            ( 32 \ address+4 )[4]
5             ( 32 \ address+4 \ 5 )
]]            ( 32 \ address+4+5*14)
PUT           ()
```

fig. 2 Step by step stack illustration. The value 32 is being stored into the 5th element of the data portion of a buffer obtained from the buffer pool. If the data structure changed somewhat (ie. the ->NEXT field was removed) this code would not need to be changed.

```
BUFFER_POOL BUFFERS                              ( this allots 74 bytes of memory)
: RESET_POOL  ( -- )                             ( initialize the buffer pool )
   4 0
   DO
   BUFFERS ->BUFFER [[ I ]]                      ( address of buffer )
   BUFFERS ->BUFFER [[ I 1+ ]] ->NEXT PUT        ( point next buffer to previous )
   LOOP
   0 BUFFERS ->BUFFER [[ 0 ]] ->NEXT PUT         ( first pointer is a null pointer )
   BUFFERS ->BUFFER [[ 4 ]]                      ( address of first buffer in list )
   BUFFERS ->FREE_LIST PUT ;                     ( store address into free list pointer )
: GET_BUFFER  ( -- address | 0 )                 ( get a buffer from the pool )
   BUFFER_POOL ->FREE_LIST GET DUP               ( first buffer in free list or 0 if empty )
   IF DUP ->NEXT GET                             ( next buffer in free list, )
   BUFFER_POOL ->FREE_LIST PUT                   ( is now the first buffer in list )
   ENDIF ;
: PUT_BUFFER  ( address -- )                     ( return a buffer to the pool )
   BUFFER_POOL ->FREE_LIST GET OVER ->NEXT PUT   ( point to previous top item )
   BUFFER_POOL ->FREE_LIST PUT ;                 ( new top item )
```

fig. 3 Buffer pool initialization and access words

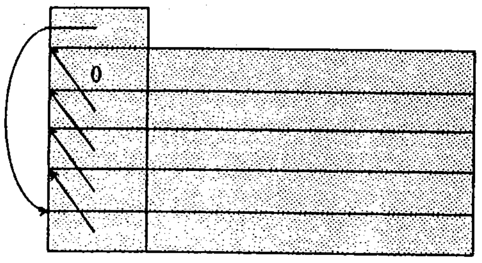


fig. 4 Buffer pool after executing: RESET_POOL. All the ->NEXT pointers have been set.

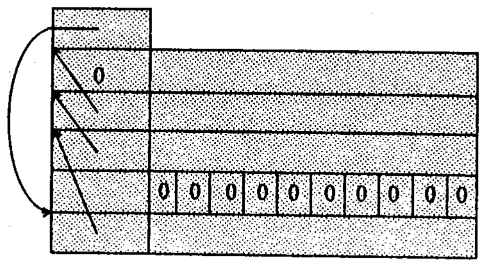


fig. 5 Buffer pool after executing: 'GET_BUFFER GET_BUFFER SWAP PUT_BUFFER ->DATA 10 ERASE'. This code removes a buffer from the pool and fills its data portion with zeroes.

[3] A 32 bit FORTH is used where the basic element is a long word. This means that VARIABLE allots 4 bytes.

[4] [[doesn't really do anything. It is an immediate word with a null body. It simply is there for readability and could even be included as part of a definition (ie. ->DATA[[to indicate that is an arrayed element).

Proposed Structure Words

This section describes the words used in creating and using data structures. The general form of a structure definition is:

```
:STRUCTURE <name>
   1{ 0{modifiers} structure-type <name> }
STRUCTURE;
```

where 'n{ }' indicates n or more of the enclosed items and <name> is a new FORTH word. The defining words :STRUCTURE and STRUCTURE; are analagous to : and ;. They may not be nested.

:STRUCTURE (--)
- makes the following word a FORTH definition that exhibits the behaviour of a structure-type. It must be used with STRUCTURE;.

STRUCTURE; (--)
- used to terminate a structure definition. Must be used with :STRUCTURE

Structure-Types

A structure-type is a word that exhibits 3 types of behaviours:

1. Inside a structure definition, it creates a word that can be used as an index into the structure.
2. If it is executed, then it creates a word and allots the space sufficient for that structure-type (ie. LONG TEMP is the same as VARIABLE TEMP in a 32-bit FORTH environment).
3. If it is used inside a colon definition then it does nothing except dictate how the access words will behave (ie. : ... BYTE GET ... ; is the same as : ... C@ ... ; this is similar to casting in C).

The following words are predefined structure-types.

LONG	(--)	- used to create a 4 byte element.
SHORT	(--)	- used to create a 2 byte element.
BYTE	(--)	- used to create a byte element.
BIT	(--)	- used to create a bit element.

Alignment is always done automatically. This means that BYTEs are done on a byte alignment and SHORTs and LONGs are done on a word alignment (this is dictated by the memory environment. ie. word-wide memory).

Modifiers

A modifier modifies the behaviour of structure-types and access operations. The words may be used inside or outside a structure definition, or inside a colon definition to 'cast' an access operation. The following words are defined and will only modify the behaviour of the predefined structure-types with the exception of ARRAY_OF which will work on any structure-type.

UNSHIFTED (--)
- used to obtain unshifted access on a bit field.

SHIFTED (--)
- used to obtain shifted access on a bit field (this is the default setting for the BIT structure-type).

SIGNED (--)
- used to make the structure-type a signed type (the most significant bit is sign extended to the full width of the stack for GET or +PUT operations).

UNSIGNED (--)
- used to make the structure-type an unsigned type (This is default and is not required unless it is desired to make it explicit or to recast an access within a FORTH definition).

ARRAY_OF (n --)
- used with any structure-type to create an array n of those types.

Access Words

The following words are used to access the data structures:

PUT (n\address --)
- used to put a value into a data structure. It is analagous to !.

+PUT (n\address --)
- used to add a value to the contents of a data structure. It is analagous to +!.

GET (address -- n)
- used to access the value of a data structure. It is analagous to @.

[[(--)
- this word is only for making source code look nice and readable. It has no affect on compiled code. It should be used with]] when indexing an array.

]] (address\n -- address+n*m)
- this is used to index into an array where n is the nth element of the array and m is the size in bytes or bits of the array element (in the case of a bit array this may be used to access a specific bit. Otherwise the whole bit array of up to 32 bits is accessed).

In addition to the above words, sometimes a word is needed to be used as a place holder in a template. This allows the element in the structure to occupy space but there is no FORTH word created for it. The word 'RESERVED' seems like a good candidate for the job as it is meaningful and it is not likely to be used for other things. This is similar to FILLER in COBOL.

```
:STRUCTURE BUFFER
      LONG                  ->NEXT
   7  ARRAY_OF  BIT         RESERVED
      BIT                   ->IN_USE
  10  ARRAY_OF  BYTE        ->DATA
STRUCTURE;
```

fig. 6 An example of a reserved area. This is the same buffer template as defined in figure 1 except it now has a bit to indicate that the buffer is in use. The bit is placed as the least significant bit by reserving the upper seven bits for future use. No word is created for the reserved bit field

An Application of Structures

I work in a 32-bit FORTH environment where our programs are on the order of 300k bytes and they are about to double as we are given more RAM to play with. The programs are rather large and usually involve more than 1 person. This places a lot of strain on FORTH to function efficiently as a group programming language. The use of data stuctures, as described in this document, helps alleviate some of the strain and maintain some conformity between programmers. Our programs deal with communication protocols which lend themselves quite well to being described in terms of data structures.

Points to Ponder

1. [[and]] might not be the best choice of words for accessing an array. However, they are being proposed since [and] are usually used to indicate arrays and FORTH already has a meaning for these words. Other alternatives include: '->' to indicate an indexing operation or '*+' to indicate that a multiply and an addition are performed.
2. By using [[and]] in a postfix manner, they are quite flexible since they may be used without being preceded by a data structure (the data structure may be left on the stack and can be indexed a number of times).
3. Signed bit fields are certainly allowable by the constructs presented here and could lead to some interesting applications. Sign extension simply extends the most significant bit to the width of the stack.
4. Since structure types may be used outside of a data structure declaration they can be quite useful. In a 32-bit environment a variable is 4 bytes. If it is just used as a flag, it would make more sense to declare it as 'BYTE FLAG' and if -1 is used as a true indicator, it could be declared as 'SIGNED BYTE FLAG'. The flag would then be accessed with PUT and GET.
5. The predefined stucture-types could also be used in making applications less tied to a certain FORTH (16-bit or 32-bit), by using them to define width-sensitive storage spaces.
6. The structure words proposed here always align memory according to the environment. If the memory is only byte wide, then no alignment is needed. If the memory is word wide then SHORT and LONG declarations must be aligned to words. If it is four bytes wide, then SHORT declarations are word aligned and LONG declarations are long word aligned. In some cases it might be desirable to not align the declarations to memory. For these cases, a modifier is needed to force no alignment.
7. It would be useful to add some words (maybe :ACCESS and ACCESS;) so that a user of this extension would be able to define their own access words. This could include words like 'GET++' which could be used to access an array sequentially.
8. The access words could work on a whole structure if no element of that structure is specified. For example, COMPLEX could be declared as a structure of two SIGNED LONGS indexed by ->REAL and ->IMAGINARY (or FLOAT could be defined as a structure of a SIGNED LONG and a SIGNED SHORT indexed by ->MANTISSA and ->EXPONENT). If a GET is performed on the structure, both elements would be returned on the stack. If the structure is indexed, then a GET only returns the indexed element. This could be extended to all operators. (ie. + or * could do complex math or floating point, if that is what is on the stack. Likewise DUP could know if it should dup 1 or n items.)

Summary

The data structure words are flexible enough to allow many simple uses, yet by combining them they are powerful enough to allow manipulation of large data structures.

Nomenclature and a methodology for data structures have been introduced. The ideas presented here are by no means final but hopefully the framework has been set for a powerful extension to FORTH.

Robert James Chapman, is a Software Engineer at IDACOM Electronics Ltd. He has used a 32 bit FORTH in several large programming projects over the last two years. He can be reached at Research Centre One, Edmonton R&D Park, 9411 - 20th Avenue, Edmonton, Alberta, Canada T5N 1E5