Data Structures for Scientific Forth Programming

J.V. Noble

Department of Physics
University of Virginia
Charlottesville, VA 22901

Abstract

This article describes methods for defining typed data (REAL, COMPLEX, REAL*8, COMPLEX*16, etc.), and for defining scalars and 1- and 2-dimensional arrays of such data. Also a method is given for run-time binding of data types that automates mixed-mode arithmetic, thereby incorporating one of FORTRAN's best features.

1. Introduction

Data structures are the soul of any computer program in any language. Some languages, most notably FORTRAN and BASIC, pre-define some data structures but require extensive contortions to define others. This straitjacket approach has virtues as well as defects:

- The pre-defined structures are what most users need to solve standard problems, so meet 80–90% of the cases in practice. That is, they are not terribly restrictive.
- Because the most-needed structures are predefined and have a standard format, they do not have to be invented each time a program is written. Standardization facilitates the exchange and portability of programs.
- Standardization of data structures also facilitates the modularization of program development, and enables subroutines written by different persons or teams to interface properly with minimal tuning.

Forth pre-defines a minimal set of data structures but permits unlimited definition of new structures. How is this different from PASCAL, ADA or even C? Forth not only permits extension of the set of data structures, it permits definition of new operators on them. Thus, e.g., Forth permits simple implementation of complex arithmetic.

This article proposes protocols for arrays and typed data that will increase the portability of code and encourage the exchange of scientific programs. The keys to this are generic operations that recognize the data type of a scalar or array variable at run-time and act appropriately.

2. Typed data structures

One of the virtues of FORTRAN or BASIC is that the programmer does not have to keep track of what type of data he is fetching and storing from memory. In fact, the user does not even have to program such operations explicitly–the compiler takes care of everything including the bookkeeping. Compiling mixed-arithmetic expressions such as

Z = −37.2E−17*CEXP(CMPLX(R**2,W)/32)/DSIN(W)

requires a great deal from a compiler. The compiler must tabulate the types of the variables and literals in the expression, and then decide which run-time routines to insert. With two types of integers and four types of floating-point numbers (REAL*4, REAL*8, COMPLEX*8 and

COMPLEX*16) a typical binary operator such as exponentiation (**) offers 36 possibilities. No wonder FORTRAN compilers are slow.

Forth sacrifices automation, opting for a small, fast, flexible compiler. The traditional Forth style gives each type of data its own operators. However, if a program demands all the standard REAL*4, REAL*8, COMPLEX*8 and COMPLEX*16 data types (not to mention INTEGER*2 and *4), having to remember them all and use them appropriately is a chore. This problem has led me to experiment with generic access operators, `G@` and `G!`. These let Forth keep track of which words to use in fetching and storing the "scientific" data types to the floating point stack (the *fstack* often resides on a co-processor like the iapx87 or MC68881 chips). Similarly, generic unary and binary floating point operators `GDUP`, `G*`, etc. keep programs general. At the suggestion of one of the referees of an earlier version of this paper, I have lately further modified the scheme to permit more complete automation. The kernel of the method is an "intelligent" *fstack*, or *ifstack*, that records the type of each number on it. The generic arithmetic operators and library functions decide from the information on the *ifstack* how to treat their operands.

An *ifstack*-based scheme for floating point and complex arithmetic has drawbacks and advantages. A major drawback is the run-time overhead in maintaining the *ifstack*, and in choosing the appropriate operator for a given situation. In other words we trade convenience for a nonnegligible execution speed penalty. To some extent this can be mitigated by computing decisions and by vectoring rather than branching (i.e. no Eaker `CASE` statements or `IF...ELSE...THEN`'s). Moreover, although the definitions are given in high-level Forth-79 for portability, the key words should be hand-coded for the target machine. Again, my definitions have plenty of error checking that could be dispensed with were speed an issue.

The chief advantages of the *ifstack* are:

- Unlike FORTRAN, this scheme permits generic routines that will accept several types of input. Hence, e.g., a matrix inversion routine will happily invert REAL, DREAL, COMPLEX and DCOMPLEX matrices.
- A FORTRAN → Forth translator [1] becomes very simple using generic operators.
- The *ifstack* permits recursive programming *a la* LISP.

2.1. Type descriptors

To decide at run-time which `@` or `!` to use for a particular datum, Forth needs to know what type of datum it is. The scheme described here wastes a little memory by attaching to each variable a label that tells `G@` and `G!` how to get hold of it.

Here is how we label types:

```
\ Data type identifiers

Ø  CONSTANT REAL              \ 4 bytes long
1  CONSTANT DREAL             \ 8 bytes long
2  CONSTANT COMPLEX           \ 8 bytes long
3  CONSTANT DCOMPLEX          \ 16 bytes long

\ a simple version of #BYTES
CREATE  <#bytes>  4 C, 8 C, 8 C, 16 C,
: #BYTES    ( type - #bytes)    <#bytes> +   C@ ;
```

2.2. Typed scalars

We want to have the machine remember for us the data-specific fetches and stores to the co-processor. To accomplish this, the typed variable has to place its address and type on the stack.

Thus we need a data structure that we might visualize diagramatically in Fig. 1 below (in such figures a cell ▨ represents 2 bytes):

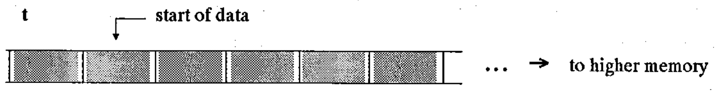

Fig. 1. Structure of a typed scalar

We implement a scalar through the defining word

```
: SCALAR  ( type - )   CREATE  DUP ,   #BYTES  ALLOT
    DOES>  DUP@   SWAP  2+  SWAP ;   ( - adr t)
```

2.3. Defining several scalars at once

One aspect of the Forth method of handling variables that seems strange to programmers familiar with Pascal, BASIC or FORTRAN is that **VARIABLE**, **CONSTANT** or a new defining word like **SCALAR** need to be repeated for each one defined, as above. That is, such defining words generally do not accept name-lists.

This idiosyncracy is not mere caprice, but follows from Forth's abhorrence of variables. Easily read (and maintained) Forth code consists of short definitions with few (generally ≤ 4) numbers on the stack. Such programs have small use for variables, especially since the top of the return stack can serve as a local variable.

In Forth as in BASIC, variables tend to be global and hence corruptible. The variables in a large program can have un-mnemonic names or names that do not express their meaning simply because we run out of names. Experienced Forth applications programmers therefore tend to reserve named variables for such special purposes as vectoring execution. The standard Forth kernel discourages named variables by making them as tedious as possible.

Most objections to variables can be resolved by making them local. Local variables are relatively easy to define in Forth: a straightforward but cumbersome method for making "headerless" words is given in Kelly's and Spies's book [2].

HS/FORTH [3] provides beheading in a particularly simple form:

BEHEAD' NAME , or **BEHEAD" NAME1 NAME2** .

Used after **NAME** has been invoked in the words that need to reference it, **BEHEAD'** removes **NAME**'s dictionary entry leaving pointers and code fields intact and recovering the unused dictionary space. The more powerful word **BEHEAD"** does the same for the range of dictionary entries **NAME1** ... **NAME2**, inclusive.

Beheading variable or constant names makes them local to the definitions that use them; they cannot be further accessed – or corrupted – by later definitions. [Pountain [4] has given another method for making variables local, using a syntax derived from "object-oriented" languages such as SMALLTALK.]

Variables are essential for scientific programming. Since we must often have more than two variables, it is silly to repeat **SCALAR**. A simple way to allow **SCALAR** to use a list is

```
: SCALARS  SWAP  Ø DO  DUP   SCALAR  LOOP  DROP ;
\ Ex:
\         2 REAL SCALARS  A B
\         5 COMPLEX SCALARS  XA XB XC XD XE
```

I find the use of SCALARs with modifiers and lists more convenient and readable than many repetitions of SCALAR. Its resemblance to FORTRAN (thereby helping me live with my FORTRAN-inspired habits) is pure coincidence. Although possible to use a terminator (", e.g.) rather than a count (to define the variable list) I feel it is desirable for the programmer to know how many variable names he has supplied, hence the counted version.

2.4. Generic access

A major theme of Forth is to replace decisions by calculation whenever possible [5]. This philosophy usually pays dividends in execution speed and brevity of code.

But there is an even more important reason to avoid IF...THEN decisions, especially when working with modern microprocessors. Chips like iapx86 and MC680x0 achieve their speed in part by pre-fetching instructions and storing them in a queue in high speed on-chip cache memory. A conditional-branch machine instruction (the crux of IF...THEN) empties the queue whenever the branch is taken. Branches should be avoided because they slow execution far more than one might expect based on their clock-counts alone.

To replace decisions, we use the standard Forth technique of the *execution array* (analogous to the familiar assembly language *jump table*). This lets us *compute* from the type descriptor which fetch or store to use.

We now define G@ and G! as execution arrays using [6] an execution-array-defining word G:

```
: G: CREATE  ]   DOES>  OVER  + +  @  EXECUTE ;  ( t -)
G: G@  R32@  R64@  X@  DX@ ;
G: G!  R32!  R64!  X!  DX! ;
```

assembled from components of the Forth compiler. For example, the ordinary colon might have the high-level definition (shorn of error detection)

```
: : CREATE ] DOES> @ EXECUTE ;
```

CREATE makes the new dictionary entry, and] switches to compile mode. DOES> specifies the run-time action (recall any word created by CREATE leaves its parameter field address on the stack at run-time, in addition to the other actions following DOES>): in the case of : it is to fetch the pfa of the new word and execute it. In the case of G: twice the type descriptor is added to the pfa, to get the offset into the execution array, before fetching and EXECUTEing.

Microprocessors like the MC680x0 and iap386 that can address large, level memories require no further elaboration for G@ and G!. However, if large arrays are to be addressed within the segmented memory addressing scheme of the 8086/80286 chips, we would have to define G@ and G! to use the "far" forms of addressing words. (For example, in HS/FORTH such words as R32@L expect a segment paragraph number and offset (32 bits total) as the complete address of the variable being fetched to the 8087 stack.) In that case we modify the definition of SCALAR to include the segment paragraph number (seg) in the definition (LISTS is nonstandard–it is HS/FORTH's word to identify the portion of the dictionary containing the headers)

```
: SCALAR  ( t - )       CREATE  DUP ,        \ make header, type
        #BYTES  ALLOT                        \ reserve space
        DOES>  >R  [ LISTS @ ] LITERAL       ( - seg)
        R@  2+  ( - seg off)  R>  @  ;       ( - seg off type)

   \ Ex:  REAL SCALAR X
```

2.5. The intelligent *fstack*

The *ifstack* is a more complex data structure than a simple *fstack* or the parameter and/or return stacks. When a typed datum is placed on the *ifstack* its type must be placed there also. But the typed data have varying lengths, from 4 to 16 bytes. Two alternatives present themselves: either ALLOT enough memory to hold a stack of the longest type, making each position on the *ifstack* 18 bytes wide; or manage the *ifstack* as a modified heap, with the address of a given datum being computable from the *ifstack*-pointer and the data type. A slight modification of the latter scheme with run-time speed advantages is to maintain a stack of pointers into the heap parallel to the type-stack. The 18-byte wide *ifstack* is extremely wasteful of memory, albeit easy to implement. (In retrospect, this is exactly the method I used to program adaptive numerical quadrature [7].) After several false attempts I settled on the last-described technique. The high level Forth code for the *ifstack* is given below. The stack comments and comments should make the code self-explanatory.

```
\ TYPED DATA STACK MANAGER  COPYRIGHT NOBLEHOUSE  1989
TASK FSTACKS

FIND X@  Ø=  ?( FLOAD C:\HSFORTH\COMPLEX.FTH )

Ø CONSTANT REAL              \ define data-type tokens
1 CONSTANT DREAL
2 CONSTANT COMPLEX
3 CONSTANT DCOMPLEX

CREATE <#bytes>  4 C,  8 C,  8 C,  16 C,
: #BYTES   <#bytes> +  C@ ;  ( type - length in bytes)

: SCALAR  ( type - )  CREATE  DUP ,   #BYTES  ALLOT
DOES>  DUP@   SWAP  2+  SWAP ;  ( - adr type)
: SCALARS  SWAP  Ø DO  DUP   SCALAR  LOOP  DROP ;
\ definitions for the parallel stack of types and addresses
\ with thanks to L. Brodie, Thinking FORTH (Brady, NY, 1984) p. 207.

CREATE  TSTACK  82  ALLOT    \ 2 tos-pointer, 2Ø 4-byte cells
HERE  CONSTANT TSTACK>
: TS.INIT   TSTACK  2+  TSTACK  ! ;     TS.INIT
: ?T.OVFLW   ( adr - adr or abort)  DUP  TSTACK>  =
   IF  ." TSTACK  OVERFLOW"  TS.INIT   ABORT  THEN  ;
: >T    ( adr t - )  4 TSTACK  +!   TSTACK  @  ?T.OVFLW   D! ;
: T.TYPE!   TSTACK @  2+  ! ;
: ?T.UFLOW   ( adr -)    DUP@  2-  >=
   IF  ." EMPTY TSTACK"   TS.INIT   ABORT  THEN  ;
: T@    ( - adr t)   TSTACK  @   D@  ;
: T.DROP   TSTACK  ?T.UFLOW   -4  TSTACK  +!  ;
: T>    ( - adr t)   T@   T.DROP  ;
: TSWAP   T>  T>  DSWAP  >T  >T  ;
```

```
\ definitions for a multi-type fstack

CREATE FHEAP    8Ø  8 *  ALLOT
HERE  CONSTANT FHEAP>

: FS.INIT   TS.INIT   FHEAP  Ø   TSTACK  2+  D!  ;   FS.INIT

: FS.ABORT   FS.INIT  ABORT  ;

: ?FS.OVFLW   ( fsp - fsp | abort on overflow)    DUP  FHEAP>  =
   IF  ." FSTACK  OVERFLOW"  FS.ABORT   THEN  ;
: >FS    ( adr[x] t[x] - :: - x )
   DUP   TS.TYPE!                   \ put type on tstack
   #BYTES                           ( - src #bytes)
   T@  DROP                         ( - src #bytes old.fsp)
   DDUP +   ?FS.OVFLW               ( - adr #bytes old.fsp new.fsp)
   Ø  >T   SWAP                     ( - src dest n)
   CMOVE  ;                         \ put number on fstack
: FS>   ( adr t - )   T.DROP   T@   ( - t fsp t')
   ROT  OVER   <>                   \ check that dest has same type
   IF  ."  ATTEMPT TO STORE TO WRONG DATA TYPE"  FS.ABORT   THEN
   #BYTES  ( - dest src n)   ROT  SWAP  ( - src dest n)    CMOVE ;
: FS.DUMP    TSTACK  ?T.UFLOW    TSTACK @    TSTACK 2+
             DO    I D@   DUP
             BEGIN-CASE  Ø  CASE-OF  CR .  R32@   F.   ELSE
                         1  CASE-OF  CR .  R64@   F.   ELSE
                         2  CASE-OF  CR .  X@     X.   ELSE
                         3  CASE-OF  CR .  DX@    X.   ELSE
                         DDROP  ABORT"  BAD DATA TYPE"  END-CASE
             4  +LOOP  ;
DCOMPLEX SCALAR TEMPØ
DCOMPLEX SCALAR TEMP1
: 'TYPE   TSTACK DUP ?T.UFLOW @  2- @  ;
: FS.SWAP    'TYPE   [ ' TEMPØ ] LITERAL !   TEMPØ  FS>
             'TYPE   [ ' TEMP1 ] LITERAL !   TEMPØ  FS>
             TEMPØ  >FS    TEMP1  >FS ;
\ execution-array defining word
( HS/FORTH has the faster CASE:…;CASE pair for the same job)
: G:   CREATE  ]   DOES>  OVER  + +  @  EXECUTE ;  ( t - )
G: G@    R32@  R64@  X@  DX@ ;
G: G!    R32!  R64!  X!  DX! ;
\ move data from ifstack to/from FPU
: FS>F     ( - t :: x - )   T.DROP  T@   UNDER  G@ ;
: F>FS     ( t - :: - x )   TS.TYPE!  T@   DDUP  G!
           ( - old.fsp t)  #BYTES   +  ( - new.fsp)  ?FS.OVFLW  ( -new.fsp)
           Ø >T  ;
```

2.6. Unary and binary generic operators

We want to define generic unary and binary operators whose run-time action selects the desired operation using information contained in the *ifstack*. A unary operator such as **FNEGATE** or **FEXP** expects one argument and leaves one result. With a floating-point coprocessor (FPU) the only distinction is between real or complex. This distinction is contained in the second bit of the type label, which we exhibit below in binary notation:

Type	16-bit Representation
REAL	00000000 00000000
DREAL	00000000 00000001
COMPLEX	00000000 00000010
DCOMPLEX	00000000 00000011

and can be extracted *via* the code fragment

(type — 0 = real | 2 = complex) **2 AND**

Most unary operators produce results of the same type as their argument. Thus we write

```
    : GU:  CREATE  ]   DOES>  ( - pfa)
       FS>F  ( - pfa t)  UNDER  2 AND   +  @  EXECUTE   F>FS ;
```

When we use **GU:** in the form

```
   GU:  GNEGATE   FNEGATE  XNEGATE ;
```

CREATE produces a dictionary entry for **GNEGATE**; **]** turns on the compiler so the previously defined words **FNEGATE** and **XNEGATE** have their addresses compiled into **GNEGATE**'s parameter field; and **DOES>** attaches the run-time code. The run-time code converts the real/complex bit into an offset, 0 or 2 which is added to the address of the daughter word to get the address where the pointer to the actual code is stored. This pointer is fetched and **EXECUTE**d.

A few unary operators like **XABS** (complex absolute value) return real values from complex arguments. If we want to use **GU:** to define, say, **GABS**, we must remember to redefine **XABS** so it zeros the second bit of the type descriptor left on the stack, before returning its result to the *ifstack*. This is just a **1 AND** so is fast.

A binary operator (one that takes two arguments) expects its arguments and their types on the *ifstack*. There is no distinction between singleand double-precision arithmetic on most numeric coprocessors. However, the result must leave the proper type-label on the stack. Here is what we want to happen, illustrated in Fig. 2 as a matrix **TYPE**(arg a, arg b) (these assignments were chosen to avoid assigning misleading precision to the result of a computation):

	a\b	R	D	X	DX
	R	R	R	X	X
TYPEab	**D**	R	D	X	DX
	X	X	X	X	X
	DX	X	DX	X	DX

Fig. 2. Matrix of types resulting from 2-argument arithmetic operations

If we think of the indices and entries in this matrix as numbers 0, 1, 2, 3 (so we can use them as indices into a table) rather than as letters, a simple algorithm emerges: the first bit of the result is the logical-AND of the first bits, and the second bit of the result is the logical-OR of the second bits of the operands. Although we would program this in assembler for speed, the high-level definition is

```
: NEW.TYPE          ( a b - a2+b2+a1b1)
   DDUP             ( - a b a b)
   AND              ( - a b  ab)
   1 AND            ( - a b  [ab]1)
   -ROT  OR         ( - [ab]1  a+b)
   2 AND            ( - [ab]1  [a+b]2)
   +  ;             ( - a2+b2+a1b1)
```

Since only logical operations are used, **NEW.TYPE** is much faster than table lookup or branching. Note that in programming this key word we have obeyed the central Forth precept: "Keep it simple!" by choosing a data structure (the numeric type tokens 0 – 3) that is easily manipulated.

We will also need a way to select the appropriate operator from a jump table of addresses. Given that the precision (internal) is irrelevant, again all that matters is whether the number is real or complex, i.e. the second bits of the numbers. The first operation must then be to divide by 2 (right-shift by one bit). We then have the matrix of Fig. 3 below where RR stands for real-real, etc. The numerical elements are generated as 2*J+I. This leads to the word

```
: WHICH.OP  ( a b - c)  2/  SWAP  2 AND  +  ;
```

$$\begin{array}{c|cc} & 0 & 1 \\ \hline 0 & RR & RX \\ 1 & XR & XX \end{array} \rightarrow \begin{array}{c|cc} & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 2 & 3 \end{array}$$

Fig. 3. Operator selection matrix

Thus,

```
: GB:  CREATE ]   DOES>                         ( - pfa)
    FS>F  FS>F                                  ( - pfa t0 t1)
    NEW.TYPE   UNDER  ( - t' pfa t')            \ make result-type
    WHICH.OP  2*   +  @  EXECUTE    F>FS  ;     \ select binop
```

with the usage

```
GB: G*   F*  F*X  X*F  X*  ;
```

The generic multiply picks out which of the four routines to use at run-time. By using only logical or shift operations we have made even the high-level definitions fairly quick in comparison with the times of floating point operations. The only instance where one might forego the overhead penalty paid for the convenience of generic coding would be in nested inner loops, such as occur in matrix operations. Here it might pay to code four inner loops, one for each type, and then access them generically, e.g.

```
: RLOOP ...real words  ;
: DRLOOP ...dreal words  ;
: XLOOP ...complex words  ;
: DXLOOP ...dcomplex words  ;
G: GLOOP   RLOOP  DRLOOP  XLOOP  DXLOOP ;
```

3. *Arrays of typed data*

Manipulation of numerical arrays represents one of the commonest categories of scientific programming. Arrays *per se* are hardly new to Forth. Arrays of typed data are, however, worth elaborating. Following Brodie's advice [8] we first specify the "user interface" (matrix notation) and only then proceed to the implementation.

3.1. Improved (FORTRAN-like) array notation

The usual notation for array elements in high-level languages (since lineprinters and terminals do not recognize subscripts) is something like V(15) – the 15th element of **V**. The RPN notation most natural to Forth, **15 V**, is hard to read and unintuitive [9]. Forth's idiosyncrasies forbid saying **V(15)** because the parser recognizes it as a single *word*. Since we want the **15** to be parsed, we must modify the FORTRAN (also BASIC) notation to **V (15)** or **V(15)**. Unfortunately, **(** is a reserved word. While we might place the matrix definitions in a separate vocabulary, **(** is too useful as a comment delineator to dispense with. This leaves the second form, in which **(** becomes part of the array name, **V(**. To make **V(15)** work, **)** has to be an operator (unless we want to leave postfix notation entirely, with all the complication *that* would entail [10]). Since **)** is *not* a reserved word, nothing in principle prevents defining it as an operator. However, I feel such usage will lead to confusion with comments. The square braces, **[ ]** are commonly used in matrix notation; however both are reserved Forth words, i.e. forbidden. This leaves the curly braces **{ }**, which are unused by Forth.

Of the two possible forms, **V { 15 }** or **V{ 15 }**, the latter has the advantage that the opening brace, **{**, is only part of the name, but reminds us that the name **V{** is an array, exactly as names ending with **$** are strings, etc. The notation suggests a further mnemonic refinement, namely to place **{{** and **}}** at the ends of 2-dimensional arrays, as in **M{{ 3 5 }}**.

How will this notation operate? Clearly, to place the (generalized) address of the n'th element (of a 1-dimensional array) on the stack we would say

```
V{ n },
```

while

```
M{{ m n }}
```

should analogously place the address of the m,n'th element of a 2-dimensional array on the stack.

3.2. Large matrices

The defining word **SCALAR** given in §2.2 above allots space in the dictionary – for most Forths, code + data must fit here – or in the **LISTS** segment of HS/FORTH (part of the dictionary). This is OK for variables, but not for arrays, since even a small matrix would exhaust the **LISTS** segment.

A filled IBM PC/XT clone has 640 Kbytes of memory. Even a generous Forth kernel (plus DOS) takes up less than 150 K; hence at least 500 K is available to hold large arrays. Up to 8 Mbytes can be added as EMS storage, assuming a suitable memory management scheme [11].

An AT/clone can address 4 Mbytes in "real" mode, and 16 Mbytes in "protected" mode. These characteristics give moderately advanced PC's the power to tackle immense systems of equations. For example, the matrix and inhomogeneous term of 300 simultaneous linear equations (in single-precision real format) take up 361,200 bytes. The time to solve them should be of order 10 minutes, on a 10 MHz machine with a co-processor. Thus it is definitely worthwhile exploring methods to use large segmented memories. Of course, level-memory machines based on e.g. the Motorola CPU's require nothing more than 32-bit addresses.

3.3. Using high memory

HS/FORTH permits accessing *all* the available memory in a PC/AT in the following manner:

- Define a named segment of length 1 byte: this marks the beginning of available memory.
- Then tell both Forth and DOS how much memory you want.

As might be expected, HS/FORTH defines non-standard words (coded as DOS function calls) to use the various DOS service routines that allocate memory, etc. [12]

```
MEMORY 4+ @  S->D   DCONSTANT  MEM.START      \ beg. of free memory
40.960 DCONSTANT   MAX.PARS                   \ 40960 = 655360 /16
: TOTAL.PARS   MAX.PARS   MEM.START  D- ;
\ # pars of memory available

1 SEGMENT SUPERSEG            \ define named segment 1 byte long
TOTAL.PARS  DROP  FREE-SIZE  \ tell DOS and HS/FORTH about it
```

Having allocated the memory, how can we address it efficiently? We would like the simplicity of double-length integer arithmetic for computing an (absolute) array address, as in

$$\text{abs.adr}(A_{ij}) = (\text{row.length}*\text{I} + \text{J}) * \#\text{BYTES}$$

However, although the absolute address referenced by a segment and offset is unique, i.e. the absolute address in bytes is

$$\text{abs.adr} = 16*\text{segment} + \text{offset},$$

the reverse translation, of an absolute address (in bytes) to the segment + offset notation expected by iapx86 processors is *not* unique. This naturally poses a problem when the processor tries to prevent segments from overlapping (protected mode). In such cases, the only answer is a memory management scheme that computes segments and offsets (by brute force) in a non-overlapping fashion. But for ordinary (8086) PC's and/or real-mode programming, we can merely ignore whether segments overlap. Although this possibility may be well-known to most PC programmers, it was unknown to me until very recently (I am indebted to Mahlon Kelly for enlightening me. It is a disgrace that none of the standard assembly programming books [13] explicitly discuss this possibility for addressing segmented memory.) That is, the PC permits 32-bit addressing as long as we translate 32-bit addresses to the segment + offset notation (expected by the iapx86 processors) after a 32-bit address computation. A word that performs this conversion is `>SEG.OFF`, defined as

```
: >SEG.OFF   ( d - seg off)
   OVER  15 AND  -ROT  ( - off d)    D16/  DROP SWAP ;
```

3.4. A general typed-array definition

For the new syntax to work the word `}` must compute the address of the *n*th element of `V{` from the information on the stack, and `}}` must do the same for `M{{`. In order to encompass matrices of typed data we specify that the results of the phrases `V{ n }` and `M{{ m n }}` be to leave the generalized address on the parameter stack, i.e. to leave the stack picture (— seg off type), exactly as with `SCALAR`s.

Before we can define `}`, however, we must specify the data structure it operates on, i.e. the array header.

Once again we begin with the user interface. We can opt for maximum generality or maximum simplicity. My first attempt fell into the first category, permitting the user to define a named segment of given length and to define an array in that segment. Lately I realized this generality accomplishes little, so have abandoned it. All arrays will be defined in the heap, named `SUPERSEG` as above. To define a length 50 `1ARRAY` of 4-byte numbers we will say

```
50 LONG REAL 1ARRAY V{
```

Now, before we work out the mechanics of **1ARRAY**, we imagine that an array will be stored as in Fig. 4 below:

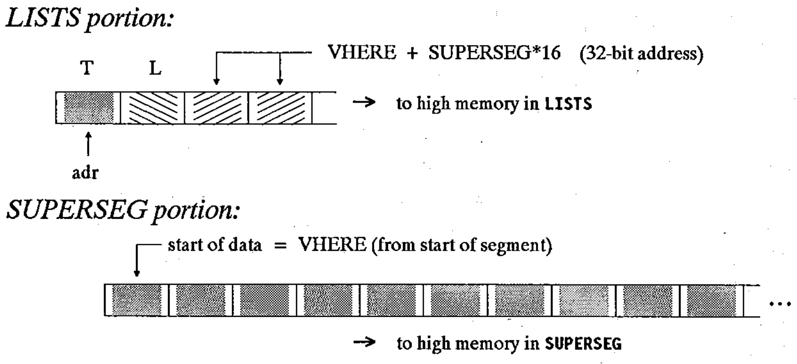
LISTS portion:
T
L
VHERE + SUPERSEG*16 (32-bit address)
→ to high memory in LISTS
adr
SUPERSEG portion:
start of data = VHERE (from start of segment)
...
→ to high memory in SUPERSEG

Fig. 4. Structure of a 1-dimensional array in SUPERSEG

The proposed array structure consists of an 8-byte header (the array descriptor) in the dictionary (LISTS in HS/FORTH). The header points to the absolute address of the array data, which is not physically contiguous with the array descriptor.

The array-defining word **1ARRAY** must perform the following tasks:

- place the length and type of the data in the first two cells (4 bytes) of the array descriptor.
- place the 32-bit address (start of data) in the next two cells (4 bytes) of the array descriptor.
- allot the necessary storage in **SUPERSEG**
- at run-time, place the generalized address, length and type on the parameter stack.

The start of data is handled by **VHERE**, a word that puts the next vacant address in **SUPERSEG** on TOS.

We define **VALLOT** to keep track of the storage used by array definitions. **VALLOT** increments the pointer in **VHERE** (and aborts with a warning if the segment length is exceeded).

We first define some auxiliary words:

```
MEMORY 4+ @  S->D   DCONSTANT  MEM.START   \ beg. of free memory
40.960 DCONSTANT   MAX.PARS                \ 40960 = 655360 /16
: TOTAL.PARS  MAX.PARS   MEM.START  D- ;   \ # pars avail. mem.

1 SEGMENT SUPERSEG           \ define named segment 1 byte long
TOTAL.PARS  DROP  FREE-SIZE  \ tell DOS and HS/FORTH about it

DVARIABLE  VHERE>
: INIT.VHERE>   0.0  VHERE>  D! ;   INIT.VHERE>
: VHERE ( - d.offset)  VHERE>  D@  ;
: D4/    D2/ D2/ ; : D16/   D4/ D4/ ;

: >SEG.OFF  ( d - seg off)
OVER  15 AND  -ROT  ( - off d)    D16/  DROP SWAP ;

: TOO.BIG?     VHERE  >SEG.OFF  0 AND   TOTAL.PARS  D>
               ABORT" INSUFFICIENT ROOM IN SUPERSEG" ;
\ check whether new value of VHERE> passes end of SUPERSEG
```

```
: VALLOT  ( d.#bytes -)    VHERE  D+  VHERE> D!   TOO.BIG? ;

\ Array-defining words
\ Ex:    5Ø LONG REAL 1ARRAY V{
\        V{        ( - adr)
\        V{ 17 }  G@L  ( :: - V[17])
: LONG  DUP ;
FIND D,  Ø=  ?(  : D,  SWAP , , ; )
: 1ARRAY  ( 1 1 t - )   CREATE
   UNDER  , ,                \ t,1 into 1st 4 bytes  ( - 1 t )
   SUPERSEG @  16 M*         \ start of
   SUPERSEG VHERE  D+  D,    \ abs. address into next 4 bytes
   #BYTES M*                 ( - #bytes to allot)
   VALLOT  ;                 \ allot space in the segment
\ run-time action: ( - adr)
```

We also need some words to go with 1ARRAY:

```
: }  ( adr n - seg.off[n] t )
   SWAP   DUP@  >R                                \ type -> rstack
   4+  D@
   ROT  R@  #BYTES  M*  D+ >SEG.OFF R> ;          ( - seg.off[n] t)
```

Finally, here is a useful diagnostic word

```
: ?TYPE   ( t -)     \ it's ok for this to be slow!
   DUP  Ø = IF  DROP  ." REAL*4"    EXIT  THEN
   DUP  1 = IF  DROP  ." REAL*8"    EXIT  THEN
   DUP  2 = IF  DROP  ." COMPLEX"   EXIT  THEN
   DUP  3 = IF  DROP  ." DCOMPLEX"  EXIT  THEN
   ." NOT A DEFINED DATA TYPE" ABORT ;
```

3.5. 2ARRAY and }}

We now want to define arrays of higher dimensionality. For example, to define a 2-dimensional array we say

```
9Ø LONG BY 9Ø WIDE COMPLEX 2ARRAY XA{{
```

This leads to the definitions

```
: BY ;                    \ a do-nothing word for style
: WIDE  *  SWAP ;         ( 1 1 w - 1*w 1 )

: 2ARRAY  ( 1*w 1 t - ) 1ARRAY ;
```

By correct factoring (putting some of the work into LONG and WIDE) we achieved an easy definition of 2ARRAY.

Now let us define }} to fetch the double-indexed address:

```
: }}  ( adr m n - a[m*1+n] t)
   >R  OVER  2+ @        ( - adr m 1*w)
   *   R> + }  ;
```

Again, factoring has let us define }} in terms of }.

3.6. Redefining G@ and G!

We defined G@ and G! using vectored execution. The fetch and store words in the execution tables implicitly assume the variable is stored in the LISTS segment. To address other segments we use the long form [14] of addressing. Thus we would define

```
    G: G@L  R32@L R64@L X@L DX@L ; G: G!L  R32!L R64!L X!L DX!L ;
```

for use with typed scalars and arrays. The *ifstack* words would be redefined accordingly.

4. Tuning for speed

Some of the words in our typed-data/matrix lexicons should be optimized or redefined in machine code. Accessing matrix elements imposes a non-trivial overhead on matrix operations. We can reduce the execution time with inline code, either in the traditional Forth manner via selected assembler definitions, or with a recursive-descent optimizer such as HS/FORTH's [15]. However, the best place to optimize is generally entire inner loops and other selected areas of code, not the access words *per se*. By hand-coding the innermost loop in matrix inversion and FFT routines, I have achieved programs that run in (asymptotically) minimum time on the 8086/8087 chip set.

Significant speed increases in data access could perhaps be achieved using multiple code field (MCF) words, as described by Shaw [16], and as implemented by HS/FORTH in the words VAR, AT, IS, and variants thereof. Thus, an MCF-defined SCALAR – call it X – would fetch itself by invoking its name alone

```
    X    ( :: - x)
```

would have its address placed on the stack by the usage

```
    AT X
```

and would have the proper storage code selected and compiled via

```
    IS X .
```

The disadvantage of MCF style is that compile-time binding, while faster in execution, loses the flexibility of run-time binding. Thus a lexicon would have to be recompiled in order to run it with a data type that would – as with FORTRAN – be specified at compile-time. The run-time binding of the code given above enables a previously compiled routine to handle all four standard scientific data types.

5. Conclusions

Forth is a protean language. This is at once its strength and its weakness. The bulk of scientific programming can be carried out with a few standard data structures. Only venerable FORTRAN, among the more readily available compiled languages, provides the one scientists and engineers find most necessary, complex floating point numbers. This, and the fact that FORTRAN is quasi-algebraic, account for its popularity.

The Hewlett-Packard RPN calculators have weaned many scientists from infix notation; however, the plethora of access commands and arithmetic operators, and the problems of keeping track of data types (demanded by a standard Forth program containing mixed data types) daunts would-be Forth converts. My aim in this article has been to ease this situation with a standard format for typed data, and a standard array notation in Forth. In addition to promoting a more FORTRAN-like programming style (thus easing the transition to Forth) these extensions permit a simple FORmula TRANslator into immediately executable Forth code [1]. A FORTRAN → Forth compiler providing access to the enormous library of tested FORTRAN subroutines then becomes an attractive possibility. My hope is that Forth's great potential for scientific programming might thus be recognized more widely.

References

[1] I have written the germ of one, namely a FORmula TRANslator. See J.V. Noble, JFAR (to be published).

[2] M.G. Kelly and N. Spies, Forth, a Text and Reference, Prentice-Hall, NJ, 1986, p. 324 ff.

[3] ©Harvard Softworks, P.O. Box 69, Springboro, Ohio 45066. Tel: (513) 748-0390.

[4] Dick Pountain, "Object-oriented Forth", *Byte Magazine*, 8/86; *Object-oriented Forth*, Academic Press, Inc., Orlando, 1987.

[5] Leo Brodie, *Thinking FORTH*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1984, p. 118ff.

[6] HS/FORTH uses a word pair `CASE:...;CASE` that performs the same task as `G: ... ;` below. The definition of `G:` was inspired by an article by Michael Ham (*Dr. Dobb's Journal*, October 1986).

[7] J.V. Noble, *Scientific Forth: A New Language for Scientific Computation* (in preparation); "Scientific Computation in Forth," *Computers in Physics*, Sept./Oct. 1989.

[8] Leo Brodie, *Thinking FORTH* (op. cit.), p. 48ff.

[9] Other authors have noted this and proposed more readable matrix notations. See, e.g., "Forth and the Fast Fourier Transform" by Joe Barnhart, *Dr. Dobb's Journal*, September, 1984, p. 34. Also Dick Pountain's book *Object-oriented Forth* (op cit.) uses an array naming convention with brackets.
I have found that scientists and engineers simply will *not* use a native Forth syntax like `m n M`; this has, I feel, been the biggest single obstacle to wider adoption of Forth.

[10] See, e.g., L. Brodie, *Thinking FORTH* (op. cit.) p. 113ff.

[11] See, e.g., Ray Duncan, "Forth support for Intel/Lotus expanded memory," *Dr. Dobb's Journal*, August 1986; also, John A. Lefor and Karen Lund, "Reaching into expanded memory," *PC Tech Journal*, May 1987.

[12] See, e.g., D.N. Jump, *Programmer's Guide to MS-DOS*, rev. edition, Brady Books, New York, 1987.

[13] C. Morgan and M. Waite, *8086/8088 16-bit Microprocessor Primer* (Byte/McGraw-Hill, Peterborough, 1982);
L. Scanlon, *IBM PC & XT Assembly Language: A Guide for Programmers*, Brady/Prentice-Hall, Bowie, Md., 1983;
R. Lafore, *Assembly Language Primer for the IBM PC & XT*. Plume/Waite, New York, 1984.

[14] Consult, e.g., L.J. Scanlon, op. cit.; or R. Lafore, op. cit.
HS/FORTH defines "far" access operators, `@L` and `!L` of all types, that expect a "long" address on the stack. For example,
```
CODE R32@L DS: POP. FWAIT. DS: [BX] DWORD-PTR. FLD. END-CODE
```

[15] J.S. Callahan, *Proc. 1988 Rochester Forth Conference*, Inst. for Applied Forth Research, Inc., 1988, p. 39.

[16] G. Shaw, "Forth Shifts Gears, I," *Computer Language*, May 1988, p. 67; "Forth Shifts Gears, II," *Computer Language*, June 1988 p. 61; *Proc. 9th Asilomar FORML Conference*, JFAR 5 (1988) 347.

Dr. Noble received his B.S. from Caltech in 1962, his M.S. from Princeton in 1963 and his Ph.D. from Princeton in 1966, all in Physics. Beginning with Fortran I in 1960, he has programmed in Basic and Assembler, but almost exclusively in Forth since 1985. His interests include theoretical physics (nuclear, particle and astrophysics), theoretical biology (epidemiology and chaos), and science and public policy. Dr. Noble's major use of Forth is in number crunching.

```
\ TYPED DATA STACK MANAGER  COPYRIGHT NOBLEHOUSE  1989
TASK FSTACKS
DECIMAL

FIND X@  Ø=  ?( FLOAD C:\HSFORTH\COMPLEX.FTH )

Ø CONSTANT REAL              \ define data-type tokens
1 CONSTANT DREAL
2 CONSTANT COMPLEX
3 CONSTANT DCOMPLEX

CREATE <#bytes>  4 C,  8 C,  8 C,  16 C,
: #BYTES   <#bytes> +  C@ ;  ( type - length in bytes)
\ : #BYTES   3 AND   DUP   1 AND   4* 4+   SWAP  2/   SAL  ;
\ definitions for the parallel stack of types
CREATE  TSTACK  82  ALLOT   \ 2 tos-pointer, 2Ø 4-byte cells
HERE  CONSTANT TSTACK>

: TS.INIT   TSTACK  2+  TSTACK  ! ;     TS.INIT

: ?T.OVFLW   ( adr - adr or abort)  DUP  TSTACK>  =
    IF  ."  TSTACK  OVERFLOW"   TS.INIT   ABORT  THEN  ;
: >T   ( adr t - )  4 TSTACK  +!   TSTACK  @  ?T.OVFLW   D! ;
: T.TYPE!   TSTACK @  2+  ! ;
: ?T.UFLOW   ( adr -)    DUP@  2-  >=
     IF  ." EMPTY TSTACK"   TS.INIT   ABORT  THEN  ;
: T@   ( - adr t)   TSTACK  @   D@  ;
: T.DROP   TSTACK  ?T.UFLOW   -4  TSTACK  +!  ;
: T>   ( - adr t)   T@   T.DROP  ;
: TSWAP   T>  T>  DSWAP  >T  >T  ;

\ definition of typed variables
: SCALAR  ( type - )   CREATE  DUP ,   #BYTES  ALLOT
    DOES>  DUP@   SWAP  2+  SWAP ;
: SCALARS   ( n t - )  SWAP  Ø DO   DUP   SCALAR   LOOP  DROP ;
```

```
\ definitions for a multi-type fstack

CREATE FHEAP    8Ø  8 *  ALLOT
HERE CONSTANT  FHEAP>

: FS.INIT   TS.INIT   FHEAP  Ø   TSTACK  2+  D!  ;   FS.INIT

: FS.ABORT   FS.INIT  ABORT  ;

: ?FS.OVFLW   ( fsp – fsp | abort on overflow)    DUP  FHEAP>  =
    IF  ." FSTACK  OVERFLOW"   FS.ABORT    THEN  ;

: >FS    ( adr[x] t[x] – :: – x )
    DUP   T.TYPE!                           \ put type on tstack
    #BYTES                                  ( – src #bytes)
    T@  DROP                                ( – src #bytes old.fsp)
    DDUP +   ?FS.OVFLW                      ( – adr #bytes old.fsp new.fsp)
    Ø  >T   SWAP                            ( – src dest n)
    CMOVE  ;                                \ put number on fstack

: FS>    ( adr t – )   T.DROP   T@    ( – t fsp t')
     ROT  OVER    <>             \ check that dest has same type
     IF  ."  ATTEMPT TO STORE TO WRONG DATA TYPE"  FS.ABORT   THEN
     #BYTES   ( – dest src n)   ROT  SWAP  ( – src dest n)    CMOVE ;

: FS.DUMP   TSTACK  ?T.UFLOW    TSTACK @    TSTACK 2+
            DO    I D@   DUP
            BEGIN-CASE  Ø  CASE-OF  CR .  R32@    F.   ELSE
                        1  CASE-OF  CR .  R64@    F.   ELSE
                        2  CASE-OF  CR .  X@      X.   ELSE
                        3  CASE-OF  CR .  DX@     X.   ELSE
                        DDROP  ABORT"  BAD DATA TYPE"  END-CASE
            4  +LOOP  ;
DCOMPLEX SCALAR TEMPØ
DCOMPLEX SCALAR TEMP1
: FS.SWAP  TSTACK DUP ?T.UFLOW @  2- @
     [ ' TEMPØ ] LITERAL !
     TEMPØ  FS>
     TSTACK DUP ?T.UFLOW @  2- @
     [ ' TEMP1 ] LITERAL !     TEMP1  FS>
     TEMPØ  >FS     TEMP1  >FS ;
CASE: G!   R32!  R64!   X!   DX!  ;CASE
CASE: G@   R32@  R64@   X@   DX@  ;CASE
\ CASE: ... ;CASE  define unary operators

: FS>F     ( – t :: x – )   T.DROP  T@   UNDER  G@ ;
: F>FS     ( t – :: – x )   T.TYPE!  T@   DDUP  G!
      ( – old.fsp t)  #BYTES   +  ( – new.fsp)  ?FS.OVFLW  ( – new.fsp)
      Ø >T  ;

: NEW.TYPE   DDUP  AND   1 AND   -ROT   OR   2 AND   +  ;
\ CODE NEW.TYPE   AX POP.   CX AX MOV.   CX BX  AND.  CX  1 IW  AND.
\      BX AX  OR.   BX 2 IW AND.  BX CX ADD.  END-CODE
( t1 t2 – t')
```

```
: WHICH.OP    ( t1 t2 - n)    2/ SWAP    2 AND    +  ;
  CODE  WHICH.OP    AX POP.  BX 1 SHR.  AX  2 IW AND.  BX AX ADD.  END-CODE

: GB:  CREATE  ]  DOES>   FS>F    FS>F    ( - adr t1 t2)   \ get args
                          DDUP  NEW.TYPE  >R               \ compute new type
                          ( - adr t1 t2)
                          WHICH.OP  2*  +  EXECUTE@        \ compute new value
                          R>    F>FS  ;                    \ put it away
\ defining word for binary operators leaving 1 argument

\ test example
: F*X   F-ROT  X*F ;

GB: G*   F*  F*X  X*F  X*  ;

\ LARGE ARRAY PACKAGE
TASK MATRIX

MEMORY 4+ @  S->D   DCONSTANT  MEM.START     \ beg. of free memory
40.9600 DCONSTANT   MAX.PARS                 \ 409600 = 6553600 /16
: TOTAL.PARS   MAX.PARS   MEM.START  D- ;    \ # pars avail. mem.

1 SEGMENT SUPERSEG                  \ define named segment 1 byte long

TOTAL.PARS  DROP  1-  FREE-SIZE  \ tell DOS and HS/FORTH about it

DVARIABLE  VHERE>
: INIT.VHERE>   0.0  VHERE>  D! ;   INIT.VHERE>
: VHERE   ( - d.offset)  VHERE>  D@  ;

: D4/    D2/ D2/ ;
: D16/   D4/ D4/ ;

: >SEG.OFF  ( d - seg off)
    OVER  15 AND  -ROT   ( - off d)    D16/  DROP  SWAP ;

: TOO.BIG?    VHERE  >SEG.OFF  0 AND   TOTAL.PARS  D>
              ABORT" INSUFFICIENT ROOM IN SUPERSEG" ;
\ check whether new value of VHERE> passes end of SUPERSEG

: VALLOT  ( d.#bytes -)    VHERE  D+   VHERE> D!   TOO.BIG? ;
```

```
\ Array-defining words
\ Ex:    5Ø LONG REAL 1ARRAY V{
\        V{       ( - adr)
\        V{ 17 }  G@L  ( :: - V[17])
: LONG   DUP ;

FIND D,  Ø=  ?(  : D,  SWAP , , ; )
CREATE <#bytes> 4 C, 8 C, 8 C, 16 C, OKLW
: #BYTES  ( t - #bytes)  <#bytes> + C@ ;

: 1ARRAY  ( 1 l t - )   CREATE
    UNDER  , ,               \ t,l into 1st 4 bytes  ( - l t )
    SUPERSEG @  16 M*        \ start of SUPERSEG
    VHERE  D+  D,            \ abs. address into next 4 bytes
    #BYTES  M*               ( - #bytes to allot)
    VALLOT  OKLW  ;          \ allot space in the segment
\ run-time action: ( - adr)
: }  ( adr n - seg.off[n] t )
    SWAP   DUP@  >R                               \ type -> rstack
    4+  D@
    ROT  R@  #BYTES  M*  D+  >SEG.OFF  R> ;       ( - seg.off[n] t)

: ?TYPE  ( t -)    \ it's ok for this to be slow!
     DUP  Ø = IF  DROP  ." REAL*4"   EXIT  THEN
     DUP  1 = IF  DROP  ." REAL*8"   EXIT  THEN
     DUP  2 = IF  DROP  ." COMPLEX"  EXIT  THEN
     DUP  3 = IF  DROP  ." DCOMPLEX" EXIT  THEN
     ." NOT A DEFINED DATA TYPE"  ABORT ;
\ 2ARRAY and }}

\ Ex.  9Ø LONG BY 9Ø WIDE COMPLEX 2ARRAY XA{{

: BY ;                   \ a do-nothing word for style
: WIDE  *  SWAP  ;       ( 1 l w - l*w l )

: 2ARRAY  ( l*w l t - )  1ARRAY ;

: }}  ( adr m n - a[m*l+n] t)
    >R  OVER  2+ @      ( - adr m l*w)
    *   R> +   }  ;
```