HANDLING MULTIPLE DATA TYPES IN FORTH

by John J Wavrik
Department of Mathematics
Univ of Calif - San Diego

ABSTRACT

While originally developed for use in hardware control, Forth is, because of its flexibility, an attractive language for developing software systems. In computer algebra, for example, it provides a programmer with an unrestricted ability to choose the representation of a large variety of data structures and the words needed to manipulate them. This paper deals with the problem of using many data types while preserving a simple Forth syntax. We introduce a mechanism for managing the storage of intermediate results of computation. This package takes advantage of special features of Forth to handle the recovery of unused storage.

INTRODUCTION

Forth systems provide certain primary data types (integers, double precision integers, and perhaps floating point numbers). It is often useful to compute with more complicated things: arrays of various types, records in a database, strings, etc. Some languages have built-in procedures for constructing certain types of complex structures but limit what can be built by limiting the tools. Forth instead provides the user with a toolkit to represent virtually any type of data and to add words to manipulate it. In some application areas this flexibility is enormously advantageous.

A single application in computer algebra may require not only several primary data types, but also types created by the user and types created by compounding several system and user types. In a recent application, for example, the following data types were used (in addition to integer and double precision integer): (1) extended precision integers up to about 100 digits; (2) extended precision rational numbers; (3) strings; (4) the integers modulo a prime; (5) polynomials whose coefficients are arbitrary precision rational numbers and whose monomial parts are strings; (6) polynomials whose coefficients are integers modulo p; (7) matrices whose entries are integers modulo p.

A major problem facing a worker in a field like "modern" algebra is to provide a means for creating a large and varied collection of new data types and integrating them into a system without making the system more complicated, harder to use, or less efficient. This paper will discuss the problem and some solutions.

We will use the word "object" in the broad sense of something we wish to manipulate. "Objects" could be strings, arrays of integers, records, etc. They usually come in classes of similar types with an appropriate set of operations. The user will often produce new defining words to create objects in a given class.

While creating a new class of objects, the Forth programmer is aware of how they are represented in memory. While defining the operations, the details of this representation are needed. There is an enormous psychological (as well as portability) advantage to then be able to suppress these details and think in terms of a collection of things with certain operations. Some may see in this the flavor of "object-oriented programming". It is actually the way that Forth programmers have been building their applications for years. In Forth, however, we are free to return to the implementation details whenever necessary—they are not hidden or obliterated -- they are forgotten, but not gone.

After we generate these objects, we'd prefer to be able to use the same syntax as used to manipulate other Forth objects:

(1) If **A** and **B** are names for strings, **$+** is concatenation, **$.** is the word for printing strings

Then **A B $+ $.** should concatenate **A** and **B** and print the result

(2) If **A** and **B** are matrices, **C*Mat** has the stack diagram **(c M—M')** and multiplies a matrix by a constant

Then we should be able to get **(n m A B—nA + mB)** by
ROT SWAP C*Mat >R C*Mat R> Mat+

This means that we must decide how the Forth parameter stack should be used with the new objects. We will examine several possibilities.

1. Placing the actual data for objects of varying size on the Forth parameter stack requires a new set of data manipulation words for each data type. Suppose for example, we put strings on the stack (in the form of a collection of characters together with a count). Strings occupy a different amount of space on the stack than do integers—what is worse, the amount of space depends on the string. So we must create special words to perform stack manipulations on strings (e.g. **$SWAP, $DUP**, etc.). Now we face the problem that we might want to swap a string on top with an integer below. Neither **$SWAP** nor **SWAP** will do what we want—this requires a mixed operation like **I$SWAP**. If double precision integers are used, we should also provide mixed operations to interchange various combinations of strings and double precisions. The data types mentioned above in connection with computer algebra occupy from 2 to several hundred bytes each. Some are of variable size. Imagine the complexity if we want to add 7 new data types: we would need 9 versions of **DUP**, 81 versions of **SWAP**, 81 versions of **OVER** and 729 versions of **ROT**.

2. Another solution is to equip each new data type with its own stack. This would still require a new set of data manipulation words for each data type, but mixed stack operations are not needed. On the other hand, it exacts a psychological penality of requiring the programmer to keep in mind the movement of several stacks all the time.

There is a very important truth that is ignored by both of these possibilities: if bulky data is put on a stack, the first step in using it will usually be to take it off! These approaches can therefore be quite inefficient.

3. The manipulation of data of varying size on a single stack would seem to require a typed stack. The stack would not only include an object, but some information about how many bytes it takes (and maybe even something which indicates what kind of object it is). No new stack operations would be required, but the stack operations would use the size information to perform their function (e.g. **SWAP** would know that it is swapping something 2 bytes in size with something 30 bytes in size). Something like this is already done by those who use a string stack: **$SWAP** does use the sizes of the strings. It is also used in the Forth-like page description language, PostScript, found in some laser printers. The inclusion of information about the type or size of each item on the parameter stack would, however, require a modification of the underlying Forth system. Even integers would need a size indicator to be put on the stack. Size information on an interactive system is usually available only at run time, and so this approach would exact some penalty in execution time. We will therefore consider a solution which is more efficient and requires no modification to Forth.

POINTERS AND STORAGE STRUCTURES

We begin by representing data on the stack by pointers. The data sits somewhere in memory—only its address is put on the stack. The addresses are stack elements, so they can be manipulated using the usual stack words. A programmer can just as well think "X is on the stack" whether it is the data or just the address of X that is actually on the stack—so this does not make conceptualizing any more difficult. The use of pointers solves the problem with stack operations discussed previously. But it does yet quite reach the goal of allowing the use of customary Forth syntax. We can see why by looking at how this works in practice.

Let's look at the simplest way to represent polynomials. A polynomial in one variable is an expression of the form:

$$a_0 + a_1x + a_2x^2 + ... + a_nx^n$$

(particular examples are $1 - x^2$ and $1 + x + 3x^3$ -- we usually don't write any term that has a zero coefficient). We will assume that the coefficients a_i are integers. The highest power that actually occurs in a polynomial is its degree. The main things of importance in describing a polynomial are the array of coefficients and the degree. To make everything as simple and efficient as possible, we will allow the same amount of space for each polynomial. Let us assume that we will not need to deal with polynomials of degree bigger than 50. Each polynomial will then look like this in memory:

$$\text{deg} \mid a_0 \mid a_1 \mid .... \mid a_{50}$$

On a 16-bit system, there is a 2-byte slot for the degree followed by 51 2-byte slots for the coefficients. Each polynomial occupies 104 bytes.

We now introduce a defining word **POLY**. **POLY F** will make a dictionary entry for a new polynomial called **F** and set aside storage for it. When **F** executes, it puts the address of its storage block on the stack.

```
50 CONSTANT MAXDEG
: POLY   CREATE  MAXDEG 2+ CELLS ALLOT  ;
```

(CELLS is found in the proposed new ANSI Standard, it converts cells to bytes. On a 16-bit system it is 2*, on a 32-bit system it is 4 *)

We have, now a way of producing a block of storage that has a name. When we use that name, the address of the block is put on the stack. What makes the block of storage a polynomial is the additional words we define to act on it. We need words to access the degree and coefficients. Words for input and output. Words to implement the arithmetic (using the methods taught in high school), etc. For example we define the degree function, for our representation, by **: DEG @ ;**. Eventually we reach the point where all the important features of a polynomial have been defined in terms of our representation—and so we forget about the details of representation. When we want the degree of **F** we say **F DEG** thinking "**F** is on the stack, and we apply **DEG** to it".

A problem arises when we we perform an operation which has a polynomial result. We do not have a way to produce a proper size block of storage for an object except to give it a name. So we find ourselves forced to give the system a name for the place where the results are to be put. This leads to the syntax **A B C op** for binary operations (it means that **op** is applied to **A** and **B** with the result stored in **C**). There is no problem, by the way, with efficiency or speed in using this syntax for operations. In many cases code is simpler to write and runs more quickly if the operation word is explicitly told where the results should go. The issue is that we don't usually write Forth this way.

The problem cannot be solved just by producing a single temporary location where all operations put their results: Suppose we assume two operations **o+** and **o*** on the objects and we want to perform **(A o* B) o+ (C o* D)**. It is clear that the sum cannot be performed until the two products are performed. To do this, the programmer must create two intermediate objects to store the results of the multiplications. Thus the computation would look like this:

```
A B T1 o*   C D T2 o*   T1 T2 Result o+
```

T1 and **T2** are "temporaries". They are pointers to blocks of memory only used to hold intermediate results. After the calculation is over, there is no interest in what they contain—and they can (and must) be reused. At the moment we have placed on the programmer the burden of explicitly managing temporaries. He must know which of them are used by what words. It should be mentioned, by the way, that temporaries are a feature of any kind of chain computation with any kind of data. Forth programmers are not usually aware of temporaries because the stack provides a mechanism for automatically dealing with them for integer computations.

A satisfactory solution to our problem involves producing an automatic mechanism for temporaries which works in conjunction with the Forth parameter stack. We will set aside an area in memory to hold the

data for intermediate results. A mechanism is provided to supply the address of an unused part of this block whenever space for a temporary is needed. We call this a storage structure.

Each data type (other than the primary types) will have its own storage structure, and each storage structure will manage a pool of temporary locations for that data type (in the code below, the storage structures each have a pool of 16 addresses). When we perform an operation, say **OP+**, on two objects **A** and **B** the code for **OP+** will ask for storage in which to put its result. The storage structure will find the next available storage location and put its address on the stack (in the process, that location will be marked as being in use). **OP+** will put its result in that location—and the address will be left on the stack. As a result, **OP+** will now follow the Forth conventions for operations.

After performing 16 (or how many ever storage slots have been set aside) operations, the storage structure finds that all of its locations have been marked. It now must find out which of these are still in use. It does so by examining the parameter stack. Any address still on the parameter stack is regarded as in use (and is marked) all other addresses are unmarked. [In thecode below, there is a provision for the programmer to add additional places for the system to check before it declares an address to be available.]

Since the number of addresses and the number of places to search are limited, this garbage-collection process is very fast. Before going into more detail on the implementation and use of this mechanism, some things should be said about the number 16. This storage mechanism has been used with objects that require several hundred bytes of memory each—so the author was anxious not to tie up more memory than necessary for temporaries. How many temporary locations are needed to perform calculations? The stack in Hewlett-Packard calculators provides 4 -- HP found that any common arithmetic computation could be performed in a way that requires no more than this. Tests have shown that in typical mathematics applications the Forth parameter stack does not become deeper than 20-25 (cells), and the number of active temporaries for a particular data type does not exceed 4. The code is implemented so that an error message and ABORT occur if a storage structure runs out of available locations. The only time we have seen the error message, while using the mechanism for two years, is in response to program bugs. Thus, for a language like Forth which does not use recursion as a primary control mechanism, 4 temporary locations is enough and 16 is more than enough. Those who work with truly massive objects might want to use a smaller number.

We distinguish the "storage structure", which is a scheme for managing a block of storage, from the block of storage it manages. A new storage structure is introduced by the defining word **TEMP-STORAGE**. Each child has its own array **ADDRESS** (containing the addresses of the storage it manages); an integer **MARKS** (a bit map showing which of its addresses are in use); and a variable **ELSEWHERE** (which contains the execution address of an alternate search routine). The most important word used in programming is the word **TEMP** which gives the address of the next available temporary storage location and initiates a garbage collection when necessary.

In keeping with the spirit of Forth's **CREATE .. DOES>** mechanism, the exact same code is used by all the children. The actual action of each child is to put the address of its data in a specific place. All of the words in the package use this to perform their actions. Thus the child provides a referencing environment for a set of shared words and the syntax makes the child's name appear as an adjective which modifies the scope of the following shared word. If **STRINGS** and **LISTS** are the names of children, **STRINGS TEMP** will return the next temporary location belonging to **STRINGS** and **LISTS TEMP** will return the next temporary location belonging to **LISTS**.

ILLUSTRATIONS

There are two steps involved in using this mechanism. Suppose we want to create a storage structure for polynomials:

(1) **TEMP-STORAGE POLY-TEMPS** is used to defined the storage structure

(2) Storage for the temporaries must be set aside and the application must install the addresses in the array **ADDRESS**

We assume that each polynomial occupies a fixed number, **PSIZE**, of bytes. The following code carries out these two steps:

```
TEMP-STORAGE POLY-TEMPS
CREATE PPOOL   PSIZE 16 * ALLOT
: PSETUP  POLY-TEMPS
          PPOOL 16 0 DO    DUP I ADDRESS ! PSIZE +
                     LOOP  DROP ;   PSETUP
```

For convenience, we introduce the word **PTEMP** to get the address of a polynomial storage location:

```
: PTEMP   POLY-TEMPS TEMP  ;
```

Here is the code for the addition of polynomials. It uses the following words (part of the polynomial package from which this example is taken). It is designed to work with any type of coefficients.

```
DEG     ( poly—n )      n is the degree of poly
DEG!    ( n poly -- )      store n as the new degree of poly
COEFF   ( n poly—c )    return the coefficient of x^n
COEFF!  ( c n poly -- )    make c the new coefficient of x^n
SETDEG  ( poly -- )        recalculate the degree of poly
C+      ( c1 c2 -- c3 )    addition of coefficients
CONS P1  CONS P2  CONS RESULT : GETRES  PTEMP IS RESULT  ;
: P+   ( poly1 poly2 --  sum )  GETRES  IS P2  IS P1
     P1 DEG  P2 DEG  MAX  DUP RESULT DEG!
     1+ 0 DO   I P1 COEFF   I P2 COEFF   C+
               I RESULT COEFF!
          LOOP   RESULT SETDEG  RESULT  ;
```

NB Initializable constants were used for a clean appearance. Forth standards have never fixed a name for these. Most Forth systems provide them using different names. F-83 allows the value of regular constants to be changed using **IS**. **CONS** is a catch-all defining word.

It is not the purpose of the illustration to get us involved in polynomials and their arithmetic. Rather it is to show that new data objects have been treated on the stack using the same kind of writing and thinking that is used with the primary types—without changing any of the basic Forth operators or adding complicated new ones. The addition operator for polynomials can be used in exactly the same way as the addition operation for integers.

There are only two things to remember when using storage structures:

(1) The 16 locations are intended only for temporarily storing intermediate results. Any data to be saved for future use should be copied to a permanent home.

(2) For speed the garbage-collector only checks the parameter stack (but see below) to find if an address is still in use. Make sure active addresses are not hidden from view when the garbage is collected.

The second point does not prevent the use of variables. Notice that the example above shows all the addresses being removed from the stack and put in constants. The only time a garbage collection could occur is when a temporary was requested at the beginning of the program. We therefore left the two original

polynomials on the stack until after the temporary was obtained. Purists who do everything with stack manipulations will have no problems.

SPECIAL SITUATIONS

Most data types use just the simple treatment outlined above and therefore only the words **TEMP-STORAGE** and **TEMP**. The next two sections explain how some other words in the package can be used to handle special situations.

The two situations are:

(1) The user wants the garbage collector to do something in addition to finding free addresses.

(2) The user wants to remove addresses from the parameter stack but still keep them visible to the garbage collector.

We illustrate by examples from an application in which they have been used. Underlying this example is a package which implements circular lists. It has mechanisms for managing the nodes of lists. The details are not needed here. We only need to know that lists are referenced by a pointer to the head node; that **<ptr> XLIST** will return the nodes of the list to the storage pool; and that this occurs automatically whenever **<ptr>** is reassigned.

The application was designed for the segmented architecture of the 80x86. The pointers all reside in a separate "pointer segment" and they place on the stack an offset in this segment. The pointers appear to the Forth system as ordinary integers—the segment manipulation is hidden in the words which handle the pointers. If the same application is used on a 32-bit system the pointers are offsets into a reserved block within the dictionary. The word **PTX** is a variable containing the offset of the last pointer defined.

```
TEMP-STORAGE LISTS
: LTEMP-SETUP  LISTS
16 0 DO  CELL PTX +! PTX @  I ADDRESS !  LOOP  ;
LTEMP-SETUP
```

As before this produces a new storage structure and fills in the addresses.

(1) The pointers which we are managing here each will point to a list of nodes. The circular list package will automatically return all the nodes attached to a pointer at the point when the pointer is to be reused. The programmer can also force the return by using the word XLIST. If very large lists are used, it can be useful to have our garbage collector do an XLIST to free them up when it finds that a pointer is not being used.

There are two different storage management schemes being used to manage two different kinds of things, but the situation is not as complicated as it might first seem. Here is an analogy: There is a large box of pop-beads (the kind that can be pushed together to form chains). Children in a school class are allowed to take beads from the box as they need them to make chains. There is a class rule that when a child finishes building one thing and starts another, all the old beads are first put back in the box. If the box becomes empty it means that all of the beads are being used—so construction must stop. Now suppose we introduce a new possibility. Suppose children have other things they could do besides work with the beads. It could happen that several children have become interested in something else but still have long strings of beads in their hands. Our storage structure is like a school teacher who periodically checks to find out which children are still busy. Her job is to deal with children, not with beads. We would like to add to her duties and have her tell children who are not currently making chains to return their beads to the box.

Our storage structure application provides several words in addition to those we have used before. One of these is **UNMARKED** which puts on the stack a list of the addresses currently unmarked, together with the number of such addresses. We use it to find which addresses are not in use. Another is **GET-NEXT** which gives the next unmarked address (or 0 if there is none). It is quite easy to define a word **RECLAIM-TEMPS** to return nodes attached to inactive pointers after a garbage collection.

```
: RECLAIM-TEMPS   LISTS COLLECT-GARBAGE  UNMARKED
      ?DUP IF  0 DO  XLIST  LOOP  THEN  ;
```

We now define the word **LTEMP** to use **RECLAIM-TEMPS** instead of ordinary garbage collection (and use it every time we need a temporary list).

```
: LTEMP ( -- addr )  LISTS GET-NEXT  ?DUP 0=
IF  RECLAIM-TEMPS  GET-NEXT ?DUP 0=
ABORT" storage exhausted "   THEN  ;
```

(2) Typically an algorithm involving circular lists starts a pointer at the head of the list and moves it to each node in succession. It is natural, then, to put the address of a list in a variable as the first step in these algorithms. In keeping with our previous points, however, putting the address of the list in a variable will remove the address from the stack—and so the garbage collection will mark it as unused. Since artificially keeping these addresses on the stack seemed unnatural, we use an extension of the storage structure mechanism to provide the garbage collector with other places to look for active addresses.

A normal assignment uses **!** to store an address in a variable. This removes the address from the stack. Here we use a new operator for assignment **!!**. At compile time a count is kept of the number of such assignments—and a runtime action word is compiled. The runtime word, **(!!)**, saves the address of the head node on an extra stack and also assigns it to the moving pointer. A word **RELEASE** is compiled at the end of the word to pop the appropriate number of addresses from the extra stack. [We have redefined the semicolon to compile **RELEASE**. Those who object to redefining a system word might want to use another name.]

```
VARIABLE LCNT
CREATE LSTACK  100 ALLOT           VARIABLE LPTR
: SET-LSTACK   LSTACK LPTR !  0 LCNT !  ;    SET-LSTACK
: >L           LPTR @ !  1 CELLS  LPTR +!  ;
: L>           1 CELLS NEGATE  LPTR +!  LPTR @ @  ;
: RELEASE      CELLS NEGATE  LPTR +!  ;
: (!!)         SWAP DUP >L POINT ;
: !!           COMPILE (!!)  1 LCNT +!  ;  IMMEDIATE
: ;  LCNT @ ?DUP  IF    [COMPILE] LITERAL COMPILE RELEASE
0 LCNT !
THEN  [COMPILE] ;  ; IMMEDIATE
```

POINT is a word from the circular lists package -- **<ptr1> <ptr2> POINT** will make **<ptr1>** point to where **<ptr2>** now points.

Anything we store using **!!** will now be saved on the **LSTACK**—so we need only tell the garbage collection to look at the **LSTACK** too before deciding that an address is inactive. The word **LSTACK?** returns **TRUE** if the given address is in the **LSTACK** and **FALSE** if not.

```
: LSTACK?  ( addr—f )
FALSE  LPTR @ LSTACK
?DO  OVER  I @  =
IF  DROP TRUE LEAVE THEN
CELL +LOOP  SWAP DROP  ;
LISTS ' LSTACK? >ELSEWHERE
```

The word **LISTS** makes this the current storage structure.
The word **>ELSEWHERE (cfa --)** installs this new search routine.

Any word used for installation in **ELSEWHERE** must take an address from the stack and return a flag (T=address was found). The default is the word **NO-ELSE** which drops the address and returns **FALSE**.

Here is a definition where this idea is used. It should be clear, without knowing what it does, that **!!** is being used in the same way as **!**. What is going on is that we have a double loop which touches all

possible combinations of a node in the first list and a node in the second (and, in each case, does some processing—the nature of which need not concern us). To do this we run a pointer (called **AF**) along the first list and a pointer (called **BF**) along the second.

The pointers which run along lists in this way are essentially variables that hold the address of a node being pointed to. The main thing to be understood is why **!!** is used rather than **!** to store the initial values of these pointers. Some of the lists could be temporaries and garbage collection could occur in the middle of this algorithm. If we remove the addresses from the parameter stack and store them, the garbage collector would believe we are no longer using them. The word **!!** not only stores the addresses in the variables, but it also puts them in a place where the garbage collector will see them. Thus **!!** is used whenever we store an address that we want to keep safe.

```
: MULTIN   ( <A><B><C> --    replace C by C + A * B )
           CF !!   BF !!   AF !!
        BEGIN  BF ADVANCE BF HEAD? NOT WHILE
             BEGIN  AF ADVANCE AF HEAD? NOT WHILE
                  FORM.PRODUCT
                  FIND.PLACE
             REPEAT
        REPEAT     ;
```

One point should be made about the use of auxilliary stacks in Forth. In cases where computations are terminated by an error or a keyboard interrupt, Forth's parameter and return stacks are reset—but user defined stacks are not. Many Forth implementations vector their error handling words allowing users to add extra things. The word **SET-LSTACK** should be inserted in the chain.

SUMMARY

This paper explores the problem of adding new data types to a Forth system (as part of an application) without making the syntax and semantics complicated and difficult to use. It presents a very compact and efficient mechanism for managing the storage needed to store the intermediate results of calculation. It was designed particularly for use with data types whose representations require a great deal of memory, yet it has also been found useful in just adding a string package to Forth.

GLOSSARY OF MAJOR WORDS

>ELSEWHERE (addr --)

Install an extended search routine in the current temporary storage structure. An example is given in the discussion below.

(1) In the high level version, the search routine should take the address of a temporary from the stack and return a true flag (-1) if the address is found in the search. The address (above) is the execution address of this search routine.

(2) In the F83 assembly language version, the search routine is a subroutine (defined by LABEL ending with RET). It searches for an address contained in the AX register and sets the zero flag if it is found.

ADDRESS (ind—addr)

Given an index, ind, between 0 and 15 this returns the address where the corresponding temporary pointer is stored. **3 ADDRESS @**, for example, is used to get the temporary location of index 3. Pointers must be stored in this array as part of the initialization.

TEMP (-- addr)

Return the next available temporary address in the current storage structure (a garbage collection is initiated if no address is immediately available and execution aborts if no address is freed by the garbage collection). Since the storage structure must be made current, it is best to define a word specific to each data type (e.g **: LTEMP LISTS TEMP ;** defines **LTEMP** to give the next address in the structure called **LISTS**.)

TEMP-STORAGE

Defining word for a temporary structure (usage: **TEMP-STORAGE LISTS** defines a temporary structure called **LISTS**. When **LISTS** executes it is made the current structure.) To use the structure the collection of addresses in the array **ADDRESS** must be filled in (and an optional extended search routine installed).

BIOGRAPHY

John Wavrik is an Associate Professor of Mathematics at the University of California - San Diego. He received his Ph.D. in 1966 from Stanford University in Several Complex Variables. His current work uses Forth to implement special purpose software systems for work in abstract algebra and allied fields. He also teaches a course in computer algebra and the Forth language.

LISTINGS

Blocks 1-4 Low level version using F-83 assembler
Blocks 6-9 High level version using Forth-83 standard code

```
Blk # 1                          File: tempsto.blk
0.  \   Temporary Storage  lo level                    29Sep88jjw
1.    LABEL  NO-OP  RET  C;
2.
3.  VARIABLE 'MARKS      ( holds base address of temp type )
4.  VARIABLE ELSEWHERE   ( subroutine—search other than stack )
5.
6.  : TEMP-STORAGE   CREATE  0 , ( marks )  NO-OP ,  ( extended )
7.         HERE  16 CELLS DUP ALLOT  ERASE  ( addresses )
8.         DOES>  DUP 'MARKS !  CELL+ @  ELSEWHERE !  ;
9.
10.  :  MARKS                   'MARKS @  ;
11.  :  ADDRESS  ( indx—addr )  2+ CELLS  MARKS +  ;
12.
13.   ( ADDRESS is an array with indices 0-15. The application
14.   must fill in specific addresses from a storage area )
15.   -->
```

```
Blk # 2                          File: tempsto.blk
0.  \   Temporary Storage  lo level                    29Sep88jjw
1.
2.  : >ELSEWHERE    MARKS CELL+ !  ;
3.
4.    LABEL  INSTACK?  CX PUSH DI PUSH  SP0 #) CX MOV  SP DI MOV
5.    6 # DI ADD  DI CX SUB  CX SHR  REPNZ SCAS
6.    0<> IF ELSEWHERE S#) CALL THEN  DI POP CX POP RET C;
7.
8.    CODE COLLECT-GARBAGE  'MARKS #) DI MOV  0 # 0 [DI] MOV
9.     1 # DX MOV  16 # CX MOV   SI PUSH  DI SI MOV  4 # SI ADD
```

```
10.   HERE  AX LODS  INSTACK? #) CALL
11.         0= IF  DX 0 [DI] OR  THEN  DX SHL
12.   LOOP  SI POP  NEXT C;
13.
14.
15.                                                              -->


Blk # 3                         File: tempsto.blk
0.  \   Temporary Storage  lo level                    11Feb88jjw
1.
2.  CODE GET-NEXT ( -- addr or 0 ) 'MARKS #) DI MOV
3.    0 [DI] BX MOV  BX AX MOV  AX INC  0<>
4.    IF  DI PUSH  1 # AX MOV   4 # DI ADD
5.        BEGIN    AX BX TEST  0<>
6.        WHILE    AX SHL  2 # DI ADD      REPEAT
7.        BX POP  AX 0 [BX] OR  0 [DI] AX MOV
8.    THEN  1PUSH  C;
9.    : TEMP ( -- addr  )  GET-NEXT  ?DUP 0=
10.   IF  COLLECT-GARBAGE  GET-NEXT ?DUP 0=
11.       IF ." storage exhausted "  ABORT THEN  THEN  ;
12.
13.   ( "<name> TEMP"  gives next available address. Best
14.   to define, e.g.  LTEMP as LISTS TEMP )
15.   -->

Blk # 4                         File: tempsto.blk
0.  \   Temporary Storage  lo level                    11Feb88jjw
1.
2.  CODE MARKED ( -- A1 .. Ak k )  0 # CX MOV  1 # AX MOV
3.    'MARKS #) DI MOV 0 [DI] BX MOV  2 # DI ADD
4.    BEGIN  DI INC DI INC  AX BX TEST
5.           0<> IF 0 [DI] PUSH CX INC THEN  AX AX ADD
6. 0= UNTIL CX PUSH NEXT C;
7.
8.  CODE UNMARKED ( -- A1 .. Ak k )  0 # CX MOV  1 # AX MOV
9.      'MARKS #) DI MOV 0 [DI] BX MOV  2 # DI ADD
10.      BEGIN  DI INC DI INC  AX BX TEST
11.          0=  IF 0 [DI] PUSH CX INC THEN  AX AX ADD
12.   0= UNTIL CX PUSH NEXT C;
13.
14.   \ These are used if the addresses point to storage to be
15.   \ reclaimed. Marked addresses are still in use.

       ============================================================

Blk # 6                         File: tempsto.blk
0.  \   Temporary Storage  hi level                    29Sep88jjw
1.
2.  : NO-ELSE  DROP FALSE ;   ' NO-ELSE CONSTANT NO-OP
3.
4.  VARIABLE 'MARKS      ( holds base address of temp type )
5.  VARIABLE ELSEWHERE   ( exec addr  -- search other than stack )
6.
7.  : TEMP-STORAGE   CREATE  0 , ( marks )  NO-OP ,  ( extended )
8.           HERE  16 CELLS DUP ALLOT  ERASE  ( addresses )
9.           DOES>  DUP 'MARKS !  CELL+ @  ELSEWHERE !  ;
10.
```

```
11.  :  MARKS                    'MARKS @  ;
12.  :  ADDRESS  ( indx—addr )  2+ CELLS  MARKS +  ;
13.   ( ADDRESS is an array with indices 0-15. The application
14.   must fill in specific addresses from a storage area )
15.   -->


Blk # 7                         File: tempsto.blk
0.  \   Temporary Storage  hi level                    11Feb88jjw
1.
2.  : >ELSEWHERE    MARKS CELL+ !  ;    QUAN ADR  VARIABLE MSK
3.
4.  : INSTACK?   ( addr --  f )  IS ADR   0  DEPTH 1
5.     ?DO  I PICK  ADR =  IF  DROP -1 LEAVE  THEN  LOOP
6.     DUP NOT IF DROP ADR ELSEWHERE @ EXECUTE THEN  ;
7.
8.
9.  : COLLECT-GARBAGE   0 MARKS !   1 MSK !  ( bit mask )
10.      16 0  DO  I ADDRESS @  INSTACK?
11.                IF  MARKS @  MSK @  OR  MARKS !  THEN
12.                MSK @ 2*  MSK !
13.            LOOP  ;
14.
15.   -->

Blk # 8                         File: tempsto.blk
0.  \   Temporary Storage  hi level                    19Aug88jjw
1.
2. : GET-NEXT ( -- addr or 0 )  MARKS @  DUP -1 =
3.          IF    DROP 0
4.          ELSE  1  ( mask )  16 0
5.                DO  2DUP AND  0=  ( unmarked bit )
6.                    IF  OR  MARKS !  I ADDRESS @  LEAVE  THEN
7.                    2*
8.                LOOP
9.          THEN  ;
10.
11. : TEMP ( -- addr  )  GET-NEXT  ?DUP 0=
12.          IF  COLLECT-GARBAGE  GET-NEXT ?DUP 0=
13.              ABORT" storage exhausted "  THEN  ;
14.
15.   -->


Blk # 9                         File: tempsto.blk
0.  \   Temporary Storage  hi level                    19Aug88jjw
1.
2.  : MARKED ( -- A1 .. Ak k ; marked addresses ) 0 MARKS @
3.        16 0  DO   0 2 UM/MOD SWAP
4.                   IF  I ADDRESS @  -ROT  SWAP 1+ SWAP  THEN
5.              LOOP   DROP  ;
6.
7.    : UNMARKED ( -- A1 .. Ak k ; unmarked addresses ) 0 MARKS @
8.        16 0  DO   0 2 UM/MOD SWAP  0=
9.    IF  I ADDRESS @  -ROT  SWAP 1+ SWAP  THEN
10.              LOOP   DROP  ;
11.
12.
```

```
13.   \ These are used if the addresses point to storage to be
14.   \ reclaimed. Marked addresses are still in use.
15.
```