A FORTH Implementation of the Heap Data Structure for Memory Management*

W. B. Dress

Instrumentation and Controls Division
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831

Abstract

The use of the heap for memory management provides the FORTH programmer with a versatile tool. Its use speeds program development at the conceptual level by allowing the program designer to consider dynamic arrays, garbage collection, and overlays; and at the implementation stage by providing a framework for easy manipulation of such data structures. An examination of the high-level code leads naturally to examples of these and other techniques of dynamic data management.

The implementation presented here has been used for some time in our FORTH-based version of the expert systems language, OPS5. This has resulted in faster execution times for OPS5 programs and has overcome problems in extending the language to the real-time domain.

Introduction

One of the major problems looming for a production version of our FORTH-based, real-time expert-systems language was that of garbage collection—a term applied to the process of reclaiming memory when the data occupying it are no longer needed. To understand the problem environment that led to exploring a heap data structure for garbage collection, a digression into the internals of the application are in order.

OPS5 (1) is a well-known and widely used expert-system language which operates by matching the patterns contained in production rules (2) with potential candidates in working memory. When all of the patterns in a rule are matched by specific instances, the rule "fires," making a few more working memory patterns for other rules to cogitate upon and perhaps deleting a few existing ones as well (3). As the system evolves in time, memory soon fills up unless the space occupied by the "removed" patterns or lists is reclaimed for subsequent use.

Since garbage collection is not essential for the short-term health of an OPS5 program, it was deferred until the FORTH implementation was completed. The flexibility of FORTH forgives such gross neglect of strict top-down coding practices. The word used to mark a pattern as removed, called **REMOVE**, could easily be rewritten to reclaim the space occupied by the unwanted list without affecting any other definitions. Indeed, when the heap was implemented, just such a process was employed to interface the existing software to the new capabilities.

*Research performed at Oak Ridge National Laboratory and sponsored by the U.S. Department of Energy under Contract No. DE-AC05-840R21400 with Martin Marietta Energy Systems, Inc.

A note on nomenclature is in order at this point. The word "heap" as used in this paper is not the classical heap data structure found in computer science texts, but rather is a convenient way of talking about a memory manager. The classical heap structure provides a means for managing priority queues and other records such that each key for a record is larger than the keys at two other specified locations [a complete binary tree with the root of each subtree being larger than its children would be an example (4)]. A classical heap may easily be embedded in the memory-management heap discussed in this paper. For example, an ingenious means of constructing trees in FORTH described by Forsley (5) could be adapted with the necessary indirection to be used inside another data structure, creating a classical heap within a memory heap.

Instrumentation

The driving force behind the effort to implement OPS5 in FORTH was the desire to have an expert-system tool for the practicing engineer interested in adding intelligence to process control systems and understanding multivariate sensor patterns. A Hewlett-Packard Series 200, Model 36 desktop computer (HP 9836) with an IEEE 488 instrument bus, fast serial (IEEE 422) and parallel ports, and a color graphics card was available for the project. Thus the hardware was oriented towards an industry-standard, user-configurable system allowing access to a variety of instruments communicating in a standard protocol.

The HP 9836 is a Motorola MC68000-based machine running at 8 MHz. The keyboard device contains a cursor pointer (a rotary knob) as well as maskable, real-time clocks allowing interrupting timers for process synchronization. A medium-resolution (512 by 390 pixels) bit-mapped graphics screen, with character generation built in, is the primary interactive I/O device. For program development, the two internal 5 1/2-inch floppy disks were supplemented by a 4-Mbyte hard disk and a printer. The hard disk also provided virtual memory space for symbolic strings, which were represented in main memory by 16-bit tokens. This allowed smaller compiled OPS5 programs and faster pattern matching than would obtain if the symbolic information were RAM-resident.

Software

Since Multi-FORTH (6), an industrial version of FORTH that includes multitasking and real-time capabilities, was available for the HP Series 200 machines, the problem of multitasking was, in essence, solved. Multi-FORTH also supports the IEEE 488 instrument bus and the bit-mapped graphics of the HP 9836.

The major departures from the 79-Standard (7) involve the use of 32-bit data and parameter stacks for addresses and data [the MC68000 has 32-bit data and address registers and 16-bit instructions (8)]. Thus the Multi-FORTH fetch and store referred to 32-bit entities; therefore 16-bit versions called `W@` and `W!` were added to the kernel by the vendor. Continuing the logic of the 32-bit data words, there is little need for the standard double-precision operators of FORTH-79 unless one needed more than 9 digits of precision. In the code presented below, the operators such as `2DUP` refer explicitly to two 32-bit stack items, unlike the 79-Standard in which `2DUP` conceptually refers to one double-precision stack item. Thus the stack picture for `2DUP`'s action is `( n1 n2 - n1 n2 n1 n2 )`, which is equivalent to `( d1 - d1 d1 )`, where `d1` is a double-precision notation for `n1 n2`. The difference is conceptual only, but it can be confusing. To round out the arithmetic, the vendor has added the words necessary for keeping 64-bit intermediate results (see ref. 6). `ON` and `OFF` are used to store logical "true" (-1) and "false" (0) at a desired address, as in `addr OFF`. The main differences between Multi-FORTH and the 79-Standard occur not in variations but in a wealth of extensions as the full user interface capabilities of the HP 9836 are supported, some in the kernel and others by supplied source code.

Implementation

Once the syntax and underlying algorithm (9) of OPS5 were incorporated into a FORTH environment, the language could be extended toward the goal of a multitasking, real-time expert system tool. Running a simple benchmark program such as the Towers of Hanoi with 12 disks or so became a memory-intensive process when reclamation was ignored. The first extension was to add garbage collection, which is normally performed automatically in the LISP environment.

Garbage Collection

A subject of numerous research papers (10), book chapters (11), and vendor hype, garbage collection is a means of marking list nodes (linked data elements) as unneeded and reclaiming memory space when it is almost exhausted. A feature of this process is that the typical application software ignores reclamation, leaving it to a garbage-collection cycle which can last a very long time indeed. Strictly speaking, garbage collection could be fatal in a real-time system. A new solution provided by some LISP-machine vendors uses a parallel garbage-collection processor. Thus a conflict between the needs of the software and the external events is less likely but is not removed since garbage collection remains asynchronous and can occur after long periods of garbage accumulation.

Enter the Heap

MacFORTH (12), a dialect of FORTH, has been adapted to the peculiarities of the Apple Macintosh personal computer. MacFORTH allows the user to make direct calls to the ROM-based heap manager (13) for reserving memory from and releasing memory to the memory pool, known as the heap. Observing the simplicity of manipulating dynamic data structures on the Macintosh with MacFORTH made it obvious that a similar implementation of a heap data structure would solve (or, more accurately stated, finesse) the garbage collection problem on the HP 9836 version of OPS5.

Implementing the Heap

Once the basics of memory allocation and deallocation were firmly in mind, it became a matter of a few hours to write code implementing a simple but adequate heap data structure in Multi-FORTH. The fundamental idea of the heap is to reserve a large pool of memory which is partitioned into two major portions, the handles and the pieces. The pieces are blocks of memory of arbitrary size which are reserved for use by a call to the heap manager. The pointers to these pieces are contained in "handles", which reside at the top of the heap. Figure 1 shows the basic internal structure as implemented.

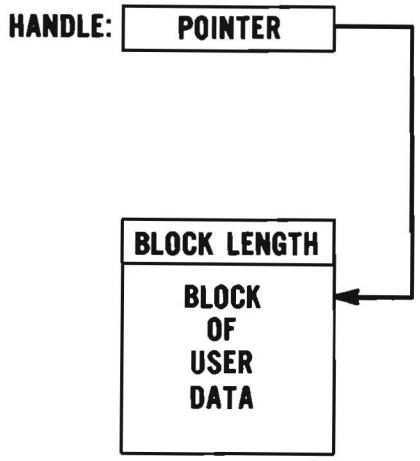

Figure 1. Pointer and Handle Relationships Within the Memory Heap.

When a block of memory is required, the address of the next available memory location is written to the next available handle cell, a pointer is advanced by the size of the piece requested, and the handle (address of the pointer to the memory actually allocated) is returned on the stack. The user application is responsible for keeping track of the handle until it is no longer needed, at which time a call should be made to the heap manager releasing the handle so that the referenced memory block can be deallocated. At this point, the collection of allocated blocks is compacted to recover the unused space. Such holes are moved to the top of the allocated portion for later use. Figure 2 illustrates this process.

Since the region of the heap containing the handles is never moved once the heap is created, the handles can be thought of as absolute memory locations containing pointers to blocks of memory which can be moved, expanded, or contracted as the application deems necessary. The virtue of such activity is its optimal use of memory even when the size and temporal characteristics of data blocks are *a priori* unknown.

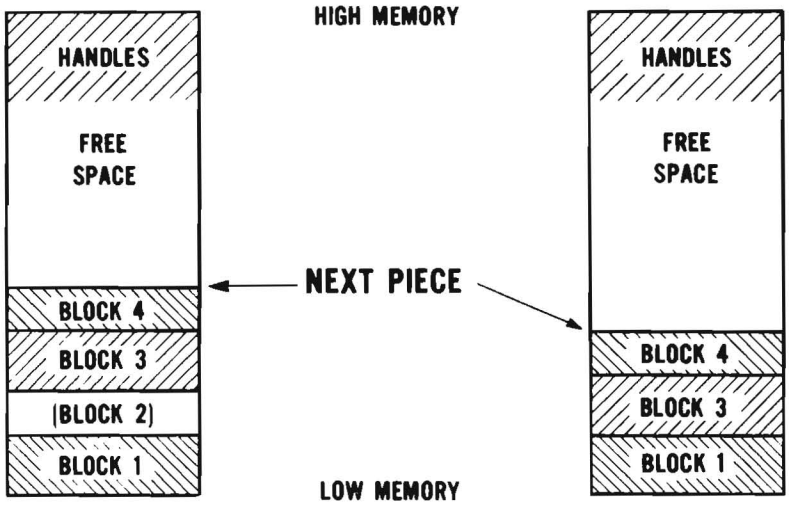

Figure 2. Removal of Block 2, Showing the Heap Before and After Compaction.

Discussion of the Source Code

The source code for the heap is presented in Figure 3. The first part creates the heap space and defines the constants used in heap management. The word `VALUE` is based on the `VALUE-TO-AT` construct discussed by McNeil (14) and originated by Bartholdi (15) (`AT` is McNeil's &). `NEXT.PIECE` contains the next pointer to be allocated, and `HEAP.TOP` is a value returning the address of the top of the heap. The values `LATEST.RELEASED` and `NEXT.HANDLE` allow a way to manage handle allocation that is more convenient than always choosing the next lower location for the new handle. Note that the handles are allocated from the top of the heap downward and the pieces or blocks are allocated from the bottom up, thus mirroring the stack/dictionary memory behavior of most FORTH systems.

Next come three low-level words used in heap management. `?COLLISION` returns "true" if the remaining free memory is less than the amount requested, `SCAN.HEAP` looks for an available handle in the space of active handles, and `BUMP.HANDLES` adjusts the pointers of each handle affected by a heap compaction. To illustrate, if a hole in the midst of a number of currently used blocks is reclaimed, all pointers greater than the address of the reclaimed block must be adjusted by the length of the reclaimed block to remain valid.

The heap lexicon (user interface to the heap) consists of the words HANDLE.SIZE, FROM.HEAP, TO.HEAP, and RESIZE.HANDLE. HANDLE.SIZE returns the size of the space allocated for a block given its handle. Note that this code uses a 2-byte field for the block size, thus restricting the maximum allocatable heap piece to 64 kbytes. This was found to be more than adequate for the expert-systems application as most of the lists and working memory elements were under 100 bytes in length. Overlay applications may well require larger blocks of dynamic memory, and it is a simple matter to alter the code to allow a full 32-bit cell for the block size.

FROM.HEAP allocates the requested block space and returns a handle containing the relevant pointer. This is accomplished by checking the remaining heap space to see if a handle has been released recently. If so, it makes use of the recent handle rather than allocating a new one. The extra bookkeeping involving the constant LATEST.RELEASED speeds up allocation when a lot of heap activity is going on. If LATEST.RELEASED is zero, the heap is scanned by SCAN.HEAP to obtain the next available handle in the pool of previously allocated handles. If none is found, a new handle is allocated at the bottom of the pool. This feature maintains the size of the handle space at a minimum at all times. RECOVER.HANDLE allows the user to recover the handle for a given pointer. If the pointer is no longer valid, a value of zero is returned.

TO.HEAP returns the handle to the pool of available handles and compacts the heap if warranted. The verb RESIZE.HANDLE is the key to allowing dynamic data structures in the heap. A request to change the size of an allocated block must be checked for heap collision first. Upon passing this check, a test to determine the location of the block is made. If it is on the top of the allocated blocks, a mere change in the NEXT.PIECE pointer suffices for the resizing. If not, the existing block is moved to the top of the allocated heap and resized.

Additional words are supplied to reset the heap, to verify whether an integer is an active handle, and to let the user obtain the amount of heap space remaining.

Uses of the Heap Data Structure

The heap was used to solve (more properly, avoid) the garbage collection problem by running the system in a synchronous mode. Thus each time a block of memory was needed, it was reserved by a simple call to the heap manager – nnn FROM.HEAP, which returned the handle for subsequent use. Alternately, when the space was no longer needed, it was returned to the heap via the TO.HEAP command. At this point the memory was reclaimed. Since the more recent working memory elements in OPS5 had a higher probability of being removed than the earlier ones, a heap compaction usually involved moving only a few tens of bytes. Note that removal of the top heap block involves only a pointer update and a handle reset.

Overlays

A particularly convenient use of the heap is for code and data overlays. The use for data overlays is straightforward unless the data contain references to nonresident information, which must be located and brought into the heap.

For executable code, assume that the compilation occurred earlier within the normal FORTH dictionary structure, and the compiled code has been copied to mass storage and then forgotten. To execute the stored code, one allocates heap space, copies the stored code into the heap, and executes the sequence @ PFA CFA EXECUTE after the handle to the code has been placed on the stack. (PFA converts the dictionary address, the link field in Multi-FORTH, into the parameter field address). Note that this direct execution of the code assumes that all PFAs in the heap refer to code or data currently resident in main memory, at the same locations as at compilation time.

The general problem of allowing heap references to nonresident structures is much more complex than the example presented above. For a hint as how to proceed, note that cells referring to nonresident memory must be so marked when stored and must be essentially recompiled into the heap when retrieved from mass storage, a method which would entail a special LOADER for

nonresident code. Another possibility is to create an additional **NEXT** to resolve nonresident references into existing references within the heap and to require loading of additional code should the required piece not yet be heap-resident. This type of an asynchronous, fragmented overlay capability has not been required by our current applications, so we have not addressed the problem in sufficient detail to be able to present a working example. A possible solution to this problem is contained in the work of Kaplan (16) who discusses linking nonresident modules.

Dynamic Arrays

The problem (insurmountable in some languages) of arrays whose size is unknown until run time is one that most every FORTH programmer meets sooner or later. The usual solution is to allot enough memory to cover every conceivable case, which is wishful thinking for an application to be turned over to an end-user. The use of the heap is quite simple in this case: the size, determined at run time, is used to request heap space, which is then used for the array. If the allotted size becomes insufficient as determined by the usual bounds tests, the piece can easily be resized. When the array is no longer needed, the space is returned to the heap. An example using a list structures array is shown in Figure 4.

Linked Lists and Sorting

Most any type of data can be represented in the form of a linked list (see Knuth, ref. 11, for examples of the versatility of such lists). The convenience of allowing such lists to be made and destroyed at run time provides a convenient solution to the problems of sorting and database searches when a rapid sort of a section of the database is followed by another such demand. Since efficiency requires that as many of the keys as possible be present in memory, and since the database is almost certainly larger than the fast memory available, a rapid means of memory allocation and deallocation becomes necessary. This is particularly true when lists of varying sizes must be arranged according to some plan without the benefit of suitable keys.

Extensions and Complications

The simple implementation presented here is useful for only single-user systems (who releases a structure in a multiuser system?) and where there are no absolute memory references to data brought into the heap. Thus, a natural extension of such a memory heap is to add features allowing absolute references to in-heap data and resolving conflicts between different users. When used as an address, the 32-bit word of the MC68000 has the high-order 8 bits free since the machine decodes only the lower 24 bits to obtain an address. This available byte can be thought of as a tag field for containing the marking bits of standard garbage collection (see ref. 11), for indicating the presence of any absolute data contained in the block, and for serving as a semaphore indicating multiple users. For hardware without this feature, a special tag field would be added to each handle.

A few verbs could be written to make appropriate use of the tag field according to the needs of the application. For example, the Macintosh heap manager (refs. 12 and 13) includes calls to lock a block in memory (assuring validity of absolute memory references) and a means to mark a block as purgeable (can be removed on the next heap compaction) and unpurgeable. A means of reallocating a block via an additional level of indirection is also provided.

Summary

The uses of the heap data structure, even in its simplest form, are varied and versatile. Ranging in complexity from the simple dynamic array to the FORTH program overlay in multiuser systems, the heap memory manager has a place in the tool box of every FORTH programmer. Once familiar with the heap concept, the programmer can defer thorny issues from the design stage of a problem, where details do not belong, to the implementation stage, where they are necessary. Creating one dynamic array on top of the current dictionary space may solve a particular problem, but without the heap it would take a great amount of gymnastics to use another dynamic structure at the same time.

The ease of bookkeeping necessary for many volatile list structures makes the heap a natural manager for LISP-like extensions to an application. The idea of keeping a list of handles in the heap (a handle to a block of handles) provides a way to manage second-order data structures in the same convenient manner as primary ones.

References

[1] Charles L. Forgy, *OPS5 User's Manual*, Department of Computer Science, Carnegie-Mellon University, 1981.

[2] Patrick Henry Winston, *Artificial Intelligence*, 2nd ed., Addison-Wesley, Reading, Mass., pp. 172-208, 1984.

[3] Charles Forgy, Anoop Gupta, Allen Newell, and Robert Wedig, "Initial Assessment of Architectures for Production Systems," *Proceedings of the National Conference on Artificial Intelligence*, pp. 116-120, 1984.

[4] Robert Sedgewick, *Algorithms*, Addison-Wesley, Reading, Mass., chap. 11, 1983.

[5] Lawrence Forsley, "Recursive Data Structures," *1982 FORML Conference Proceedings*, Forth Interest Group, San Carlos, Calif., 1982.

[6] Creative Solutions, Inc., *Multi-FORTH Version 2.00 User's Manual*, Rockville, Maryland, 1984.

[7] FORTH Standards Team, *FORTH-79*, Forth Interest Group, San Carlos, Calif., October 1980.

[8] Motorola Inc., *MC68000 16-Bit Microprocessor User's Manual*, 3rd ed., Prentice-Hall, Englewood Cliffs, N.J., 1982.

[9] Charles L. Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence*, Vol. 19, No. 1, 1982.

[10] Tim Hickey and Jacques Cohen, "Performance Analysis of On-the-Fly Garbage Collection," *Communications of the ACM*, Vol. 27, No. 11, pp. 1143-1154, November 1984.

[11] Donald E. Knuth, *The Art of Computer Programming*, 2nd ed., Vol. 1, pp. 406-420, Addison-Wesley, Reading, Mass., 1973.

[12] Creative Solutions, Inc., *MacFORTH Level Two User's Manual*, Rockville, Maryland, 1985.

[13] Apple Computer, Inc., *Inside Macintosh*, Cupertino, California, 1984.

[14] Michael McNeil, "The 'TO' Variable" in *1980 FORML Conference Proceedings*, Forth Interest Group, San Carlos, CA, pp. 75-77, 1980.

[15] Paul Bartholdi, "The 'TO' Solution," *Forth Dimensions*, Vol. 1, No. 4, June/July 1978.

[16] George Kaplan, "Modular Forth," *1984 FORML Conference Proceedings*, Forth Interest Group, San Carlos, CA, 1985.

Manuscript received June 1985.

Author graduated from Harvard with a PhD in Physics in 1968. Worked in the Physics Division at Oak Ridge National Laboratory in neutron magnetic resonance, symmetry principals, electron spectroscopy, and dielectronic recombination. Joined the Instrumentation and Controls Division at the laboratory in 1981 to work on sensor development. Subsequently started a project to bring expert-systems technology to the practicing engineer by rewriting OPS5 in a multitasking version of FORTH for process instrumentation.

Figure 3. Source code for the heap memory manager.

```
SCR # 55
  0 ( Set up Heap and Heap Pointers)                    ( 120385 WBD)
  1 ( Following for readability only )
  2 : VALUE CONSTANT ;       ( Values are constants with following: )
  3 : TO [COMPILE] ' STATE @ IF COMPILE ! ELSE ! THEN ; IMMEDIATE
  4 : AT [COMPILE] ' ; IMMEDIATE
  5 ( Now the Heap )
  6 HEX
  7 CREATE HEAP 200 400 * ALLOT              ( 512 Kbyte heap       )
  8 HERE
  9 HEAP     VALUE  NEXT.PIECE               ( Heap Pointer         )
 10 DUP      VALUE  HEAP.TOP
 11 0        VALUE  LATEST.RELEASED          ( Increased efficiency )
 12 4-       VALUE  NEXT.HANDLE              ( Increased efficiency )
 13 HEAP 200 400 * 0 FILL
 14 DECIMAL
 15

SCR # 56
  0 ( Low-level manager words )                          (120385 WBD)
  1
  2 : ?COLLISION  ( proposed size -- flag )
  3    NEXT.HANDLE NEXT.PIECE ROT + 2+ < ;
  4
  5 : SCAN.HEAP  ( -- handle )
  6    HEAP.TOP 4-              ( Start looking for an empty handle )
  7    BEGIN DUP @                              ( Is handle in use? )
  8      IF 4- ELSE DUP NEXT.HANDLE =     ( Are we finsihed looking?)
  9         IF -4 AT NEXT.HANDLE +! THEN EXIT     ( with new handle )
 10         THEN DUP NEXT.HANDLE =                  ( Keep looking )
 11    UNTIL DUP OFF                     ( None found, so create one )
 12    DUP TO LATEST.RELEASED          ( at NEXT.HANDLE while giving )
 13    4- DUP 4- TO NEXT.HANDLE ;      ( a LATEST.RELEASED and a new )
 14                                    ( NEXT.HANDLE for efficiency! )
 15

SCR #57
  0 ( Adjust handle contents after heap compaction)     ( 120385 WBD)
  1
  2 : BUMP.HANDLES  ( n\addr -- |     Look for pointers above point)
  3    SWAP NEGATE >R   HEAP.TOP 4-                  ( of compaction)
  4    BEGIN 2DUP @ < IF R@ OVER +! Then ( When found,re-point them)
  5      4- DUP NEXT.HANDLE =        ( No more handles to examine? )
  6    UNTIL   2DROP R>DROP ;
  7
  8 ( Using 16-bit sizes -- 64K max pieces )
  9 : HANDLE.SIZE  ( handle -- size )  @ 2- W@ ;
 10
 11
 12
 13
 14
 15
```

```
SCR #58
 0 ( Allocate a section of memory )                    ( 120385 WBD)
 1
 2 : FROM.HEAP  ( size -- handle )
 3    =CELLS   DUP ?COLLISION
 4    IF DROP FALSE EXIT THEN    ( Returns 0 if no room in heap)
 5    LATEST.RELEASED ?DUP       ( Look here first -- may save time)
 6    IF AT LATEST.RELEASED OFF  ( Use LATEST.RELEASED )
 7    ELSE SCAN.HEAP THEN        ( Oops! Need to look unused handle)
 8    NEXT.PIECE 2+ OVER !       ( Put pointer into handle )
 9    OVER NEXT.PIECE W!         ( Put size at pointer -2)
10    SWAP 2+ AT NEXT.PIECE +!   ( Adjust pointer to NEXT.PIECE )
11    ;
12
13
14
15

SCR #59
 0 ( Return a handle and compact heap )                ( 120385 WBD)
 1 : TO.HEAP  ( handle -- | compacts heap)
 2    DUP HANDLE.SIZE 2+  OVER @ 2-  2DUP + DUP>R
 3    SWAP NEXT.PIECE R@ - 4/ 1+ MOVE     ( Compact Heap )
 4    DUP NEGATE AT NEXT.PIECE +!         ( Update pointer )
 5    NEXT.HANDLE    (Start search for returned handles )
 6    BEGIN DUP @ NOT WHILE 4+ REPEAT     ( Loops usually once only)
 7    4- TO NEXT.HANDLE
 8    SWAP DUP OFF TO LATEST.RELEASED
 9    R> BUMP.HANDLES                     ( Adjust pointers )
10    LATEST.RELEASED NEXT.HANDLE > NOT   ( Advance NEXT.HANDLE? )
11    If NEXT.HANDLE DUP TO LATEST.RELEASED
12       4- TO NEXT.HANDLE ( Needed if LATEST.RELEASED gets behind)
13    THEN ;                ( NEXT.HANDLE which could happen if an
14                  old handle in the body of handles gets returned
15                  before the latest one allocated is returned. )

SCR # 60
 0 ( Change size of an allocated region )              ( 120385 WBD)
 1
 2 : RESIZE.HANDLE   ( handle\size -- flag: 0 = ok)
 3    OVER @ 2- 2DUP W@ - 2- ?COLLISION
 4    IF 2DROP EXIT THEN
 5    DUP DUP W@ + 2+ NEXT.PIECE =    \ Avoid moving heap?
 6    IF 2DUP W@ - AT NEXT.PIECE +! W! NOT EXIT THEN
 7    NEXT.PIECE 3 PICK 2+ 4/ 1+ MOVE DUP>R
 8    FROM.HEAP OVER TO.HEAP DUP @ SWAP DUP OFF
 9    TO LATEST.RELEASED
10    R> OVER 2- W! SWAP    ! 0 ;
11
12
13
14
15
```

```
SCR # 61
  0 ( Reset the Heap -- use with care!!)                    ( 120385
  1 : RELEASE.HEAP
  2      HEAP.TOP NEXT.HANDLE DO I OFF 4 +LOOP
  3      HEAP TO NEXT.PIECE
  4      0 TO LATEST.RELEASED
  5      HEAP.TOP 4- TO NEXT.HANDLE  ;
  6
  7 ( Search thru the handles for the pointer)
  8 : RECOVER.HANDLE  ( pointer -- handle )
  9     NEXT.HANDLE 4+
 10     BEGIN 2DUP @ = NOT WHILE 4+ DUP HEAP.TOP < NOT
 11       IF DROP NOT EXIT THEN
 12     REPEAT SWAP DROP  ;
 13
 14
 15

SCR # 62
  0 ( Misc stuff useful for debugging )                    ( 120385 WBD)
  1 : ?HANDLE   ( n -- flag| flag = -1 if n is a current handle )
  2     NEXT.HANDLE 4+  HEAP.TOP RANGE IF @ ELSE DROP FALSE THEN ;
  3
  4 : HEAP.SIZE  ( -- size )  NEXT.HANDLE NEXT.PIECE -
  5   0 NEXT.HANDLE BEGIN DUP 4+ @ NOT WHILE
  6       SWAP 4+ SWAP 4+ REPEAT HEAP.TOP 4- MIN TO NEXT.HANDLE + ;
  7
  8 : HEAP.ERROR  ( flag -- )  ERROR' Out of Heap Space' ;
  9
 10 : HANDLES?  ( Counts # of handles in use )
 11   0 HEAP.TOP NEXT.HANDLE DO I @ IF 1+ THEN 4 +LOOP CR . ;
 12
 13 : .HANDLES  ( Prints all handles & pointers)
 14    BASE @ HEAP.TOP NEXT.HANDLE HEX
 15    DO I @ IF CR I DUP 10 .R @ 10 .R THEN 4 +LOOP BASE ! ;
```

Figure 4. Some examples in list building and manipulation of the heap words.

```
SCR # 66
 0 ( Build a new list or add to a list)                ( 120385 WBD)
 1
 2 : >LIST  ( new entry\'handle to list -- )
 3     DUP @ ?DUP
 4     If  DUP HANDLE.SIZE 4+ RESIZE.HANDLE HEAP.ERROR
 5     ELSE 8 FROM.HEAP OVER ! THEN
 6     @ DUP HANDLE.SIZE SWAP @ + 4- 0 OVER ! 4- ! ;
 7 ( Stores a 4-byte value at the end of the list. If this is the
 8   first list item, contents of 'handle (handle address) should
 9   be zero -- the result is a new list with the new item in first
10   place, and the list is terminated by a 32-bit 0. )
11
12 : @LIST  ( position\'handle -- 32-bit item)
13    @ @ SWAP 4* + @ ;
14 ( Get item from a list -- specifying the list position
15   { 0, 1, etc } and the address of the stored handle.)

SCR # 67
 0 ( Treat list as a fifo stack )                      ( 120385 WBD)
 1
 2 : POP.LIST  ( 'handle--top item)   DUP @ DUP NOT
 3    IF SWAP DROP  EXIT THEN
 4    DUP HANDLE.SIZE DUP 3 PICK @ + 8- @
 5    ROT ROT 4- DUP 4 =
 6    IF DROP TO.HEAP SWAP OFF
 7    ELSE RESIZE.HANDLE DROP SWAP DROP THEN ;
 8 ( Pops top item off the list whose handle address is given,
 9   then compacts the list, returning the handle to the heap
10   and storing zero at the handle address should the list
11   become empty.)
12
13
14
15

SCR # 68
 0 ( List Removal & Compaction )                       ( 120385 WBD)
 1
 2 : -LIST  ( 32-bit item\'handle -- )
 3    DUP>R @ ?DUP NOT
 4    IF R>DROP DROP EXIT THEN
 5    SWAP R@ ?LIST 1+ ?DUP
 6    IF 1- 4* OVER HANDLE.SIZE DUP 8 =
 7       IF 2DROP TO.HEAP R> OFF EXIT THEN
 8       +DUP>R OVER - 4- >R     ( Handle size & Length of move >R)
 9       SWAP @ + DUP 4+ SWAP R> 4/ MOVE
10       R> 4- R> @ SWAP RESIZE.HANDLE  DROP
11    ELSE DROP R>DROP THEN ;     ( Zero-Terminated lists only!!!!)
12 ( Removes specified item from the list whose handle address
13   is given.)
14
15
```