22 Memory (memory.hhf)

The memory unit (header file is memory.hhf) contains the routines used to allocate and deallocate dynamic storage on the heap. There are a set of routines that allocate storage for general objects and a set of routines used to specifically allocate storage for strings.

As of HLA v1.69, the "allocation granularity" is eight bytes (that is, these routines always allocate data in multiples of eight byte chunks) and there is a 24-byte metadata overhead associated with each allocation. Therefore, you should avoid doing a large number of small allocations if you want to use memory efficiently. Note that these values are subject to change in future versions of the library.

These memory allocation routines associate a *reference counter* with each block. Whenever you first allocate a block on the heap, the reference counter is initialized with one. A "*mem.newref*" call instructs the heap management routines to increment this reference counter. The reference counter tracks how many different pointers in an application are referring to a single block of memory in the heap. When you call the mem.free routine to return storage to the heap, the heap management code will decrement the reference counter and only free up the storage when the reference counter decrements to zero. This can help avoid dangling pointers if you use the mem.newref routine in an appropriate fashion.

A Note About Thread Safety: The memory management routines maintain a couple of static global variables that track free and in-use blocks of memory. Currently, these values apply to all threads in a process. As such, the current implementation is not thread-safe. When the process module is added to the standard library, the memory management system will be modified to be thread safe. Until then, you should explicitly synchronize access to the HLA memory manager if you are writing multi-threaded applications.

22.1 Memory Module

To call functions in the Memory module, you must include one of the following statements in your HLA application:

```
#include( "memory.hhf" )
or
#include( "stdlib.hhf" )
```

Note: be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) before reading this chapter.

22.2 Deprecated Names

The HLA Standard Library has inherited some older, deprecated, names from the HLA stdlib v1.x. If you look at the include files for the Standard Library, those names might still be present. This document, however, will not describe those deprecated names from the v1.x library.

22.3 Generic Memory Allocation

The following functions allocate, deallocate, and operate on blocks of memory that may contain arbitrary data.

mem.alloc overloads mem.alloc1 and mem.alloc2

If you invoke mem.alloc with one parameter, it calls mem.alloc1; if you call mem.alloc with two parameters, it calls mem.alloc2.

procedure mem.alloc1(size:dword); @returns("eax");

The *mem.alloc1* routine allocates the requested number of bytes. If successful, this routine returns a pointer to the allocated storage in the EAX register. This routine raises an *ex.MemoryAllocationFailure* exception or an *ex.MemoryAllocationCorruption* exception if it fails. Note that this function does not initialize the block of memory to any particular value when it allocates it. In particular, do not count on this function setting the block of memory to zeros.

```
HLA high-level calling sequence example:
    mem.alloc1( 1024 );
    mov( eax, memBlkPtr );


    HLA low-level calling sequence example:

      pushd( 1024 );
      call mem.alloc1;
      mov( eax, memBlkPtr );
```

procedure mem.alloc2(size:dword; callback:thunk);
@returns("eax");

This function is very similar to *mem.alloc1* with one major difference: after allocating the block, it will call the *callback* thunk. This allows the caller to track memory usage, initialize the memory block, or perform any other activity before returning from mem.alloc2. On entry into the thunk, ECX will contain the block size and EAX will point at the memory block. The direction flag will be clear . Anything you do in the thunk is entirely up to you, but you will want to return a pointer to an appropriately sized memory block in the EAX register. You can use the other registers (ebx, ecx, edx, esi, and edi) as you see fit.

```
  HLA high-level calling sequence example:

  mem.alloc2( 2048, thunk #{ call savePtrInEAX; }# );
  mov( eax, memBlkPtr );

  HLA low-level calling sequence example:

      pushd( 2048 );
      push( ebp );// Thunk pointer
      pushd( &thunkCode );
      jmp callrealloc2;
      thunkCode:
         call savePtrInEAX;
         ret();

      callrealloc2:
      call mem.alloc2;
      mov( eax, memBlkPtr );
```

procedure mem.zalloc(size:dword); @returns("eax");

The *mem.zalloc* routine allocates the requested number of bytes and zeros out the data storage allocated. If successful, this routine returns a pointer to the allocated storage in the EAX register. This routine raises an *ex.MemoryAllocationFailure* exception or an *ex.MemoryAllocationCorruption* exception if it fails.

```
    HLA high-level calling sequence example:

    mem.zalloc( 1024 );
    mov( eax, memBlkPtr );


    HLA low-level calling sequence example:

      pushd( 1024 );
      call mem.zalloc;
      mov( eax, memBlkPtr );
```

procedure mem.free(memptr:dword);

This function frees up storage previously allocated by the *mem.alloc* routine. A pointer returned from *mem.alloc* must be passed as the parameter to this function. This routine actually decrements a *reference counter* and only frees the storage when the reference counter becomes zero. See the discussion of *mem.newref* for more details.

```
HLA high-level calling sequence example:

mem.free( memBlkPtr );


HLA low-level calling sequence example:

  push( memBlkPtr );
  call mem.free;
```

mem.realloc overloads mem.realloc1 and mem.realloc2

If you invoke mem.realloc with two parameters, it calls mem.realloc1; if you call mem.realloc with three parameters, it calls mem.realloc2.

procedure mem.realloc1(memptr:dword; newsize:dword); @returns("eax");

The *mem.realloc1* routine resizes a previous allocated block of memory. The first parameter is the pointer to the original block, the second parameter is the new size. If the new block is smaller, this routine truncates the data beyond the new size. If the new block is larger, this routine will copy the data if it cannot expand the block in-place.

If the address of the block does not change, then the block created by *mem.realloc1* inherits the reference counter value from the original block. However, if the *mem.realloc1* function must create a new block and copy the data to that new block, then the reference counter of the new block is set to one. If the reference counter of the original block was not one prior to the realloc operation, then the system simply decrements the original reference counter and does not deallocate the original storage. It is important to realize that the *mem.realloc1* operation may leave two allocated blocks and any previous pointers (noted by *mem.newref* calls) are still valid and still point at the original data. The pointer returned by *mem.realloc1* points at the new block.

```
HLA high-level calling sequence example:

mem.realloc1( memBlkPtr, 2048 );
mov( eax, memBlkPtr );


HLA low-level calling sequence example:

  push( memBlkPtr );
  pushd( 2048 );
  call mem.realloc1;
  mov( eax, memBlkPtr );
```

procedure mem.realloc2(memptr:dword; newsize:dword; copycallback:thunk); @returns("eax");

This function is very similar to *mem.realloc1* with one major difference: if, during the reallocation operation, *mem.realloc2* needs to copy a block of data because it cannot expand the existing block in-place, it will call the *copycallback* thunk to handle the copy operation. This allows the caller to readjust application-dependent pointers and do other activities if the block has to be moved during a realloc operation. On entry into the thunk, ECX will contain the block size, ESI will point at the source block, and EDI will point at the destination block. The direction flag will be clear and you can assume that the blocks do not overlap. You should, at the very least, execute a "rep.movsb;" instruction to copy the source block to the destination block.

Anything else you do in the thunk is entirely up to you, but typically, you will want to adjust any pointers in your application that point at the source block so that they point at the destination block.

```
HLA high-level calling sequence example:

mem.realloc2( memBlkPtr, 2048, thunk #{ rep.movsb }# );
mov( eax, memBlkPtr );


HLA low-level calling sequence example:

  push( memBlkPtr );
  pushd( 2048 );
  push( ebp );// Thunk pointer
  pushd( &thunkCode );
  jmp callrealloc2;
  thunkCode:
      rep.movsb();
      ret();

  callrealloc2:
  call mem.realloc2;
  mov( eax, memBlkPtr );
```

#macro mem.talloc(size);   (returns "eax" as macro result)

This is a macro that "temporarily" allocates the specified storage. This macro allocates the specified storage on the stack and returns the address of the storage (i.e., the ESP value) in the EAX register. The address is always dword aligned; *mem.talloc* will allocate up to three additional bytes to ensure dword alignment.

You may use the *mem.talloc* call anywhere a single instruction is legal (including using *mem.talloc* as an operand to another instruction).

There is no corresponding "tfree" routine since leaving the current procedure automatically deallocates the storage. That is, when a standard procedure exits, it resets the stack pointer, automatically removing the *mem.talloc'd* data. If you would like to explicitly free the data, then you should save the value of ESP prior to calling *mem.talloc* and this restore ESP from this saved value when you want to "free" the storage.

Warning: in order for a function to properly free the storage allocated by *mem.talloc*, the function must have a standard activation record or must otherwise restore ESP to the value it held prior to the invocation of *mem.talloc*. HLA procedures that generate a standard activation record (e.g., those that don't have the @noframe option) do this automatically. But if you write a procedure that has the @noframe option, you must take responsibility for restoring ESP's value to deallocate the storage set aside by *mem.talloc*.

Obviously, you cannot continue referencing the data allocated by *mem.talloc* once the enclosing procedure returns.

```
HLA high-level calling sequence example:

mem.talloc( 128 );
mov( eax, memBlkPtr );

Note: Because this is a macro, there is no low-level calling sequence.
```

procedure mem.isInHeap(memptr:dword);

This function returns false (NULL) in EAX if the *memptr* parameter does not point at a valid (allocated) object on the heap. It returns a pointer to the start of the data block on the heap if *memptr* does point within the data area of a valid block. You can use this function to determine whether an object was previously allocated via a call to *mem.alloc* (and should be free'd via a call to *mem.free*). Note that this function only returns non-NULL if the block is currently allocated. If you've free'd all instances of the block, this function will return NULL. In older versions of this routine, the function simply returned true or false. Assuming older code treated false as zero and true as anything else, that code will continue to function with this new version of the routine.

```
HLA high-level calling sequence example:

mem.isInHeap( memBlkPtr );
if( eax <> NULL ) then

  mem.free( eax );

endif;


HLA low-level calling sequence example:

  push( memBlkPtr );
  call mem.isInHeap;
  test( eax, eax );
  jz noFree;

      push( eax );
      call mem.free;

  noFree:
```

procedure mem.size(memptr:dword);

This function returns the amount of storage allocated in the block pointed at by *memptr*. The value of *memptr* must be a value returned by *mem.alloc*, *mem.realloc*, or *mem.realloc2*. Note that the value that *mem.size* returns might be slightly larger than the original request. This function returns the actual size of the allocated block, including any padding bytes added to the end of the block for alignment purposes.

```
HLA high-level calling sequence example:

mem.size( memBlkPtr );
mov( eax, blockSize );


HLA low-level calling sequence example:

  push( memBlkPtr );
  call mem.size;
mov( eax, blockSize );
```

procedure mem.stat;

This funtion returns statistics concerning the heap space in use by the memory allocation routines. This function returns the following values:

EAX - Total amount of space currently in use by the heap (this may not be contiguous!).

EBX - Total amount of free space in the heap.

ECX - Largest block of contiguous free space in the heap.

EDX - Number of blocks on the heap (free and in use).

EDI - Number of free blocks on the heap.

Note that the value in EBX, the total amount of free space in the heap, does not indicate the maximum amount of space that you can allocate. This simply indicates the amount of space that was previously allocated and has been freed. Generally, it is quite possible to allocate more storage than is available in the heap at any one time. Indeed, prior to the first *mem.alloc* operation, the *mem.stat* function will return zero in all these registers.

```
HLA high-level calling sequence example:

mem.stat();
mov( eax, spaceInUse );
mov( ebx, freeSpace );
mov( ecx, largestBlock );
mov( edx, numBlocks );
mov( edi, numFreeBlocks );


HLA low-level calling sequence example:

call mem.stat;
mov( eax, spaceInUse );
mov( ebx, freeSpace );
mov( ecx, largestBlock );
mov( edx, numBlocks );
mov( edi, numFreeBlocks );
```

mem.newref(memblk:dword);

This funtion increments a *reference counter* for the memory block whose address you pass as the parameter (this must be a block allocated by *mem.alloc*). The heap routines will not deallocate storage for a block of memory until you've called *mem.free* the number of times specified by the reference counter. The *mem.alloc* call initializes the reference counter to one, calls to *mem.newref* increment this value by one, calls to *mem.free* decrement this value by one (and frees the storage once the reference counter hits zero).

```
HLA high-level calling sequence example:

mem.newref( memPtr );


HLA low-level calling sequence example:

push( memPtr );
call mem.newref;
```

mem.getref(memblk:dword);

This funtion returns the *reference counter* value for the specified memory block. This function raises an *ex.PointerNotInHeap* exception if *memblk* does not point within a valid memory block. Note that if the block has been deallocated, this function returns zero, it does not raise an exception.

```
HLA high-level calling sequence example:

mem.getref( memPtr );
mov( eax, refCnt );

HLA low-level calling sequence example:

push( memPtr );
call mem.getref;
mov( eax, refCnt );
```

iterator mem.blockInHeap;

This is an iterator (used in a foreach loop) that returns the following information for each block (free and in-use) in the heap, one block per iteration:

EAX - Size of block

EBX - Address of data block

ECX - Reference count for block

This function is mainly intended for debugging purposes.

```
HLA high-level calling sequence example:

foreach mem.blockInHeap() do

  stdout.put
  (
     "size=$", eax,
     " adrs=$", ebx,
     " refcnt=$", ecx,
     nl
     );

  endfor;

HLA low-level calling sequence example:

  pushd( &endLoopBody );
  call mem.blockInHeap;
  stdout.put
  (
     "size=$", eax,
     " adrs=$", ebx,
     " refcnt=$", ecx,
     nl
     );
     ret();

  endLoopBody:
```

iterator mem.allocBlockInHeap;

This iterator is similar to mem.blockInHeap except it only iterates over the allocated blocks in the heap.

This function is mainly intended for debugging purposes.

```
HLA high-level calling sequence example:

foreach mem.allocBlockInHeap() do

  stdout.put
  (
     "size=$", eax,
     " adrs=$", ebx,
     " refcnt=$", ecx,
     nl
     );

  endfor;
```

```
HLA low-level calling sequence example:

  pushd( &endLoopBody );
  call mem.allocBlockInHeap;
  stdout.put
  (
     "size=$", eax,
     " adrs=$", ebx,
     " refcnt=$", ecx,
     nl
     );
     ret();

  endLoopBody:
```

iterator mem.freeBlockInHeap;

This iterator is similar to mem.blockInHeap except it only iterates over the free blocks in the heap. This function is mainly intended for debugging purposes.

```
HLA high-level calling sequence example:

foreach mem.freeBlockInHeap() do

  stdout.put
  (
     "size=$", eax,
     " adrs=$", ebx,
     " refcnt=$", ecx,
     nl
     );

  endfor;

HLA low-level calling sequence example:

  pushd( &endLoopBody );
  call mem.freeBlockInHeap;
  stdout.put
  (
     "size=$", eax,
     " adrs=$", ebx,
     " refcnt=$", ecx,
     nl
     );
     ret();

  endLoopBody:
```

22.4 String Memory Allocation

The memory-related functions in this category are used to allocate, deallocate, and manipulate dynamic string data. The main difference between these functions and the "standard" memory allocation functions is the pointer values these function manipulate. Because HLA string pointers must contain an address that is eight bytes into the string data structure (unlike standard memory allocation functions that work with pointers that point at the beginning of the memory block), these string functions automatically add or subtract that offset.

Because the calls to these functions are identical to the standard memory functions boasting the same names, please see the calling sequence examples given earlier.

```
procedure str.alloc( size:dword ); @returns( "eax" );
procedure str.realloc( strPtr:dword; size:dword ); @returns( "eax" );
procedure str.free( strPtr:dword );
procedure str.isInHeap( strPtr:dword ); @returns( "eax" );
```

The string allocation routines are used just like the general memory allocation routines except they allocate storage for a string variable and initialize the string object's *maxlength* and *length* fields. They return a pointer to the first character position of the string's data (that is, the address of the byte just beyond the *maxlength* and *length* fields). Note that the *str.isInHeap* function returns a pointer to the start of the string's data (the first character in the string) if it determines that the string has been allocated on the heap. See the discussion of *mem.realloc* to understand how *str.realloc* affects the reference counter for a string on the heap.

```
#macro str.talloc( size );     (returns pointer to new string in EAX ).
```

This is a macro that initializes storage on the stack for a string capable of holding *size* characters. This routine has the same benefits and drawbacks as the *mem.talloc* routine.

Note that the *size* parameter is the actual number of characters needed. the *str.talloc* routine automatically bumps this value up by nine to make room for the *length*, *maxLength*, and zero terminator fields of the string object. This macro also ensures that the stack (and, therefore, the string) is dword aligned in memory (it does this by adding up to three additional bytes to the string).

```
procedure str.newref( strPtr:dword );
```

This funtion increments a *reference counter* for the memory block whose address you pass as the parameter (this must be a block allocated by *str.alloc*). The heap routines will not deallocate storage for a block of memory until you've called *str.free* the number of times specified by the reference counter. The *str.alloc* call initializes the reference counter to one, calls to *str.newref* increment this value by one, calls to *str.free* decrement this value by one (and frees the storage once the reference counter hits zero).

```
str.getref( strPtr:dword );
```

This funtion returns the *reference counter* value for the specified string memory block. This function raises an exception if *strPtr* does not point within a valid memory block allocated for a string. Note that if the string has been deallocated, this function returns zero, it does not raise an exception.