



Memory Management in Mac OS

Mindfire Solutions

www.mindfiresolutions.com

March 6, 2002

Abstract:

This paper discusses memory management by macintosh operating system. This paper is a summarized form of “Inside Macintosh: Memory” and is directed towards developers who are new to Mac development but had previous development experience on other operating system. After going through this article you will be familiar with memory architecture in Mac, ways to allocate and deallocate memory, using temporary memory, A5 world, heap management, heap zones, heap fragmentation and several other features provided by memory manager.

MEMORY MANAGEMENT IN MAC OS.....	1
ORGANIZATION OF MEMORY IN MAC OS	3
<i>The System Heap</i>	<i>4</i>
<i>The System Global Variables.....</i>	<i>4</i>
ORGANIZATION OF MEMORY IN AN APPLICATION PARTITION 1.....	4
<i>The Application Stack 1</i>	<i>6</i>
<i>The Application Heap 1</i>	<i>6</i>
<i>The Application Global Variables and A5 World 1</i>	<i>7</i>
TEMPORARY MEMORY 1	8
VIRTUAL MEMORY 1	10
ADDRESSING MODES 1	10
HEAP MANAGEMENT.....	10
<i>Relocatable and Nonrelocatable Blocks 1.....</i>	<i>10</i>
<i>Properties of Relocatable Blocks 1</i>	<i>12</i>
<i>Locking and Unlocking Relocatable Blocks 1</i>	<i>13</i>
<i>Purging and Reallocating Relocatable Blocks</i>	<i>13</i>
MEMORY RESERVATION 1	14
HEAP PURGING AND COMPACTION 1	14
HEAP FRAGMENTATION 1	15
<i>Deallocating Nonrelocatable Blocks 1.....</i>	<i>15</i>
<i>Locking Relocatable Blocks 1</i>	<i>15</i>
<i>Allocating Nonrelocatable Blocks 1.....</i>	<i>16</i>
DANGLING POINTERS 1.....	16
<i>Callback Routines 1.....</i>	<i>17</i>
INVALID HANDLES 1	18
<i>Disposed Handles 1.....</i>	<i>18</i>



<i>Empty Handles 1</i>	18
<i>Fake Handles 1</i>	18
LOW-MEMORY CONDITIONS.....	19
<i>Grow-Zone Functions 1</i>	19
SETTING UP THE APPLICATION HEAP	19
<i>Changing the Size of the Stack 1</i>	19
<i>Expanding the Heap 1</i>	20
<i>Allocating Master Pointer Blocks 1</i>	20
<i>Defining a Grow-Zone Function 1</i>	21
ABOUT THE MEMORY MANAGER.....	21
TEMPORARY MEMORY 2	21
INSTALLING A PURGE-WARNING PROCEDURE 2	22
CREATING HEAP ZONES 2.....	22
BLOCK HEADERS 2.....	23
VIRTUAL MEMORY 3.....	23
MEMORY MANAGEMENT UTILITIES	24
USING QUICKDRAW GLOBAL VARIABLES IN STAND-ALONE CODE 4.....	24
THE A5 REGISTER 4	25
ACCESSING THE A5 WORLD IN COMPLETION ROUTINES 4	26
ACCESSING THE A5 WORLD IN INTERRUPT TASKS 4.....	27

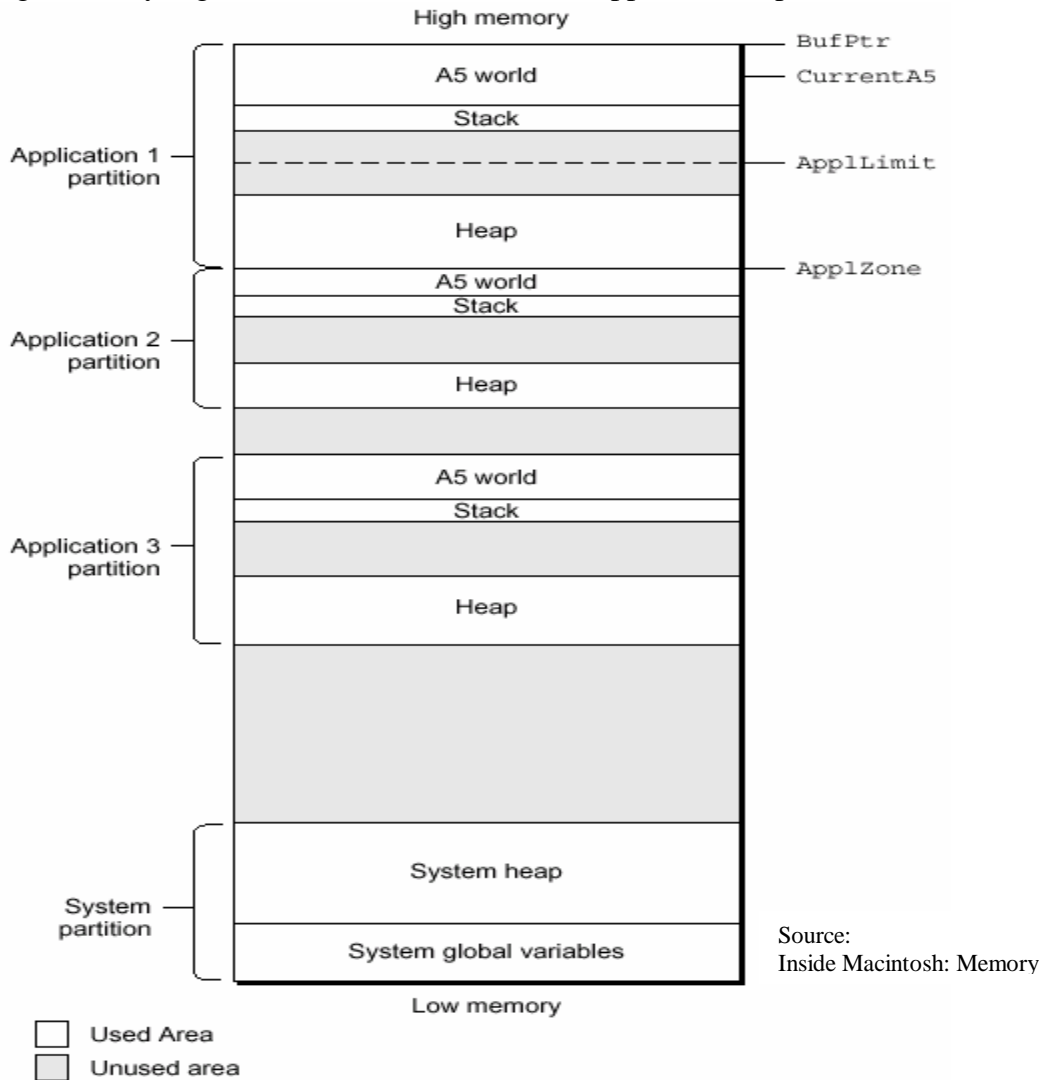


Organization of Memory in Mac OS

When the Macintosh Operating System starts up, it divides the available RAM into two broad sections. It reserves for itself a zone or **partition** of memory known as the **system partition**. The system partition always begins at the lowest addressable byte of memory (memory address 0) and extends upward.

All memory outside the system partition is available for allocation to applications or other software components. In system software version 7.0 and later (or when MultiFinder is running in system software versions 5.0 and 6.0), the user can have multiple applications open at once. When an application is launched, the Operating System assigns it a section of memory known as its **application partition**. In general, an application uses only the memory contained in its own application partition.

Fig: Memory organization in Mac with several applications open.





The System Heap

The main part of the system partition is an area of memory known as the **system heap**. In general, the system heap is reserved for exclusive use by the Operating System and other system software components, which load into it various items such as system resources, system code segments, and system data structures. All system buffers and queues, for example, are allocated in the system heap.

Hardware device drivers (stored as code resources of type 'DRVR') are loaded into the system heap when the driver is opened.

The System Global Variables

The lowest part of memory is occupied by a collection of global variables called **system global variables** (or **low-memory system global variables**). The Operating System uses these variables to maintain different kinds of information about the operating environment. For example, the `Ticks` global variable contains the number of ticks (sixtieths of a second) that have elapsed since the system was most recently started up.

Other low-memory global variables contain information about the current application. For example, the `AppZone` global variable contains the address of the first byte of the active application's partition. The `AppLimit` global variable contains the address of the last byte the active application's heap can expand to include. The `CurrentA5` global variable contains the address of the boundary between the active application's global variables and its application parameters. Because these global variables contain information about the active application, the Operating System changes the values of these variables whenever a context switch occurs.

Usually, when the value of a low-memory global variable is likely to be useful to applications, the system software provides a routine that you can use to read or write that value. For example, you can get the current value of the `Ticks` global variable by calling the `TickCount` function.

Organization of Memory in an Application Partition

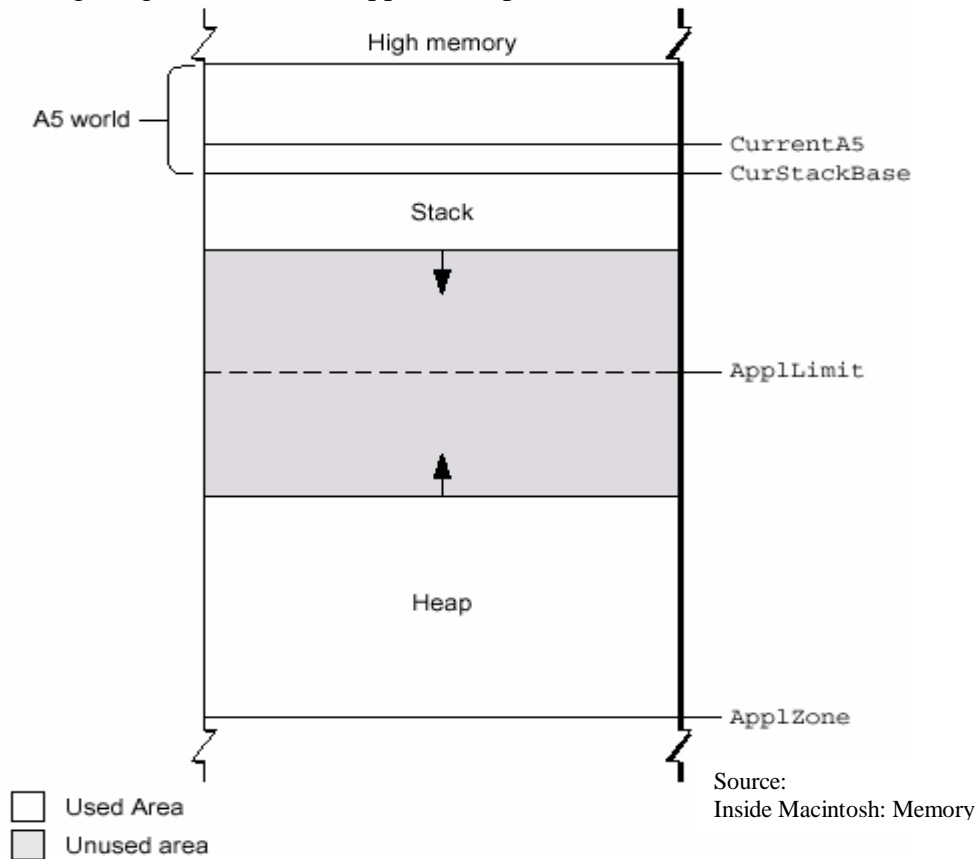
When your application is launched, the Operating System allocates for it a partition of memory called its **application partition**.

Your application partition is divided into three major parts:

1. the application stack
2. the application heap
3. the application global variables and A5 world



Fig: Organization of an application partition



The heap is located at the low-memory end of your application partition and always expands (when necessary) toward high memory. The A5 world is located at the high-memory end of your application partition and is of fixed size. The stack begins at the low-memory end of the A5 world and expands downward, toward the top of the heap.

There is usually an unused area of memory between the stack and the heap. This unused area provides space for the stack to grow without encroaching upon the space assigned to the application heap. In some cases, however, the stack might grow into space reserved for the application heap. If this happens, it is very likely that data in the heap will become corrupted. The `ApplLimit` global variable marks the upper limit to which your heap can grow. If you call the `MaxApplZone` procedure at the beginning of your program, the heap immediately extends all the way up to this limit. If you were to use all of the heap's free space, the Memory Manager would not allow you to allocate additional blocks above `ApplLimit`. If you do not call `MaxApplZone`, the heap grows toward `ApplLimit` whenever the Memory Manager finds that there is not enough memory in the heap to fill a request. However, once the heap grows up to `ApplLimit`, it can grow no further. Thus, whether you maximize your application heap or not, you can use only the space between the bottom of the heap and `ApplLimit`.



Unlike the heap, the stack is not bounded by `AppLLimit`. If your application uses heavily nested procedures with many local variables or uses extensive recursion, the stack could grow downward beyond `AppLLimit`. Because you do not use Memory Manager routines to allocate memory on the stack, the Memory Manager cannot stop your stack from growing beyond `AppLLimit` and possibly encroaching upon space reserved for the heap. However, a vertical retrace task checks approximately 60 times each second to see if the stack has moved into the heap. If it has, the task, known as the “stack sniffer,” generates a system error. This system error alerts you that you have allowed the stack to grow too far, so that you can make adjustments.

Note: To ensure during debugging that your application generates this system error if the stack extends beyond `AppLLimit`, you should call `MaxAppLZone` at the beginning of your program to expand the heap to `AppLLimit`.

The Application Stack

The **stack** is an area of memory in your application partition that can grow or shrink at one end while the other end remains fixed. This means that space on the stack is always allocated and released in LIFO (last-in, first-out) order. The last item allocated is always the first to be released. It also means that the allocated area of the stack is always contiguous. Space is released only at the top of the stack, never in the middle, so there can never be any unallocated “holes” in the stack.

By convention, the stack grows from high memory toward low memory addresses. The end of the stack that grows or shrinks is usually referred to as the “top” of the stack, even though it’s actually at the lower end of memory occupied by the stack.

When your application calls a routine, space is automatically allocated on the stack for a stack frame. A **stack frame** contains the routine’s parameters, local variables, and return address.

Note: Dynamic memory allocation on the stack is usually handled automatically if you are using a high-level development language such as Pascal. The compiler generates the code that creates and deletes stack frames for each function or procedure call.

The Application Heap

An **application heap** is the area of memory in your application partition in which space is dynamically allocated and released on demand. The heap contains virtually all items that are not allocated on the stack. For instance, your application heap contains the application’s code segments and resources that are currently loaded into memory. The heap also contains other dynamically allocated items such as window records, dialog records, document data, and so forth.



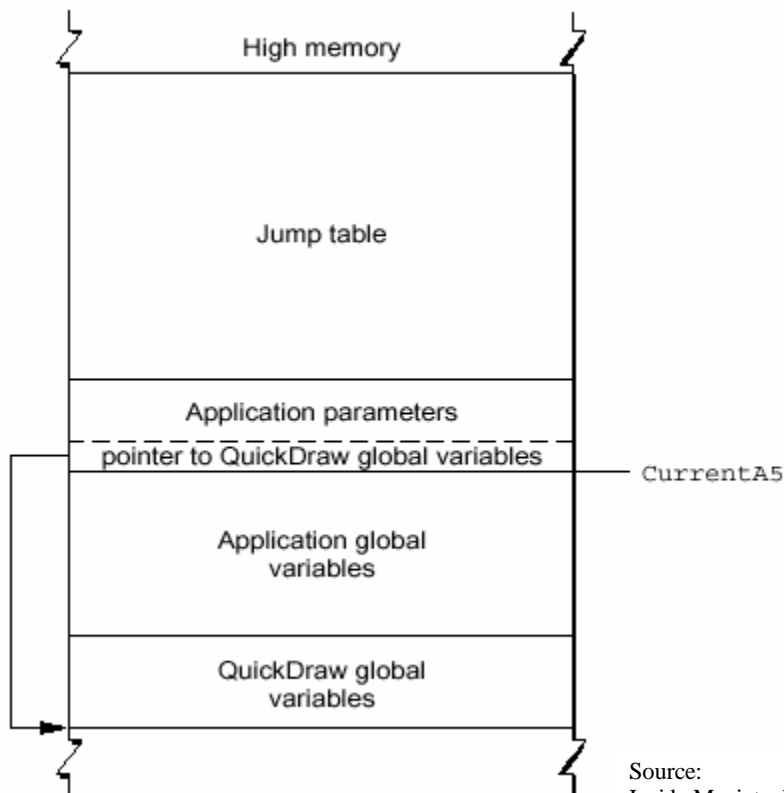
The Memory Manager does all the necessary housekeeping to keep track of blocks in the heap as they are allocated and released. Because these operations can occur in any order, the heap doesn't usually grow and shrink in an orderly way, as the stack does. Instead, after your application has been running for a while, the heap can tend to become fragmented into a patchwork of allocated and free blocks. This fragmentation is known as **heap fragmentation**.

One result of heap fragmentation is that the Memory Manager might not be able to satisfy your application's request to allocate a block of a particular size. Even though there is enough free space available, the space is broken up into blocks smaller than the requested size. When this happens, the Memory Manager tries to create the needed space by moving allocated blocks together, thus collecting the free space in a single larger block. This operation is known as **heap compaction**.

Heap fragmentation is generally not a problem as long as the blocks of memory you allocate are free to move during heap compaction. There are, however, two situations in which a block is not free to move: when it is a nonrelocatable block, and when it is a locked, relocatable block.

The Application Global Variables and A5 World

Fig: Organization of an application's A5 world



Source:
Inside Macintosh: Memory



Your application's global variables are stored in an area of memory near the top of your application partition known as the application **A5 world**. The A5 world contains four kinds of data:

1. application global variables
2. application QuickDraw global variables
3. application parameters
4. the application's jump table

Each of these items is of fixed size, although the sizes of the global variables and of the jump table may vary from application to application.

The system global variable `CurrentA5` points to the boundary between the current application's global variables and its application parameters. For this reason, the application's global variables are found as negative offsets from the value of `CurrentA5`. This boundary is important because the Operating System uses it to access the following information from your application: its global variables, its QuickDraw global variables, the application parameters, and the jump table. This information is known collectively as the A5 world because the Operating System uses the microprocessor's A5 register to point to that boundary.

Your application's **QuickDraw global variables** contain information about its drawing environment. For example, among these variables is a pointer to the current graphics port.

Your application's **jump table** contains an entry for each of your application's routines that is called by code in another segment. The Segment Manager uses the jump table to determine the address of any externally referenced routines called by a code segment.

The **application parameters** are 32 bytes of memory located above the application global variables; they're reserved for use by the Operating System. The first long word of those parameters is a pointer to your application's QuickDraw global variables.

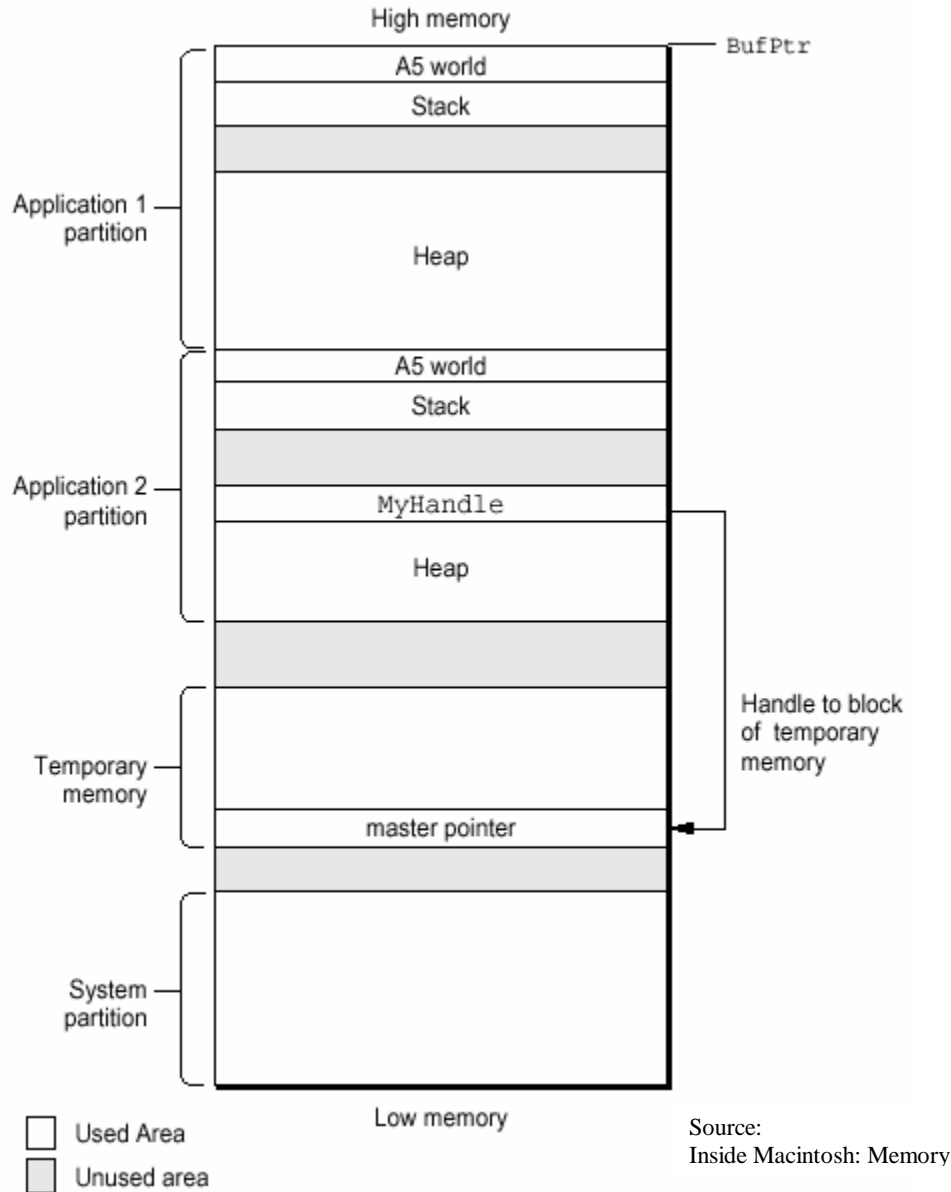
Temporary Memory

In the Macintosh multitasking environment, each application is limited to a particular memory partition (whose size is determined by information in the 'SIZE' resource of that application). The size of your application's partition places certain limits on the size of your application heap and hence on the sizes of the buffers and other data structures that your application uses. In general, you specify an application partition size that is large enough to hold all the buffers, resources, and other data that your application is likely to need during its execution.

If for some reason you need more memory than is currently available in your application heap, you can ask the Operating System to let you use any available memory that is not yet allocated to any other application. This memory, known as **temporary memory**, is allocated from the available unused RAM; usually, that memory is not contiguous with the memory in your application's zone.



Fig: Using temporary memory allocated from unused RAM



Your application should use temporary memory only for occasional short-term purposes that could be accomplished in less space, though perhaps less efficiently. For example, if you want to copy a large file, you might try to allocate a fairly large buffer of temporary memory. If you receive the temporary memory, you can copy data from the source file into the destination file using the large buffer. If, however, the request for temporary memory fails, you can instead use a smaller buffer within your application heap.

One good reason for using temporary memory only occasionally is that you cannot assume that you will always receive the temporary memory you request.



Virtual Memory

In system software version 7.0 and later, suitably equipped Macintosh computers can take advantage of a feature of the Operating System known as **virtual memory**, by which the machines have a logical address space that extends beyond the limits of the available physical memory.

It is important to realize that virtual memory operates transparently to most applications. Unless your application has time-critical needs that might be adversely affected by the operation of virtual memory or installs routines that execute at interrupt time, you do not need to know whether virtual memory is operating.

Addressing Modes

On suitably equipped Macintosh computers, the Operating System supports **32-bit addressing**, that is, the ability to use 32 bits to determine memory addresses. Earlier versions of system software use 24-bit addressing, where the upper 8 bits of memory addresses are ignored or used as flag bits. When 32-bit addressing is in operation, the maximum program address space is 1 GB.

Heap Management

Relocatable and Nonrelocatable Blocks

You can use the Memory Manager to allocate two different types of blocks in your heap: nonrelocatable blocks and relocatable blocks. A **nonrelocatable block** is a block of memory whose location in the heap is fixed. In contrast, a **relocatable block** is a block of memory that can be moved within the heap (perhaps during heap compaction).

To reference a nonrelocatable block, you can use a **pointer** variable, defined by the `Ptr` data type.

A pointer is simply the address of an arbitrary byte in memory, and a pointer to a nonrelocatable block of memory is simply the address of the first byte in the block. Because a pointer is the address of a block of memory that cannot be moved, all copies of the pointer correctly reference the block as long as you don't dispose of it.

The pointer variable itself occupies 4 bytes of space in your application partition. Often the pointer variable is a global variable and is therefore contained in your application's A5 world. But the pointer can also be allocated on the stack or in the heap itself. To reference relocatable blocks, the Memory Manager uses a scheme known as **double indirection**. The Memory Manager keeps track of a relocatable block internally with a **master pointer**, which itself is part of a nonrelocatable **master pointer block** in your application heap and can never move.



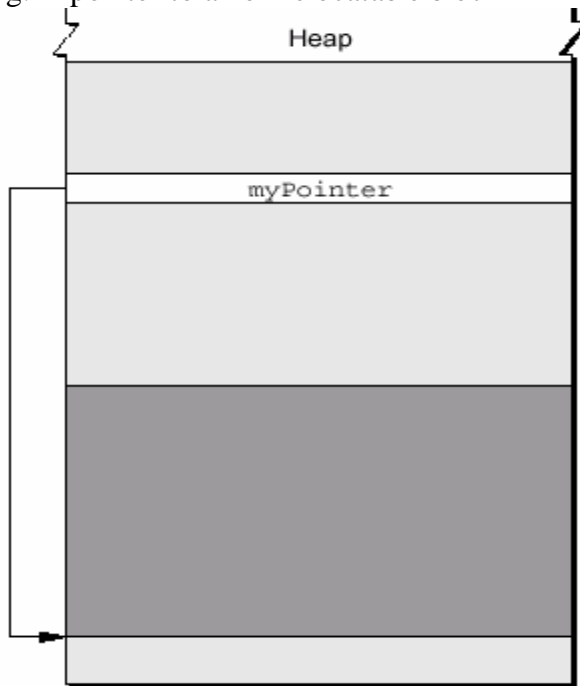
Note: The Memory Manager allocates one master pointer block (containing 64 master pointers) for your application at launch time, and you can call the `MoreMasters` procedure to request that additional master pointer blocks be allocated.



When the Memory Manager moves a relocatable block, it updates the master pointer so that it always contains the address of the relocatable block. You reference the block with a **handle**, defined by the `Handle` data type.

A handle is a Memory Manager structure which is basically a pointer to a pointer (the first pointer is a handle, the second one is called a master pointer). Your code remembers the handle, and then, to resize it, the Memory Manager can free the master pointer and allocate new space anywhere in the heap. You allocate a handle using `NewHandle` and you release it using `DisposeHandle`.

One of the most common times that you will use handles is when dealing with resources. All resources on the Macintosh are allocated as handles. You allocate a resource handle using `GetResource` and you release it using `ReleaseResource` or by closing the resource file.

Fig: A pointer to a non relocatable block

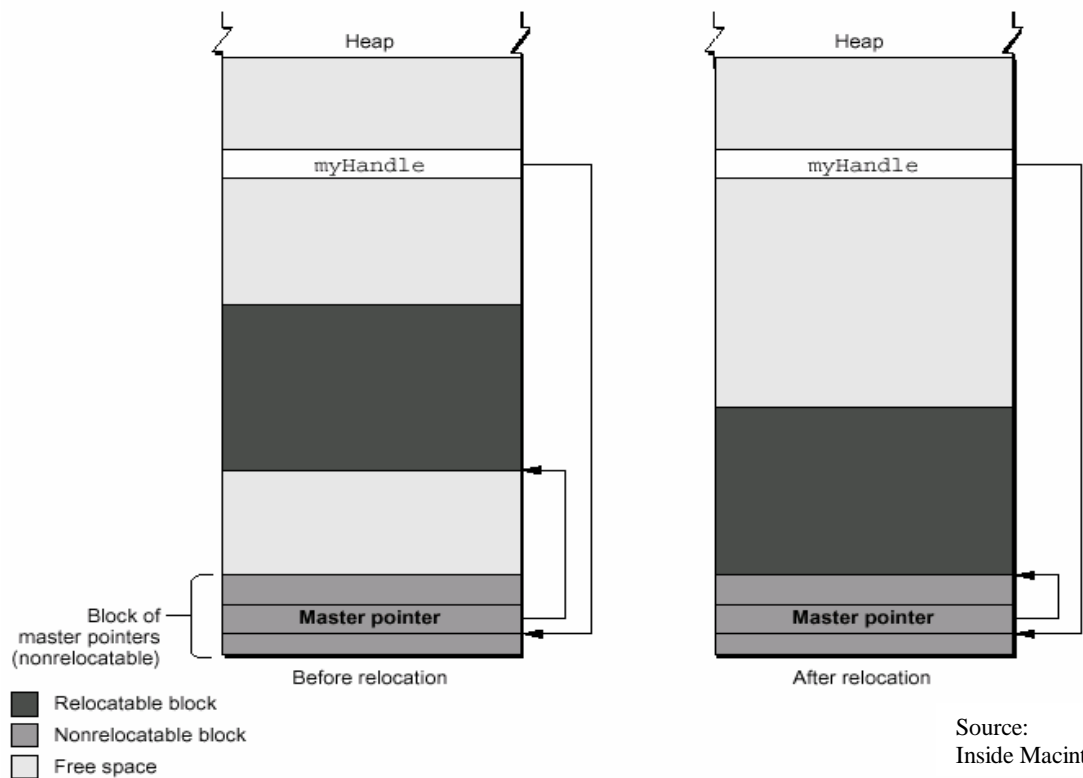


-  Nonrelocatable
-  Free space

Source:
Inside Macintosh: Memory



Fig: A handle to a relocatable block



Master pointers for relocatable objects in your heap are always allocated in your application heap. Because the blocks of masters pointers are nonrelocatable, it is best to allocate them as low in your heap as possible. You can do this by calling the `MoreMasters` procedure when your application starts up.

In some cases, however, you may be forced to allocate a nonrelocatable block of memory. When you call the Window Manager function `NewWindow`, for example, the Window Manager internally calls the `NewPtr` function to allocate a new nonrelocatable block in your application partition.

Using relocatable blocks makes the Memory Manager more efficient at managing available space, but it does carry some overhead. As you have seen, the Memory Manager must allocate extra memory to hold master pointers for relocatable blocks.

Properties of Relocatable Blocks

If relocatable, a block can be either locked or unlocked; if it's unlocked, a block can be either purgeable or un-purgeable. These attributes of relocatable blocks can be set and changed as necessary



Locking and Unlocking Relocatable Blocks

To prevent a block from moving, you can **lock** it, using the `HLock` procedure. Once you have locked a block, it won't move. Later, you can **unlock** it, using the `HUnlock` procedure, allowing it to move again. In general, you need to lock a relocatable block only if there is some danger that it might be moved during the time that you read or write the data in that block. This might happen, for instance, if you dereference a handle to obtain a pointer to the data and (for increased speed) use the pointer within a loop that calls routines that might cause memory to be moved. If, within the loop, the block whose data you are accessing is in fact moved, then the pointer no longer points to that data; this pointer is said to dangle.

Note : Locking a block is only one way to prevent a dangling pointer.

Using locked relocatable blocks can, however, slow the Memory Manager down as much as using nonrelocatable blocks. The Memory Manager can't move locked blocks. In addition, except when you allocate memory and resize relocatable blocks, it can't move relocatable blocks around locked relocatable blocks (just as it can't move them around nonrelocatable blocks). Thus, locking a block in the middle of the heap for long periods of time can increase heap fragmentation.

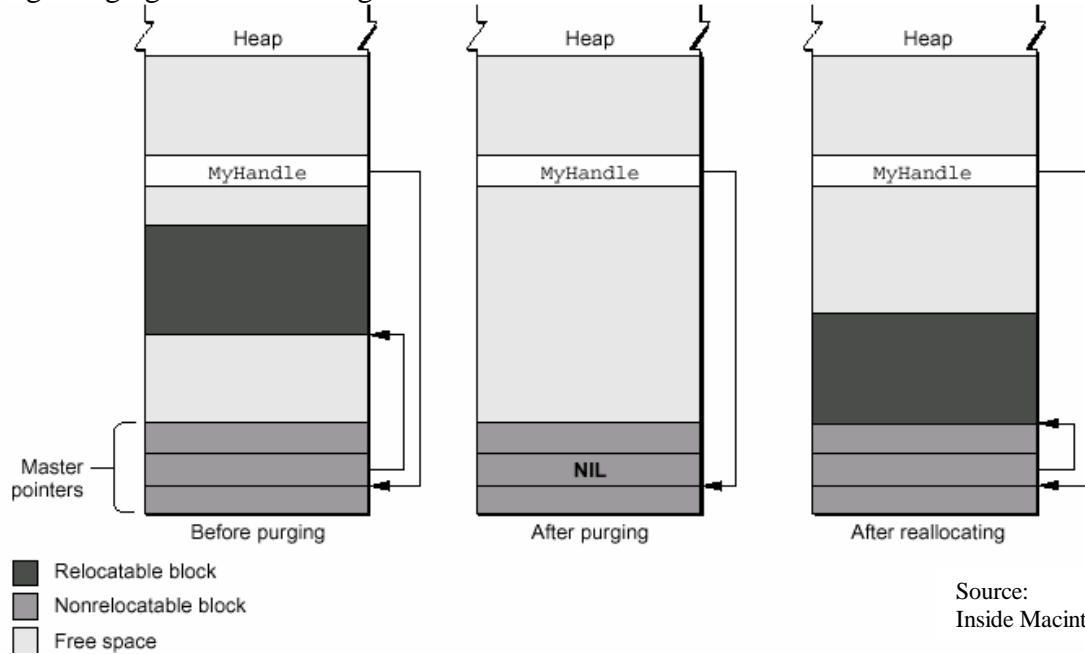
Locking and unlocking blocks every time you want to prevent a block from moving can become troublesome. Fortunately, the Memory Manager moves unlocked, relocatable blocks only at well-defined, predictable times. In general, each routine description in *Inside Macintosh* indicates whether the routine could move or purge memory. If you do not call any of those routines in a section of code, you can rely on all blocks to remain stationary while that code executes. Note that the Segment Manager might move memory if you call a routine located in a segment that is not currently resident in memory.

Purging and Reallocating Relocatable Blocks

By making a relocatable block **purgeable**, you allow the Memory Manager to free the space it occupies if necessary. If you later want to prohibit the Memory Manager from freeing the space occupied by a relocatable block, you can make the block **unpurgeable**. You can use the `HPurge` and `HNoPurge` procedures to change back and forth between these two states. A block you create by calling `NewHandle` is initially unpurgeable. Once you make a relocatable block purgeable, you should subsequently check handles to that block before using them if you call any of the routines that could move or purge memory. If a handle's master pointer is set to `NIL`, then the Operating System has purged its block. To use the information formerly in the block, you must reallocate space for it (perhaps by calling the `ReallocateHandle` procedure) and then reconstruct its contents (for example, by rereading the preferences file).



Fig: Purging and reallocating a relocatable block



Source:
Inside Macintosh: Memory

Memory Reservation

The Memory Manager does its best to prevent situations in which nonrelocatable blocks in the middle of the heap trap relocatable blocks. When it allocates new nonrelocatable blocks, it attempts to **reserve** memory for them as low in the heap as possible. The Memory Manager reserves memory for a nonrelocatable block by moving unlocked relocatable blocks upward until it has created a space large enough for the new block. When the Memory Manager can successfully pack all nonrelocatable blocks into the bottom of the heap, no nonrelocatable block can trap a relocatable block, and it has successfully prevented heap fragmentation.

During this process, the Memory Manager might even move a relocatable block over a nonrelocatable block to make room for another nonrelocatable block.

Heap Purging and Compaction

When your application attempts to allocate memory (for example, by calling either the `NewPtr` or `NewHandle` function), the Memory Manager might need to **compact** or **purge** the heap to free memory and to fuse many small free blocks into fewer large free blocks. The Memory Manager first tries to obtain the requested amount of space by compacting the heap; if compaction fails to free the required amount of space, the Memory Manager then purges the heap.

When compacting the heap, the Memory Manager moves unlocked, relocatable blocks down until they reach nonrelocatable blocks or locked, relocatable blocks. You can compact the heap manually, by calling either the `CompactMem` function or the `MaxMem` function.



If you want, you can manually purge a few blocks or an entire heap in anticipation of a memory shortage. To purge an individual block manually, call the `EmptyHandle` procedure. To purge your entire heap manually, call the `PurgeMem` procedure or the `MaxMem` function.

Heap Fragmentation

Throughout this section, you should keep in mind the following rule: the Memory Manager can move a relocatable block around a nonrelocatable block (or a locked relocatable block) at these times only:

When the Memory Manager reserves memory for a nonrelocatable block (or when you manually reserve memory before allocating a block), it can move unlocked, relocatable blocks upward over nonrelocatable blocks to make room for the new block as low in the heap as possible.

When you attempt to resize a relocatable block, the Memory Manager can move that block around other blocks if necessary.

In contrast, the Memory Manager cannot move relocatable blocks over nonrelocatable blocks during compaction of the heap.

Deallocating Nonrelocatable Blocks

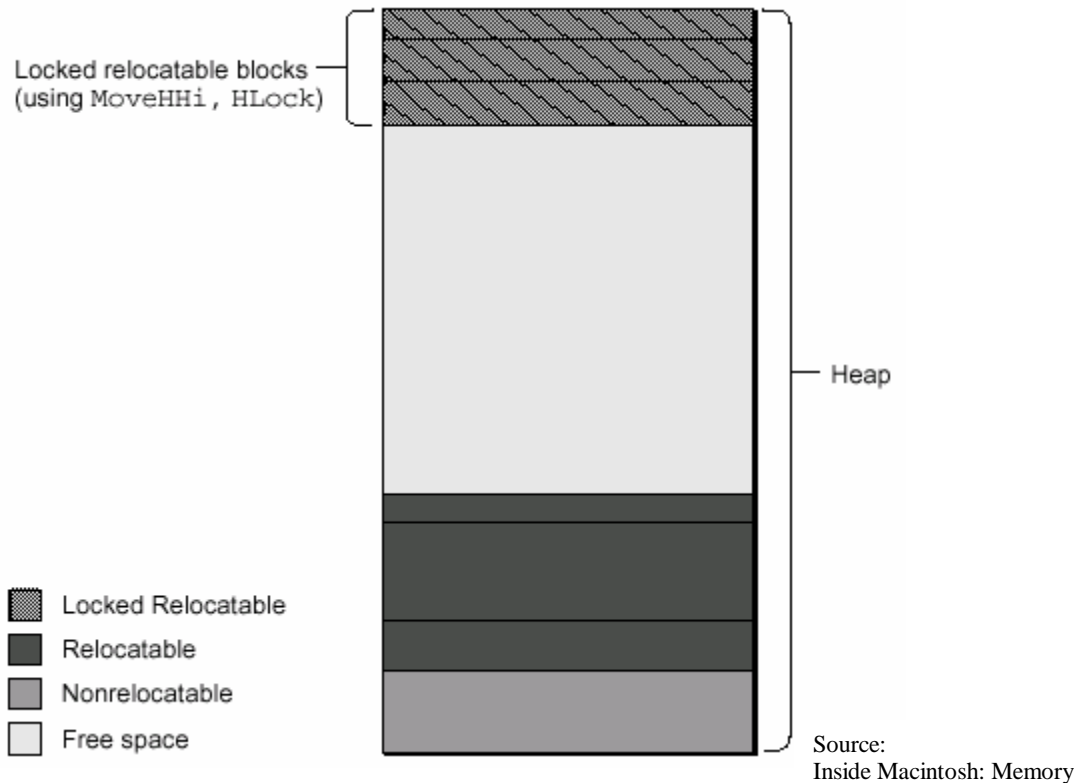
One of the most common causes of heap fragmentation is also one of the most difficult to avoid. The problem occurs when you dispose of a nonrelocatable block in the middle of the pile of nonrelocatable blocks at the bottom of the heap. Unless you immediately allocate another nonrelocatable block of the same size, you create a gap where the nonrelocatable block used to be. If you later allocate a slightly smaller, nonrelocatable block, that gap shrinks. However, small gaps are inefficient because of the small likelihood that future memory allocations will create blocks small enough to occupy the gaps. It would not matter if the first block you allocated after deleting the nonrelocatable block were relocatable. The Memory Manager would place the block in the gap if possible. If you were later to allocate a nonrelocatable block as large as or smaller than the gap, the new block would take the place of the relocatable block, which would join other relocatable blocks in the middle of the heap, as desired. However, the new nonrelocatable block might be smaller than the original nonrelocatable block, leaving a small gap.

Locking Relocatable Blocks

Locked relocatable blocks present a special problem. When relocatable blocks are locked, they can cause as much heap fragmentation as nonrelocatable blocks. One solution is to reserve memory for all relocatable blocks that might at some point need to be locked, and to leave them locked for as long as they are allocated. This solution has drawbacks, however, because then the blocks would lose any flexibility that being relocatable otherwise gives them. Deleting a locked relocatable block can create a gap, just as deleting a nonrelocatable block can.



Fig: An effectively partitioned heap



Allocating Nonrelocatable Blocks

As you have seen, there are two reasons for not allocating nonrelocatable blocks during the middle of your application's execution. First, if you also dispose of nonrelocatable blocks in the middle of your application's execution, then allocation of new nonrelocatable blocks is likely to create small gaps, as discussed earlier. Second, even if you never dispose of nonrelocatable blocks until your application terminates, memory reservation is an imperfect process, and the Memory Manager could occasionally place new nonrelocatable blocks above relocatable blocks.

Dangling Pointers

Accessing a relocatable block by double indirection, through its handle instead of through its master pointer, requires an extra memory reference. For efficiency, you might sometimes want to **dereference** the handle—that is, make a copy of the block's master pointer—and then use that pointer to access the block by single indirection. When you do this, however, you need to be particularly careful. Any operation that allocates space from the heap might cause the relocatable block to be moved or purged. In that event, the block's master pointer is correctly updated, but your copy of the master pointer is not. As a result, your copy of the master pointer is a **dangling pointer**. Dangling pointers are likely to make your application crash or produce garbled output.



The easiest way to prevent dangling pointers is simply to lock the relocatable block whose data you want to read or write.

Callback Routines

Code segmentation can also lead to a different type of dangling-pointer problem when you use callback routines. The problem rarely arises, but it is difficult to debug. Some Toolbox routines require that you pass a pointer to a procedure in a variable of type `ProcPtr`. Ordinarily, it does not matter whether the procedure you pass in such a variable is in the same code segment as the routine that calls it or in a different code segment. For example, suppose you call `TrackControl` as follows:

```
myPart := TrackControl(myControl, myEvent.where,  
@MyCallback);
```

If `MyCallback` were in the same code segment as this line of code, then a compiler would pass to `TrackControl` the absolute address of the `MyCallback` procedure. If it were in a different code segment, then the compiler would take the address from the jump table entry for `MyCallback`. Either way, `TrackControl` should call `MyCallback` correctly.

Occasionally, you might use a variable of type `ProcPtr` to hold the address of a callback procedure and then pass that address to a routine. Here is an example:

```
myProc := @MyCallback;
```

```
...
```

```
myPart := TrackControl(myControl, myEvent.where, myProc);
```

As long as these lines of code are in the same code segment and the segment is not unloaded between the execution of those lines, the preceding code should work perfectly. Suppose, however, that `myProc` is a global variable, and the first line of the code is in a different segment from the call to `TrackControl`. Suppose, further, that the `MyCallback` procedure is in the same segment as the first line of the code (which is in a different segment from the call to `TrackControl`). Then, the compiler might place the absolute address of the `MyCallback` routine into the variable `myProc`. The compiler cannot realize that you plan to use the variable in a different code segment from the one that holds both the routine you are referencing and the routine you are using to initialize the `myProc` variable. Because `MyCallback` and the call to `TrackControl` are in different code segments, the `TrackControl` procedure requires that you pass an address in the jump table, not an absolute address. Thus, in this hypothetical situation, `myProc` would reference `MyCallback` incorrectly. To avoid this problem, make sure to place in the same segment any code in which you assign a value to a variable of type `ProcPtr` and any code in which you use that variable. If you must put them in different code segments, then be sure that you place the callback routine in a code segment different from the one that initializes the variable.

Note: Some development systems allow you to specify compiler options that force jump table references to be generated for routine addresses. If you specify those options, the problems described in this section cannot arise.



Invalid Handles

An invalid handle refers to the wrong area of memory, just as a dangling pointer does. There are three types of invalid handles: empty handles, disposed handles, and fake handles.

Disposed Handles

A **disposed handle** is a handle whose associated relocatable block has been disposed of. When you dispose of a relocatable block (perhaps by calling the procedure `DisposeHandle`), the Memory Manager does not change the value of any handle variables that previously referenced that block. Instead, those variables still hold the address of what once was the relocatable block's master pointer. Because the block has been disposed of, however, the contents of the master pointer are no longer defined. (The master pointer might belong to a subsequently allocated relocatable block, or it could become part of a linked list of unused master pointers maintained by the Memory Manager.)

You can avoid these problems quite easily by assigning the value `NIL` to the handle variable after you dispose of its associated block.

Empty Handles

An **empty handle** is a handle whose master pointer has the value `NIL`. When the Memory Manager purges a relocatable block, for example, it sets the block's master pointer to `NIL`. The space occupied by the master pointer itself remains allocated, and handles to the purged block continue to point to the master pointer. This is useful, because if you later reallocate space for the block by calling `ReallocateHandle`, the master pointer will be updated and all existing handles will correctly access the reallocated block.

Note: Don't confuse empty handles with **0-length handles**, which are handles whose associated block has a size of 0 bytes. A 0-length handle has a non-`NIL` master pointer and a block header.

Fake Handles

A **fake handle** is a handle that was not created by the Memory Manager. Normally, you create handles by either directly or indirectly calling the Memory Manager function `NewHandle` (or one of its variants, such as `NewHandleClear`). You create a fake handle—usually inadvertently—by directly assigning a value to a variable of type `Handle`, as illustrated in following listing.

Listing: Creating a fake handle

```
FUNCTION MakeFakeHandle: Handle; {DON'T USE THIS FUNCTION!}  
CONST  
kMemoryLoc = $100; {a random memory location}
```



VAR

Low-Memory Conditions

You can take several steps to help maximize the amount of free space in your heap. For example, you can mark as purgeable any relocatable blocks whose contents could easily be reconstructed.

Before you call `NewHandle` or `NewPtr`, you should check that, if the requested amount of memory were in fact allocated, the remaining amount of space free in the heap would not fall below a certain threshold. The free memory defined by that threshold is your **memory cushion**.

Grow-Zone Functions

The Memory Manager provides a particularly easy way for you to make sure that the emergency memory reserve is released when necessary. You can define a **grow-zone function** that is associated with your application heap. The Memory Manager calls your heap's grow-zone function only after other techniques of freeing memory to satisfy a memory request fail (that is, after compacting and purging the heap and extending the heap zone to its maximum size). The grow-zone function can then take appropriate steps to free additional memory. A grow-zone function might dispose of some blocks or make some un-purgeable blocks purgeable. When the function returns, the Memory Manager once again purges and compacts the heap and tries to reallocate memory. If there is still insufficient memory, the Memory Manager calls the grow-zone function again (but only if the function returned a nonzero value the previous time it was called). This mechanism allows your grow-zone function to release just a little bit of memory at a time. If the amount it releases at any time is not enough, the Memory Manager calls it again and gives it the opportunity to take more drastic measures. As the most drastic step to freeing memory in your heap, you can release the emergency reserve.

Setting Up the Application Heap

To help prevent heap fragmentation, you should also perform some setup of your own early in your application's execution. Depending on the needs of your application, you might want to

- change the size of your application's stack
- expand the heap to the heap limit
- allocate additional master pointer blocks

The following sections describe in detail how and when to perform these operations.

Changing the Size of the Stack

Most applications allocate space on their stack in a predictable way and do not need to monitor stack space during their execution. For these applications, stack usage usually reaches a maximum in some heavily nested routine. If the stack in your application can



never grow beyond a certain size, then to avoid collisions between your stack and heap you simply need to ensure that your stack is large enough to accommodate that size. If you never encounter system error 28 (generated by the stack sniffer when it detects a collision between the stack and the heap) during application testing, then you probably do not need to increase the size of your stack. To increase the size of your stack, you simply reduce the size of your heap. Because the heap cannot grow above the boundary contained in the `AppLLimit` global variable, you can lower the value of `AppLLimit` to limit the heap's growth. By lowering `AppLLimit`, technically you are not making the stack bigger; you are just preventing collisions between it and the heap.

By default, the stack can grow to 8 KB on Macintosh computers without Color QuickDraw and to 32 KB on computers with Color QuickDraw.

Listing below defines a procedure that increases the stack size by a given value. It does so by determining the current heap limit, subtracting the value of the `extraBytes` parameter from that value, and then setting the application limit to the difference.

Listing : Increasing the amount of space allocated for the stack

```
PROCEDURE IncreaseStackSize (extraBytes: Size);
BEGIN
SetAppLLimit(Ptr(ORD4(GetAppLLimit) - extraBytes));
END;
```

You should call this procedure at the beginning of your application, before you call the `MaxAppLZone` procedure.

Expanding the Heap

Near the beginning of your application's execution, before you allocate any memory, you should call the `MaxAppLZone` procedure to expand the application heap immediately to the application heap limit. If you do not do this, the Memory Manager gradually expands your heap as memory needs require. This gradual expansion can result in significant heap fragmentation if you have previously moved relocatable blocks to the top of the heap (by calling `MoveHHi`) and locked them (by calling `HLock`). When the heap grows beyond those locked blocks, they are no longer at the top of the heap. Your heap then remains fragmented for as long as those blocks remain locked.

Allocating Master Pointer Blocks

After calling `MaxAppLZone`, you should call the `MoreMasters` procedure to allocate as many new nonrelocatable blocks of master pointers as your application is likely to need during its execution. Each block of master pointers in your application heap contains 64 master pointers. The Operating System allocates one block of master pointers as your application is loaded into memory, and every relocatable block you allocate needs one master pointer to reference it.



Defining a Grow-Zone Function

The Memory Manager calls your heap's grow-zone function only after other attempts to obtain enough memory to satisfy a memory allocation request have failed. A grow-zone function should be of the following form:

```
FUNCTION MyGrowZone (cbNeeded: Size): LongInt;
```

The Memory Manager passes to your function (in the `cbNeeded` parameter) the number of bytes it needs. Your function can do whatever it likes to free that much space in the heap.

```
PROCEDURE InitializeEmergencyMemory;  
BEGIN  
gEmergencyMemory := NewHandle(kEmergencyMemorySize);  
SetGrowZone(@MyGrowZone);  
END;
```

About the Memory Manager

The Memory Manager provides a large number of routines that you can use to perform various operations on blocks within your application partition. You can use the Memory Manager to set up your application partition allocate and release both relocatable and nonrelocatable blocks in your application heap copy data from nonrelocatable blocks to relocatable blocks, and vice versa determine how much space is free in your heap determine the location of the top of your stack determine the size of a memory block and, if necessary, change that size change the properties of relocatable blocks install or remove a grow-zone function for your heap obtain the result code of the most recent Memory Manager routine executed. The Memory Manager also provides routines that you can use to access areas of memory outside your application partition. You can use the Memory Manager to n allocate memory outside your partition that is currently unused by any open application or by the Operating System allocate memory in the system heap This section describes the areas of memory that lie outside your application partition. It also describes multiple heap zones.

Temporary Memory

In the Macintosh multitasking environment, your application is limited to a particular memory partition (whose size is determined by information in the 'SIZE' resource of your application). The size of your application's partition places certain limits on the size of your application heap and hence on the sizes of the buffers and other data structures that your application can use.

If for some reason you need more memory than is currently available in your application heap, you can ask the Operating System to let you use any available memory that is not yet allocated to any other application. This memory, called **temporary memory**, is allocated from the available unused RAM; in general, that memory is not contiguous with the memory in your application's zone Your application should use temporary memory



only for occasional short-term purposes that could be accomplished in less space, though perhaps less efficiently.

Installing a Purge-Warning Procedure

You can define a **purge-warning procedure** that the Memory Manager calls whenever it is about to purge a block from your application heap. You can use this procedure to save the data in the block, if necessary, or to perform other processing in response to this notification.

Creating Heap Zones

You can create heap zones as subzones of your application heap zone or (in rare instances) either in space reserved for the application global variables or on the stack. You can also create heap zones in a block of temporary memory or within the system heap zone. This section describes how to create new heap zones by calling the `InitZone` procedure.

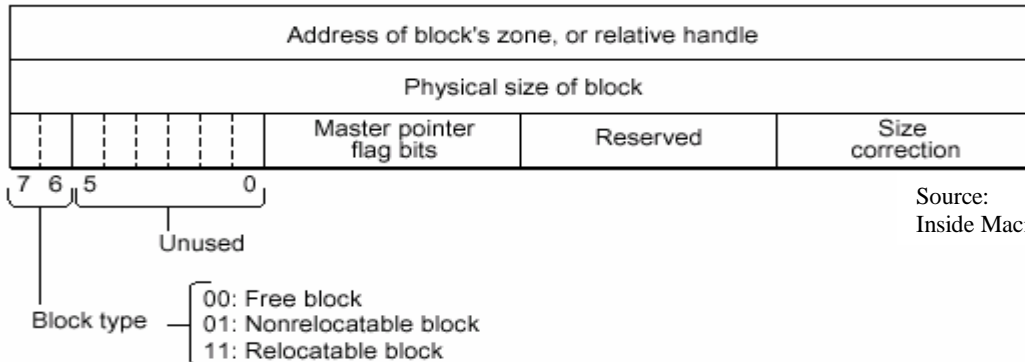
Listing : Creating a subzone of the original application heap zone

```
FUNCTION CreateSubZone: THz;  
CONST  
kZoneSize = 10240; {10K zone}  
kNumMasterPointers = 16; {num of master ptrs for new zone}  
VAR  
start: Ptr; {first byte in zone}  
limit: Ptr; {first byte beyond zone}  
BEGIN  
start := NewPtr(kZoneSize); {allocate storage for zone}  
IF MemError <> noErr THEN  
BEGIN {allocation successful}  
limit := Ptr(ORD4(start) + kZoneSize);  
{compute byte beyond end of zone}  
InitZone(NIL, kNumMasterPointers, limit, start);  
{initialize zone header, trailer}  
END;  
CreateSubZone := THz(start); {cast storage to a zone  
pointer}  
END;
```



Block Headers

Fig: A block header in a 32-bit zone



Every block in a heap zone, whether allocated or free, has a **block header** that the Memory Manager uses to find its way around in the zone. Block headers are completely transparent to your application. All pointers and handles to allocated blocks reference the beginning of the block's logical contents, following the end of the header. Similarly, whenever you use a variable of type `Size`, that variable refers to the number of bytes in the block's logical contents, not including the block header. That size is known as the block's **logical size**, as opposed to its **physical size**, the number of bytes it actually occupies in memory, including the header and any unused bytes at the end of the block.

Virtual Memory

A user can select (in the Memory control panel) whether to enable this larger or "virtual" address space. Most applications are completely unaffected by the operation of the Virtual Memory Manager and have no need to know whether any virtual memory is available. You might, however, need to intervene in the otherwise automatic workings of the Virtual Memory Manager if your application has critical timing requirements, executes code at interrupt time, or performs debugging operations.

The Virtual Memory Manager extends the logical address space by using part of the available secondary storage (such as a hard disk) to hold portions of applications and data that are not currently in use in physical memory. When an application needs to operate on portions of memory that have been transferred to disk, the Virtual Memory Manager loads those portions back into physical memory by making them trade places with other, unused segments of memory. This process of moving portions (or **pages**) of memory between physical RAM and the hard disk is called **paging**.

Users control and configure virtual memory through the Memory control panel. Controls in this panel allow the user to turn virtual memory on or off, set the size of virtual memory, and set the volume on which the invisible backing-store file resides. (The **backing-store file** is the file in which the Operating System stores the contents of nonresident pages of memory.)



Memory Management Utilities

Using QuickDraw Global Variables in Stand-Alone Code

If you are writing a stand-alone code segment such as a definition procedure for a window, menu, or control, you might want routines in that segment to examine the QuickDraw global variables of the current application. For example, you might want a control definition function to reference some of the QuickDraw global variables, such as `thePort`, `screenBits`, or the predefined patterns. Stand-alone segments, however, have no A5 world; if you try to link a stand-alone code segment that references your application's global variables, the linker may be unable to resolve those references. To solve this problem, you can have the definition function find the value of the application's A5 register (by calling the `SetCurrentA5` function) and then use that information to copy all of the application's QuickDraw global variables into a record in the function's own private storage. Listing below defines a record type with the same structure as the QuickDraw global variables. Note that `randSeed` is stored lowest in memory and `thePort` is stored highest in memory.

Listing : Structure of the QuickDraw global variables

```
TYPE
QDVarRecPtr = ^QDVarRec;
QDVarRec =
RECORD
randSeed: LongInt; {for random-number generator}
screenBits: BitMap; {rectangle enclosing screen}
arrow: Cursor; {standard arrow cursor}
dkGray: Pattern; {75% gray pattern}
ltGray: Pattern; {25% gray pattern}
gray: Pattern; {50% gray pattern}
black: Pattern; {all-black pattern}
white: Pattern; {all-white pattern}
thePort: GrafPtr; {pointer to current GrafPort}
END;
```

The location of these variables is linker-dependent. However, the A5 register always points to the last of these global variables, `thePort`. The Operating System references all other QuickDraw global variables as negative offsets from `thePort`. Therefore, you must dereference the value in A5 (to obtain the address of `thePort`), and then subtract the combined size of the other QuickDraw global variables from that address. The difference is a pointer to the first of the QuickDraw global variables, `randSeed`. You can copy the entire record into a local variable simply by dereferencing that pointer, as illustrated in Listing below.

Listing : Copying the QuickDraw global variables into a record



```
PROCEDURE GetQDVars (VAR qdVars: QDVarRec);
TYPE
LongPtr = ^LongInt;
BEGIN
qdVars := QDVarRecPtr(LongPtr(SetCurrentA5)^ -
(SizeOf(QDVarRec) - SizeOf(thePort)))^;
END;
```

Thereafter, your stand-alone code segment can read QuickDraw global variables through the structure returned by GetQDVars. Listing below defines a very simple draw routine for a control definition function. After reading the calling application's QuickDraw global variables, the draw routine paints a rectangle with a pattern.

Listing : A control's draw routine using the calling application's QuickDraw patterns

```
PROCEDURE DoDraw (varCode: Integer; myControl:
ControlHandle;
flag: Integer);
VAR
cRect: Rect;
qdVars: QDVarRec;
origPenState: PenState;
CONST
kDraw = 1; {constant to specify drawing}
BEGIN
GetPenState(origPenState); {get original pen state}
cRect := myControl^.ctrlRect; {get control's rectangle}
IF flag = kDraw THEN
BEGIN
GetQDVars(qdVars); {patterns are QD globals}
PenPat(qdVars.gray); {install desired pattern}
PaintRect(cRect); {paint the control}
END;
SetPenState(origPenState); {restore original pen state}
END;
```

The DoDraw drawing routine defined in Listing 4-6 retrieves the calling application's QuickDraw global variables and paints the control rectangle with a light gray pattern. It also saves and restores the pen state, because the PenPat procedure changes that state.

The A5 Register

If you write code that accesses your application's A5 world (usually to read or write the application global variables) at a time that your application is not the current application, you must ensure that the A5 register points to the boundary between your application's parameters and global variables. Because the Operating System accesses your A5 world relative to the address stored in the A5 register, you can obtain unpredictable results if you attempt to read or write data in your A5 world when the contents of A5 are not valid.



Accessing the A5 World in Completion Routines

Some Toolbox and Operating System routines require you to pass the address of an application-defined **callback routine**, usually in a variable of type `ProcPtr`. After a certain condition has been met, the Toolbox executes the specified routine. The exact time at which the Toolbox executes the routine varies. The timing of execution is determined by the Toolbox routine to which you passed the routine's address and the action that must be completed before the routine is called. Callback routines are quite common in the Macintosh system software. A grow-zone function, for instance, is an application-defined callback routine that is called every time the Memory Manager cannot find enough space in your heap to honor a memory-allocation request. Similarly, if your application plays a sound asynchronously, you can have the Sound Manager execute a **completion routine** after the sound is played. The completion routine might release the sound channel used to play the sound or perform other cleanup operations. In general, you cannot predict what your application will be doing when an asynchronous completion or callback routine is actually executed. The routine could be called while your application is executing code of its own or executing another Toolbox or Operating System routine.

Note: The completion or callback routine might even be called when your application is in the background. Before executing a completion or callback routine belonging to your application, the Process Manager checks whether your application is in the foreground. If not, the Process Manager performs a minor switch to give your application temporary control of the CPU.

Many Toolbox and Operating System routines do not need to access the calling application's global variables, QuickDraw global variables, or jump table. As a result, they sometimes use the A5 register for their own purposes. They save the current value of the register upon entry, modify the register as necessary, and then restore the original value on exit. As you can see, if one of these routines is executing when your callback routine is executed, your callback routine cannot depend on the value in the A5 register. This effectively prevents your callback routine from using any part of its A5 world. To solve this problem, simply use the strategy that the Toolbox employs when it takes over the A5 register: save the current value in the A5 register at the start of your callback procedure, install your application's A5 value, and then restore the original value when you exit. Listing below illustrates a very simple grow-zone function that uses this technique. It uses the `SetCurrentA5` and `SetA5` utilities to manipulate the A5 register.

Listing : A sample grow-zone function

```
FUNCTION MyGrowZone (cbNeeded: Size): LongInt;  
VAR  
theA5: LongInt; {value of A5 when function is called}  
BEGIN  
theA5 := SetCurrentA5; {remember current value of A5;  
install ours}
```



```
IF (gEmergencyMemory^ <> NIL) & (gEmergencyMemory <>
GZSaveHnd) THEN
BEGIN
EmptyHandle(gEmergencyMemory);
MyGrowZone := kEmergencyMemorySize;
END
ELSE
MyGrowZone := 0; {no more memory to release}
theA5 := SetA5(theA5); {restore previous value of A5}
END;
```

The function `SetCurrentA5` does two things: it returns the current value in the A5 register, and it sets the A5 register to the value of the `CurrentA5` low-memory global variable. This global variable always contains a value that points to the boundary between the current application's parameters and its global variables. The `MyGrowZone` function defined in Listing 4-1 calls `SetCurrentA5` on entry to make sure that it can read the value of the `gEmergencyMemory` global variable. The function `SetA5` also does two things: it returns the current value in the A5 register, and it sets the A5 register to whatever value you pass to the function. The `MyGrowZone` function calls `SetA5` with the original value of the A5 register as the parameter. In this case, the value returned by `SetA5` is ignored. There is no way to test whether, at the time your callback routine is called, your application is executing a Toolbox routine that could change the A5 register. Therefore, to be safe, you should save and restore the A5 register in any callback routine that accesses any part of your A5 world. Such routines include

- grow-zone functions
- Sound Manager completion routines
- File Manager I/O completion routines
- control-action procedures
- TextEdit word-break and click-loop routines
- trap patches
- custom menu definition, window definition, and control definition procedures

Accessing the A5 World in Interrupt Tasks

Sometimes, an application-defined routine executes at a time when you can't reliably call `SetCurrentA5`. For example, if your application is not the current application and you call `SetCurrentA5` as illustrated in Listing 4-1, the function will not return your application's value of `CurrentA5`. The `SetCurrentA5` function always returns the value of the low-memory global variable `CurrentA5`, which always belongs to the *current* application. You'll end up reading some other application's A5 world.

In general, you cannot reliably call `SetCurrentA5` in any code that is executed in response to an interrupt, including the following:

1. Time Manager tasks
2. VBL tasks
3. tasks installed using the Deferred Task Manager
4. Notification Manager response procedures



Instead of calling `SetCurrentA5` at interrupt time, you can call it at noninterrupt time when yours is the current application. Then store the returned value where you can read it at interrupt time. For example, the Notification Manager allows you to store information in the notification record passed to `NMInstall`. When you set up a notification record, you can use the `nmRefCon` field to hold the value in the A5 register. Listing below illustrates how to save the current value in the A5 register and pass that value to a response procedure.

Listing : Passing A5 to a notification response procedure

```
VAR
gMyNotification: NMRec; {a notification record}
BEGIN
WITH gMyNotification DO
BEGIN
qType := ORD(nmType); {set queue type}
nmMark := 1; {put mark in Application menu}
nmIcon := NIL; {no alternating icon}
nmSound := Handle(-1); {play system alert sound}
nmStr := NIL; {no alert box}
nmResp := @SampleResponse; {set response procedure}
nmRefCon := SetCurrentA5; {pass A5 to notification task}
END;
END;
```

The key step is to save the value of `CurrentA5` where the response procedure can find it—in this case, in the `nmRefCon` field. You must call `SetCurrentA5` at noninterrupt time; otherwise, you cannot be certain that it will return the correct value. When the notification response procedure is executed, its first task should be to call the `SetA5` function, which sets register A5 to the value stored in the `nmRefCon` field. At the end of the routine, the notification response procedure should call the `SetA5` function again to restore the previous value of register A5. Listing below shows a simple response procedure that sets up the A5 register, modifies a global variable, and then restores the A5 register.

Listing : Setting up and restoring the A5 register at interrupt time

```
PROCEDURE SampleResponse (nmReqPtr: NMRecPtr);
VAR
oldA5: LongInt; {A5 when procedure is called}
BEGIN
oldA5 := SetA5(nmReqPtr^.nmRefCon);
{set A5 to the application's A5}
gNotifReceived := TRUE; {set an application global }
{ to show alert was received}
oldA5 := SetA5(oldA5); {restore A5 to original value}
END;
```



Conclusion

After going through this article, you might have got a broad idea of what memory management in Mac alike and how it differs from the way other operating systems handles memory.

For more insight into Mac memory management concepts and associated Toolbox routines refer to:

<http://developer.apple.com/techpubs/macos8/OSSvcs/MemoryManager/memorymanager.html>

Mindfire Solutions is an IT and software services company in India, providing expert capabilities to the global market. Mindfire possesses experience in multiple platforms, and has built a strong track record of delivery. Our continued focus on fundamental strengths in computer science has led to a unique technology-led position in software services.

To explore the potential of working with Mindfire, please drop us an email at info@mindfiresolutions.com. We will be glad to talk with you.

To know more about Mindfire Solutions, please visit us on www.mindfiresolutions.com
