CHAPTER 1

Introduction to Memory Management

Contents

This chapter is a general introduction to memory management on Macintosh computers. It describes how the Operating System organizes and manages the available memory, and it shows how you can use the services provided by the Memory Manager and other system software components to manage the memory in your application partition effectively.

You should read this chapter if your application or other software allocates memory dynamically during its execution. This chapter describes how to

- set up your application partition at launch time
- determine the amount of free memory in your application heap
- allocate and dispose of blocks of memory in your application heap
- minimize fragmentation in your application heap caused by blocks of memory that cannot move
- implement a scheme to avoid low-memory conditions

You should be able to accomplish most of your application's memory allocation and management by following the instructions given in this chapter. If, however, your application needs to allocate memory outside its own partition (for instance, in the system heap), you need to read the chapter "Memory Manager" in this book. If your application has timing-critical requirements or installs procedures that execute at interrupt time, you need to read the chapter "Virtual Memory Manager" in this book. If your application's executable code is divided into multiple segments, you might also want to look at the chapter "Segment Manager" in *Inside Macintosh: Processes* for guidelines on how to divide your code into segments. If your application uses resources, you need to read the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* for information on managing memory allocated to resources.

This chapter begins with a description of how the Macintosh Operating System organizes the available physical random-access memory (RAM) in a Macintosh computer and how it allocates memory to open applications. Then this chapter describes in detail how the Memory Manager allocates blocks of memory in your application's heap and how to use the routines provided by the Memory Manager to perform the memory-management tasks listed above.

This chapter ends with descriptions of the routines used to perform these tasks. The "Memory Management Reference" and "Summary of Memory Management" sections in this chapter are subsets of the corresponding sections in the remaining chapters in this book.

About Memory

A Macintosh computer's available RAM is used by the Operating System, applications, and other software components, such as device drivers and system extensions. This section describes both the general organization of memory by the Operating System and the organization of the memory partition allocated to your application when it is launched. This section also provides a preliminary description of three related memory topics:

- temporary memory
- virtual memory
- 24- and 32-bit addressing

For more complete information on these three topics, you need to read the remaining chapters in this book.

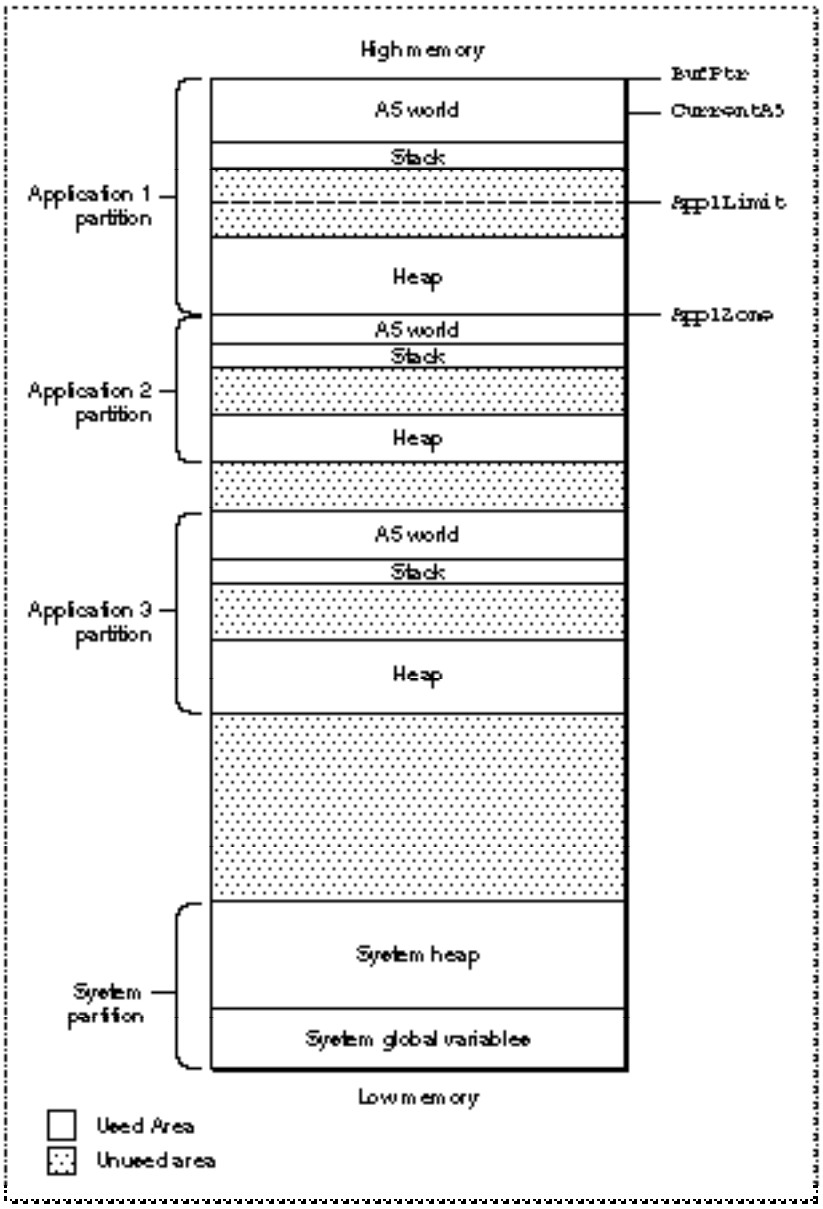
Organization of Memory by the Operating System

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. The system partition contains a system heap and a set of global variables, described in the next two sections.

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.

Figure 1-1 illustrates the organization of memory when several applications are open at the same time. The system partition occupies the lowest position in memory. Application partitions occupy part of the remaining space. Note that application partitions are loaded into the top part of memory first.

Figure 1-1 Memory organization with several applications open



In Figure 1-1, three applications are open, each with its own application partition. The application labeled Application 1 is the active application. (The labels on the right side of the figure are system global variables, explained in “The System Global Variables” on page 1-6.)

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.

The system heap is also used for code and other resources that do not belong to specific applications, such as code resources that add features to the Operating System or that provide control of special-purpose peripheral equipment. System patches and system extensions (stored as code resources of type `'INIT'`) are loaded into the system heap during the system startup process. Hardware device drivers (stored as code resources of type `'DRVR'`) are loaded into the system heap when the driver is opened.

Most applications don't need to load anything into the system heap. In certain cases, however, you might need to load resources or code segments into the system heap. For example, if you want a vertical retrace task to continue to execute even when your application is in the background, you need to load the task and any data associated with it into the system heap. Otherwise, the Vertical Retrace Manager ignores the task when your application is in the background.

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. Similar variables contain, for example, the height of the menu bar (`MBarHeight`) and pointers to the heads of various operating-system queues (`DTQueue`, `FSQHdr`, `VBLQueue`, and so forth). Most low-memory global variables are of this variety: they contain information that is generally useful only to the Operating System or other system software components.

Other low-memory global variables contain information about the current application. For example, the `ApplZone` global variable contains the address of the first byte of the active application's partition. The `ApplLimit` 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.

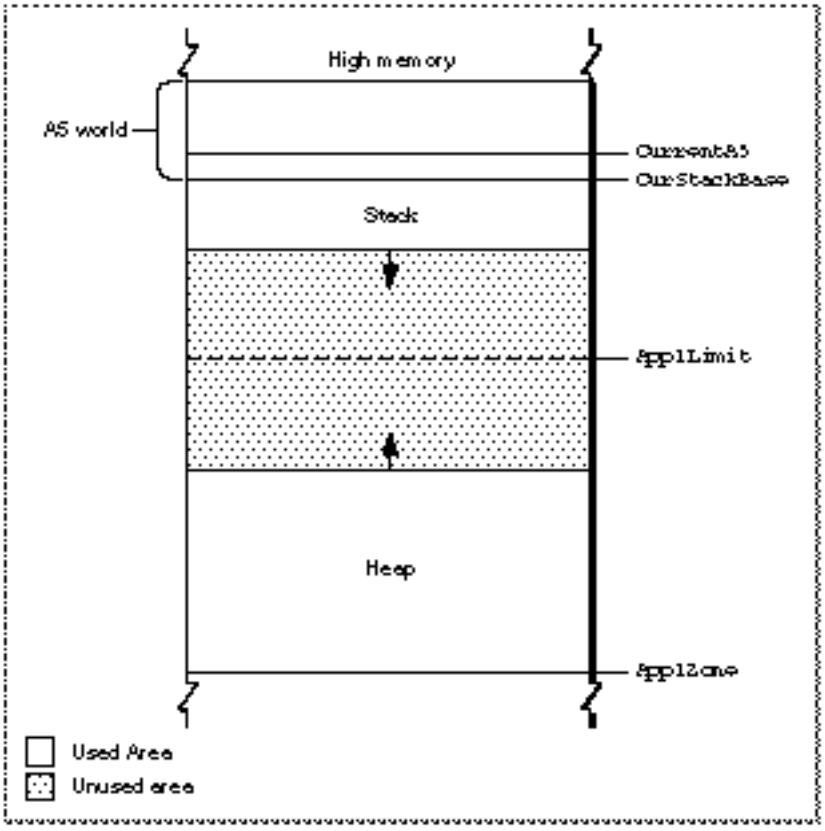
In general, it is best to avoid reading or writing low-memory system global variables. Most of these variables are undocumented, and the results of changing their values can be unpredictable. Usually, when the value of a low-memory global variable is likely to be useful to applications, the system software 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.

In rare instances, there is no routine that reads or writes the value of a documented global variable. In those cases, you might need to read or write that value directly. See the chapter "Memory Manager" in this book for instructions on reading and writing the values of low-memory global variables from a high-level language.

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.** That partition contains required segments of the application's code as well as other data associated with the application. Figure 1-2 illustrates the general organization of an application partition.

Figure 1-2 Organization of an application partition



Your application partition is divided into three major parts:

- the application stack
- the application heap
- the application global variables and A5 world

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.

As you can see in Figure 1-2, 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. See "Changing the Size of the Stack" on page 1-39 for instructions on how to change the size of your application stack.

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`. For more information on expanding the heap, see "Setting Up the Application Heap" beginning on page 1-38. ◆

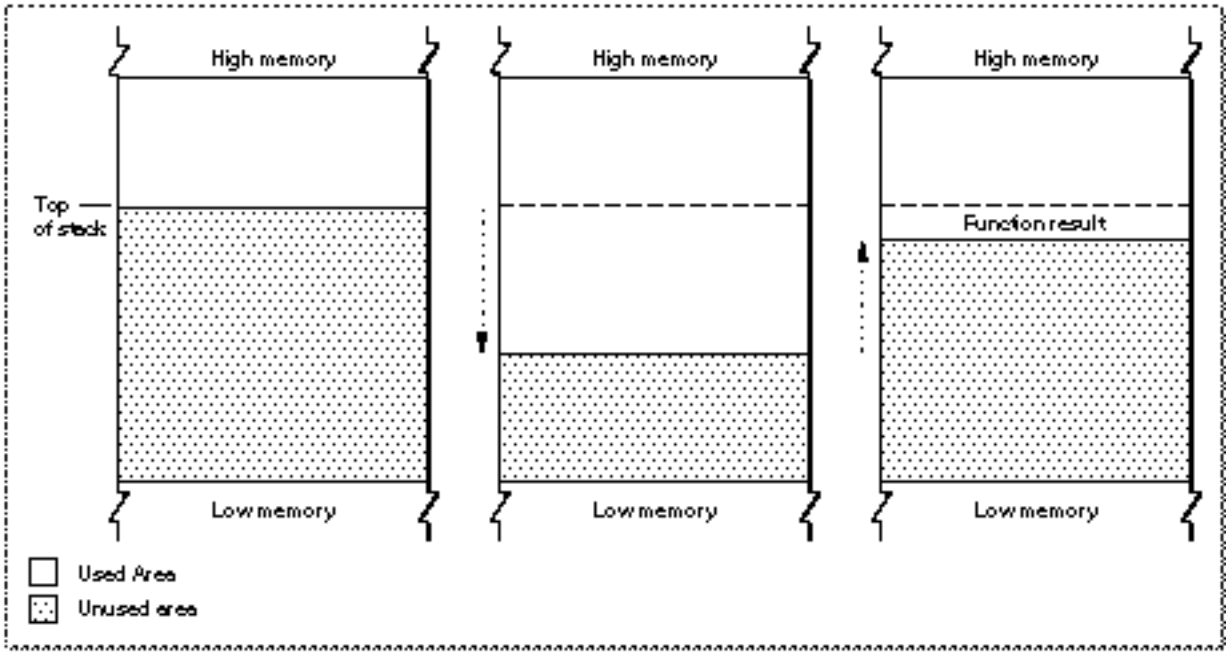
The Application Stack

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.

Because of its LIFO nature, the stack is especially useful for memory allocation connected with the execution of functions or procedures. 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. Figure 1-3 illustrates how the stack expands and shrinks during a function call. The leftmost diagram shows the stack just before the function is called. The middle diagram shows the stack expanded to hold the stack frame. Once the function is executed, the local variables and function parameters are popped off the stack. If the function is a Pascal function, all that remains is the previous stack with the function result on top.

Figure 1-3 The application stack



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

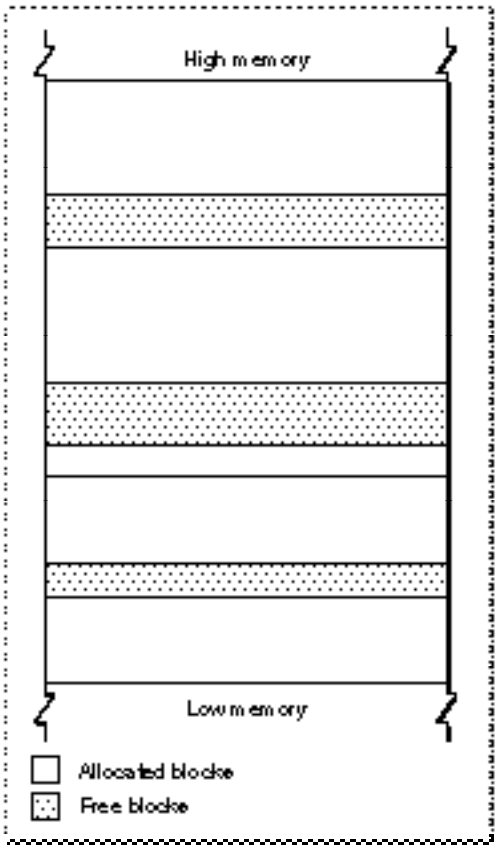
An **application heap** is the area of memory in your application partition in which space is dynamically allocated and released on demand. The heap begins at the low-memory

end of your application partition and extends upward in memory. 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.

You allocate space within your application's heap by making calls to the Memory Manager, either directly (for instance, using the `NewHandle` function) or indirectly (for instance, using a routine such as `NewWindow`, which calls Memory Manager routines). Space in the heap is allocated in **blocks,** which can be of any size needed for a particular object.

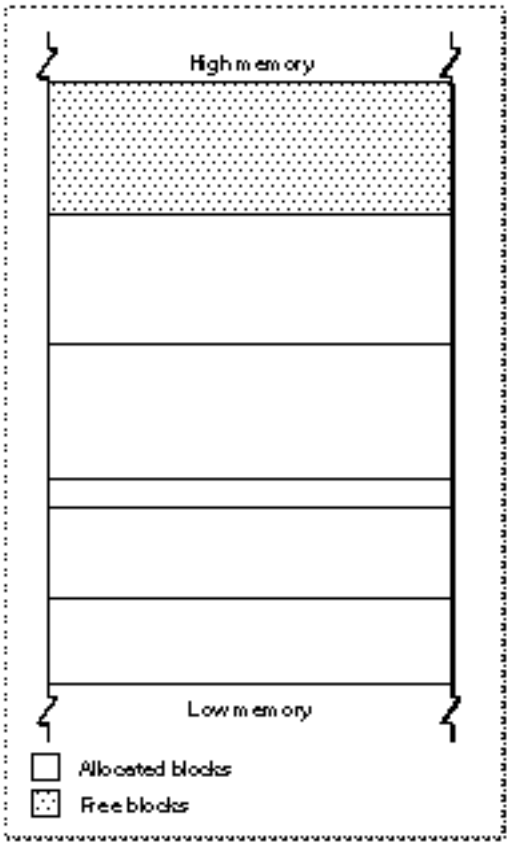
The Memory Manager does all the necessary housekeeping to keep track of blocks in the heap as they are allocated and released. Because these operations can occur in any order, the heap doesn't usually grow and shrink in an orderly way, as the stack does. Instead, after your application has been running for a while, the heap can tend to become fragmented into a patchwork of allocated and free blocks, as shown in Figure 1-4. This fragmentation is known as **heap fragmentation.**

Figure 1-4 A fragmented heap



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.** Figure 1-5 shows the results of compacting the fragmented heap shown in Figure 1-4.

Figure 1-5 A compacted heap



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. To minimize heap fragmentation, you should use nonrelocatable blocks sparingly, and you should lock relocatable blocks only when absolutely necessary. See "Relocatable and Nonrelocatable Blocks" starting on page 1-16 for a description of relocatable and nonrelocatable blocks, and "Heap Fragmentation" on page 1-24 for a description of how best to avoid fragmenting your heap.

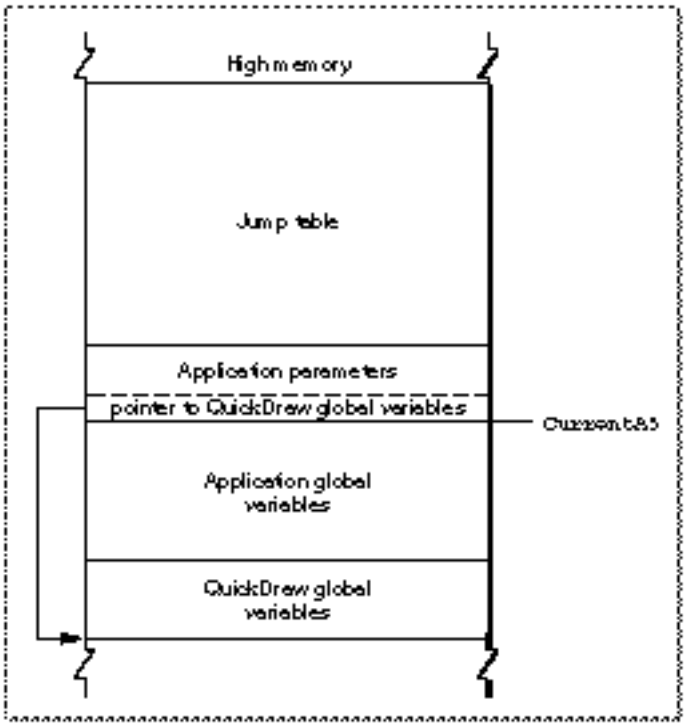
The Application Global Variables and A5 World

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:

- application global variables
- application QuickDraw global variables
- application parameters
- 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. Figure 1-6 shows the standard organization of the A5 world.

Figure 1-6 Organization of an application's A5 world



Note

An application's global variables may appear either above or below the QuickDraw global variables. The relative locations of these two items are determined by your development system's linker. In addition, part of the jump table might appear below the boundary pointed to by `CurrentA5`. ◆

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. For more information on jump tables, see the chapter "Segment Manager" in *Inside Macintosh: Processes*.

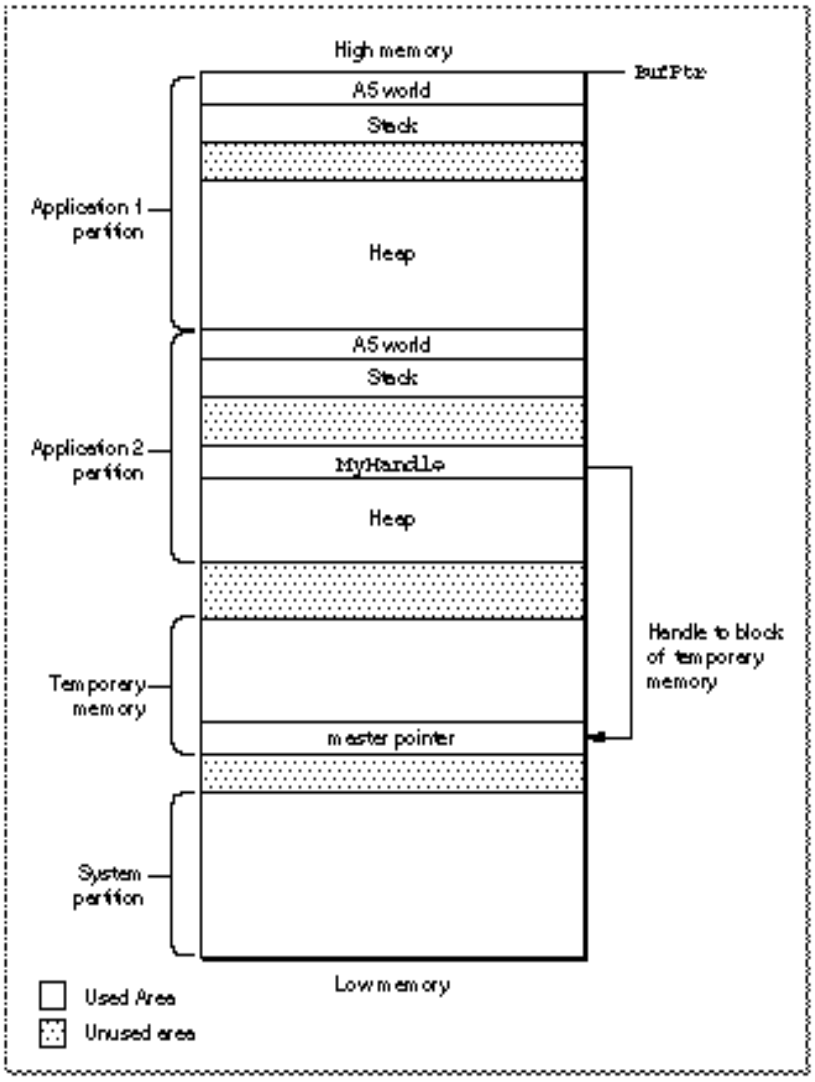
The **application parameters** are 32 bytes of memory located above the application global variables; they're reserved for use by the Operating System. The first long word of those parameters is a pointer to your application's QuickDraw global variables.

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. Figure 1-7 shows an application using some temporary memory.

Figure 1-7 Using temporary memory allocated from unused RAM



In Figure 1-7, Application 1 has almost exhausted its application heap. As a result, it has requested and received a large block of temporary memory, extending from the top of Application 2's partition to the top of the allocatable space. Application 1 can use the temporary memory in whatever manner it desires.

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.

Although using the smaller buffer might prolong the copying operation, the file is nonetheless copied.

One good reason for using temporary memory only occasionally is that you cannot assume that you will always receive the temporary memory you request. For example, in Figure 1-7, all the available memory is allocated to the two open applications; any further requests by either one for some temporary memory would fail. For complete details on using temporary memory, see the chapter "Memory Manager" in this book.

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. Because of virtual memory, a user can load more programs and data into the logical address space than would fit in the computer's physical RAM.

The Operating System extends the address space by using part of the available secondary storage (that is, part of a hard disk) to hold portions of applications and data that are not currently needed in RAM. When some of those portions of memory are needed, the Operating System swaps out unneeded parts of applications or data to the secondary storage, thereby making room for the parts that are needed.

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. For complete details on virtual memory, see the chapter "Virtual Memory Manager" later in this book.

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. In a 24-bit addressing scheme, the logical address space has a size of 16 MB. Because 8 MB of this total are reserved for I/O space, ROM, and slot space, the largest contiguous program address space is 8 MB. When 32-bit addressing is in operation, the maximum program address space is 1 GB.

The ability to operate with 32-bit addressing is available only on certain Macintosh models, namely those with systems that contain a 32-bit Memory Manager. (For compatibility reasons, these systems also contain a 24-bit Memory Manager.) In order for your application to work when the machine is using 32-bit addressing, it must be **32-bit clean,** that is, able to run in an environment where all 32 bits of a memory address are significant. Fortunately, writing applications that are 32-bit clean is relatively easy if you follow the guidelines in *Inside Macintosh*. In general, applications are not 32-bit clean because they manipulate flag bits in master pointers directly (for instance, to mark the associated memory blocks as locked or purgeable) instead of using Memory Manager

routines to achieve the desired result. See “Relocatable and Nonrelocatable Blocks” on page 1-16 for a description of master pointers.

▲ **WARNING**
You should never make assumptions about the contents of Memory Manager data structures, including master pointers and zone headers. These structures have changed in the past and they are likely to change again in the future. ▲

Occasionally, an application running when 24-bit addressing is enabled might need to modify memory addresses to make them compatible with the 24-bit Memory Manager. In addition, drivers or other code might need to use 32-bit addresses, even when running in 24-bit mode. See the descriptions of the routines `StripAddress` and `Translate24to32` in the chapter “Memory Management Utilities” for details.

Heap Management

Applications allocate and manipulate memory primarily in their application heap. As you have seen, space in the application heap is allocated and released on demand. When the blocks in your heap are free to move, the Memory Manager can often reorganize the heap to free space when necessary to fulfill a memory-allocation request. In some cases, however, blocks in your heap cannot move. In these cases, you need to pay close attention to memory allocation and management to avoid fragmenting your heap and running out of memory.

This section provides a general description of how to manage blocks of memory in your application heap. It describes

- relocatable and nonrelocatable blocks
- properties of relocatable blocks
- heap purging and compaction
- heap fragmentation
- dangling pointers
- low-memory conditions

For examples of specific techniques you can use to implement the strategies discussed in this section, see “Using Memory” beginning on page 1-38.

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).

The Memory Manager sometimes moves relocatable blocks during memory operations so that it can use the space in the heap optimally.

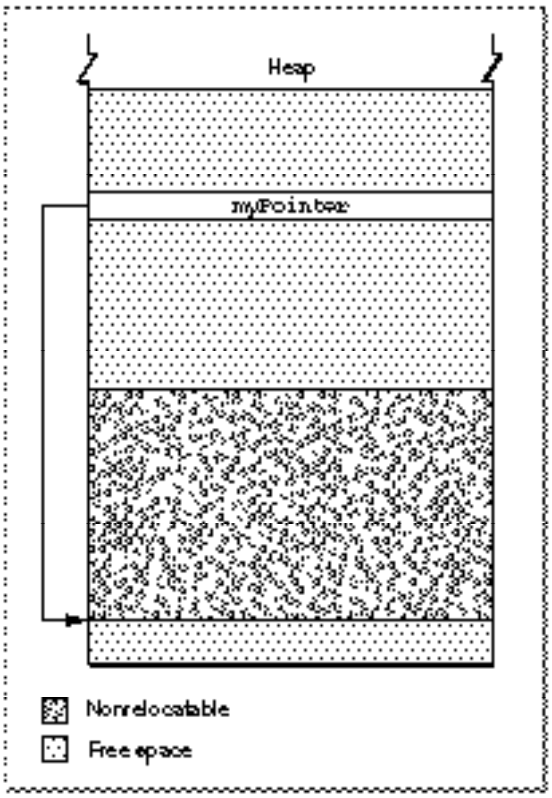
The Memory Manager provides data types that reference both relocatable and nonrelocatable blocks. It also provides routines that allow you to allocate and release blocks of both types.

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

```
TYPE
   SignedByte     = –128..127;
   Ptr            = ^SignedByte;
```

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, as illustrated in Figure 1-8. After you allocate a nonrelocatable block, you can make copies of the pointer variable. 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.

Figure 1-8 A pointer to a nonrelocatable block



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

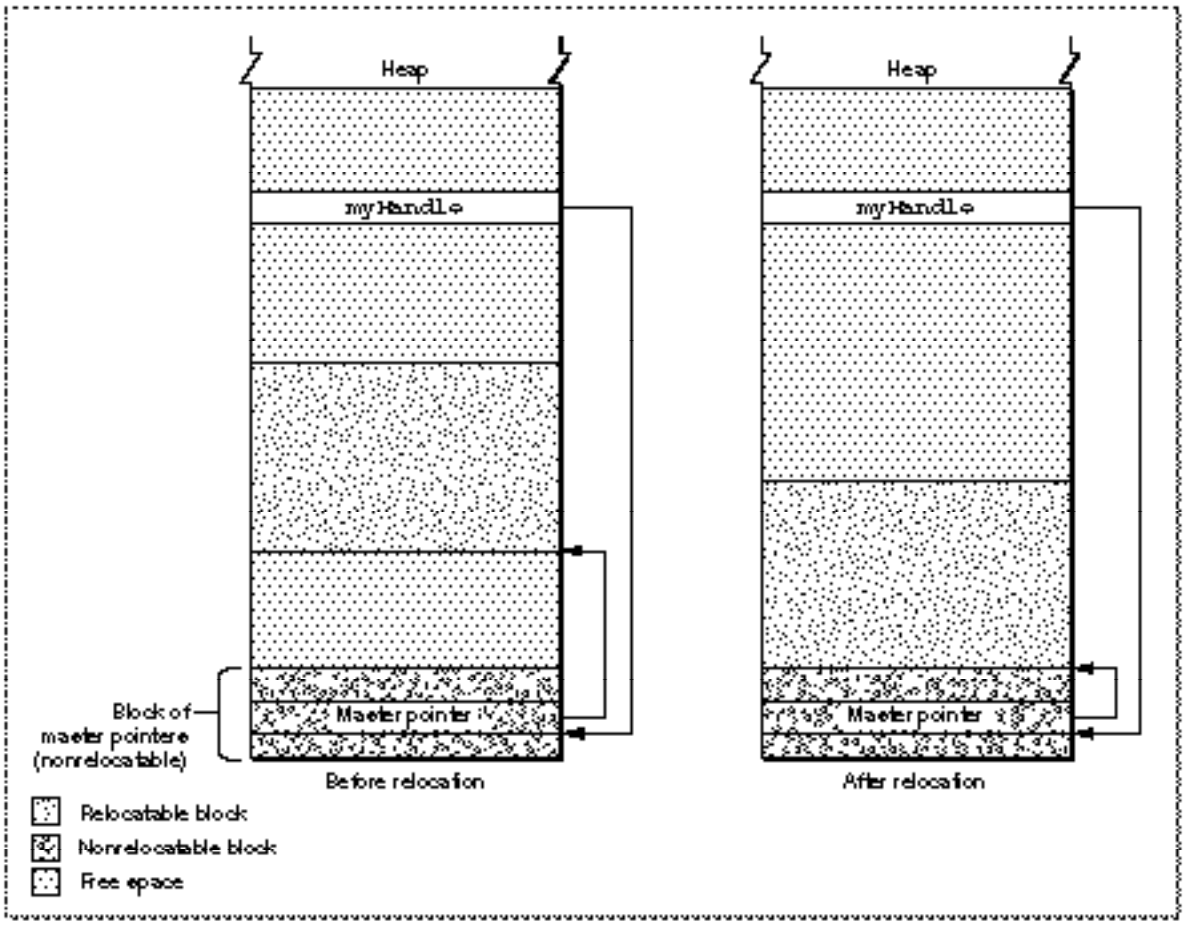
The Memory Manager allocates one master pointer block (containing 64 master pointers) for your application at launch time, and you can call the `MoreMasters` procedure to request that additional master pointer blocks be allocated. See "Setting Up the Application Heap" beginning on page 1-38 for instructions on allocating master pointer blocks. ◆

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.

```
TYPE
   Handle          = ^Ptr;
```

A handle contains the address of a master pointer. The left side of Figure 1-9 shows a handle to a relocatable block of memory located in the middle of the application heap. If necessary (perhaps to make room for another block of memory), the Memory Manager can move that block down in the heap, as shown in the right side of Figure 1-9.

Figure 1-9 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.

Whenever possible, you should allocate memory in relocatable blocks. This gives the Memory Manager the greatest freedom when rearranging the blocks in your application heap to create a new block of free memory. 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. You need to exercise care when calling Toolbox routines that allocate such blocks, lest your application heap become overly fragmented. See “Allocating Blocks of Memory” on page 1-44 for specific guidelines on allocating nonrelocatable blocks.

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. It groups these master pointers into nonrelocatable blocks. For large relocatable blocks, this extra space is negligible, but if you allocate many very small relocatable blocks, the cost can be considerable. For this reason, you should avoid allocating a very large number of handles to small blocks; instead, allocate a single large block and use it as an array to hold the data you need.

Properties of Relocatable Blocks

As you have seen, a heap block can be either relocatable or nonrelocatable. The designation of a block as relocatable or nonrelocatable is a permanent property of that block. If relocatable, a block can be either locked or unlocked; if it's unlocked, a block can be either purgeable or unpurgeable. These attributes of relocatable blocks can be set and changed as necessary. The following sections explain how to lock and unlock blocks, and how to mark them as purgeable or unpurgeable.

Locking and Unlocking Relocatable Blocks

Occasionally, you might need a relocatable block of memory to stay in one place. 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. See "Dangling Pointers" on page 1-29 for a complete discussion of how to avoid dangling pointers. ◆

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. See "Loading Code Segments" on page 1-31 for details.

Purging and Reallocating Relocatable Blocks

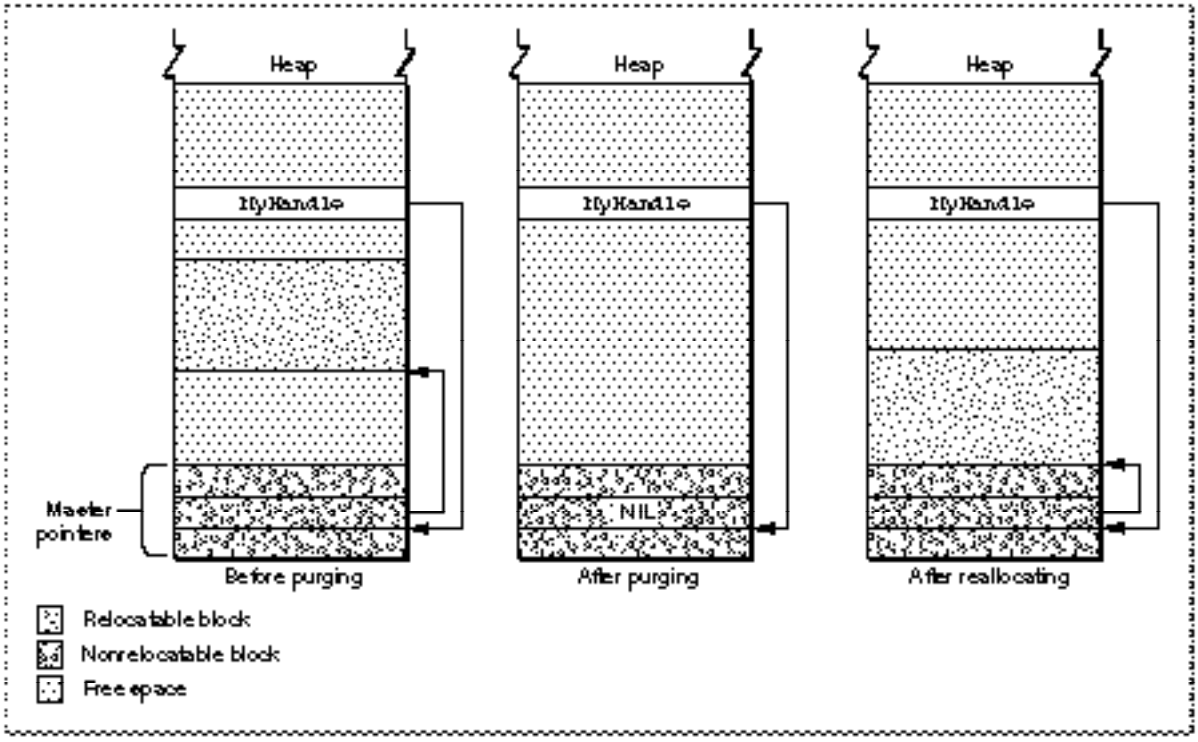
One advantage of relocatable blocks is that you can use them to store information that you would like to keep in memory to make your application more efficient, but that you don't really need if available memory space becomes low. For example, your application might, at the beginning of its execution, load user preferences from a preferences file into a relocatable block. As long as the block remains in memory, your application can access information from the preferences file without actually reopening the file. However, reopening the file probably wouldn't take enough time to justify keeping the block in memory if memory space were scarce.

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).

Figure 1-10 illustrates the purging and reallocating of a relocatable block. When the block is purged, its master pointer is set to `NIL`. When it is reallocated, the handle correctly references a new block, but that block's contents are initially undefined.

Figure 1-10 Purging and reallocating a relocatable block

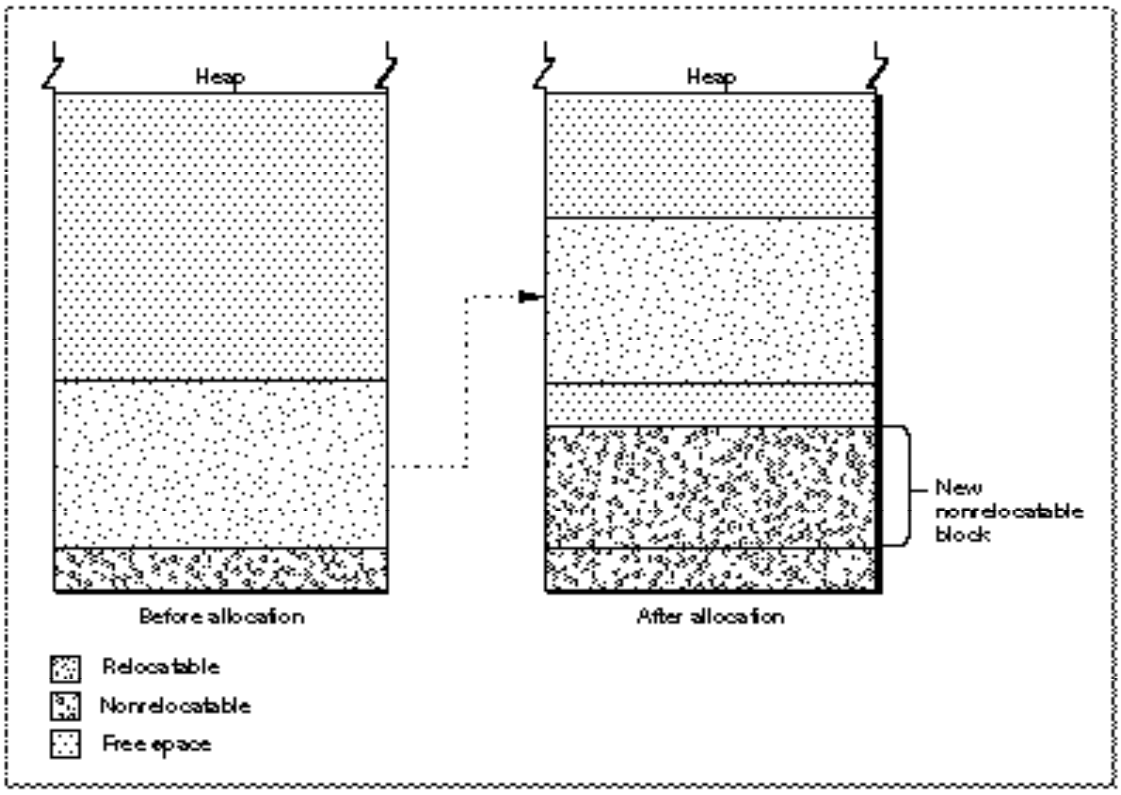


Memory Reservation

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.

Figure 1-11 illustrates how the Memory Manager allocates nonrelocatable blocks. Although it could place a block of the requested size at the top of the heap, it instead reserves space for the block as close to the bottom of the heap as possible and then puts the block into that reserved space. During this process, the Memory Manager might even move a relocatable block over a nonrelocatable block to make room for another nonrelocatable block.

Figure 1-11 Allocating a nonrelocatable block



When allocating a new relocatable block, you can, if you want, manually reserve space for the block by calling the `ReserveMem` procedure. If you do not, the Memory Manager looks for space big enough for the block as low in the heap as possible, but it does not create space near the bottom of the heap for the block if there is already enough space higher in the heap.

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.

In a purge of the heap, the Memory Manager sequentially purges unlocked, purgeable relocatable blocks until it has freed enough memory or until it has purged all such blocks. It purges a block by deallocating it and setting its master pointer to `NIL`.

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.

Note

In general, you should let the Memory Manager purge and compact your heap, instead of performing these operations yourself. ◆

Heap Fragmentation

Heap fragmentation can slow your application by forcing the Memory Manager to compact or purge your heap to satisfy a memory-allocation request. In the worst cases, when your heap is severely fragmented by locked or nonrelocatable blocks, it might be impossible for the Memory Manager to find the requested amount of contiguous free space, even though that much space is actually free in your heap. This can have disastrous consequences for your application. For example, if the Memory Manager cannot find enough room to load a required code segment, your application will crash.

Obviously, it is best to minimize the amount of fragmentation that occurs in your application heap. It might be tempting to think that because the Memory Manager controls the movement of blocks in the heap, there is little that you can do to prevent heap fragmentation. In reality, however, fragmentation does not strike your application's heap by chance. Once you understand the major causes of heap fragmentation, you can follow a few simple rules to minimize it.

The primary causes of heap fragmentation are indiscriminate use of nonrelocatable blocks and indiscriminate locking of relocatable blocks. Each of these creates immovable blocks in your heap, thus creating "roadblocks" for the Memory Manager when it rearranges the heap to maximize the amount of contiguous free space. You can significantly reduce heap fragmentation simply by exercising care when you allocate nonrelocatable blocks and when you lock relocatable blocks.

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.

Whenever you dispose of a nonrelocatable block that you have allocated, you create small gaps, unless the next nonrelocatable block you allocate happens to be the same size as the disposed block. These small gaps can lead to heavy fragmentation over the course of your application's execution. Thus, you should try to avoid disposing of and then reallocating nonrelocatable blocks during program execution.

Reserving Memory

Another cause of heap fragmentation ironically occurs because of a limitation of memory reservation, a process designed to prevent it. Memory reservation never makes fragmentation worse than it would be if there were no memory reservation. Ordinarily, memory reservation ensures that allocating nonrelocatable blocks in the middle of your application's execution causes no problems. Occasionally, however, memory reservation can cause fragmentation, either when it succeeds but leaves small gaps in the reserved space, or when it fails and causes a nonrelocatable block to be allocated in the middle of the heap.

The Memory Manager uses memory reservation to create space for nonrelocatable blocks as low as possible in the heap. (You can also manually reserve memory for relocatable blocks, but you rarely need to do so.) However, when the Memory Manager moves a block up during memory reservation, that block cannot overlap its previous location. As a result, the Memory Manager might need to move the relocatable block up more than is necessary to contain the new nonrelocatable block, thereby creating a gap between the top of the new block and the bottom of the relocated block. (See Figure 1-11 on page 1-23.)

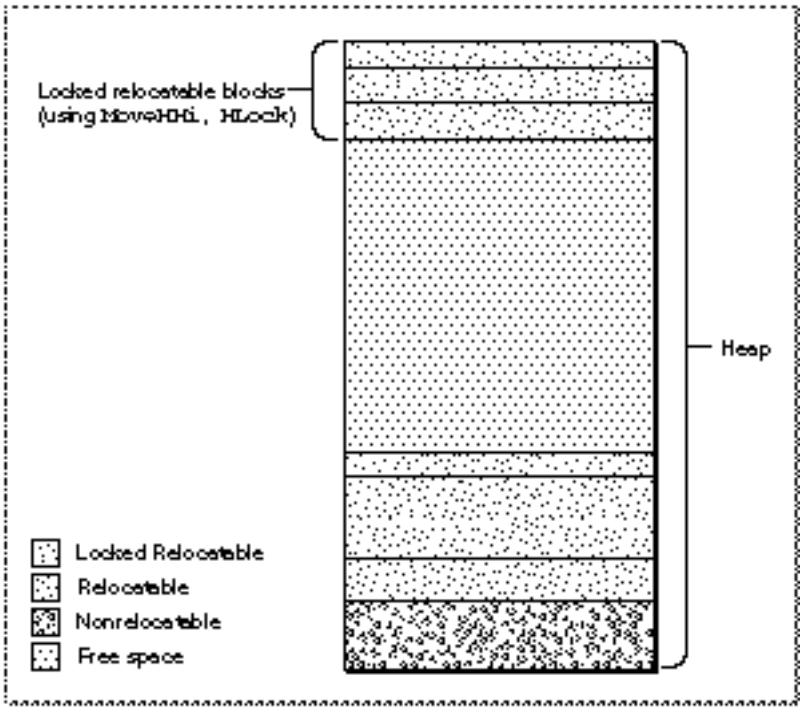
Memory reservation can also fragment the heap if there is not enough space in the heap to move the relocatable block up. In this case, the Memory Manager allocates the new nonrelocatable block above the relocatable block. The relocatable block cannot then move over the nonrelocatable block, except during the times described previously.

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.

An alternative partial solution is to move relocatable blocks to the top of the heap before locking them. The `MoveHHi` procedure allows you to move a relocatable block upward until it reaches the top of the heap, a nonrelocatable block, or a locked relocatable block. This has the effect of partitioning the heap into four areas, as illustrated in Figure 1-12. At the bottom of the heap are the nonrelocatable blocks. Above those blocks are the unlocked relocatable blocks. At the top of the heap are locked relocatable blocks. Between the locked relocatable blocks and the unlocked relocatable blocks is an area of free space. The principal idea behind moving relocatable blocks to the top of the heap and locking them there is to keep the contiguous free space as large as possible.

Figure 1-12 An effectively partitioned heap



Using `MoveHHi` is, however, not always a perfect solution to handling relocatable blocks that need to be locked. The `MoveHHi` procedure moves a block upward only until it reaches either a nonrelocatable block or a locked relocatable block. Unlike `NewPtr` and `ReserveMem`, `MoveHHi` does not currently move a relocatable block around one that is not relocatable.

Even if `MoveHHi` succeeds in moving a block to the top area of the heap, unlocking or deleting locked blocks can cause fragmentation if you don't unlock or delete those blocks beginning with the lowest locked block. A relocatable block that is locked at the top area of the heap for a long period of time could trap other relocatable blocks that were locked for short periods of time but then unlocked.

This suggests that you need to treat relocatable blocks locked for a long period of time differently from those locked for a short period of time. If you plan to lock a relocatable block for a long period of time, you should reserve memory for it at the bottom of the heap before allocating it, then lock it for the duration of your application's execution (or as long as the block remains allocated). Do not reserve memory for relocatable blocks you plan to allocate for only short periods of time. Instead, move them to the top of the heap (by calling `MoveHHi`) and then lock them.

Note
You should call `MoveHHi` only on blocks located in your application heap. Don't call `MoveHHi` on relocatable blocks in the system heap. Desk accessories should not call `MoveHHi`. ◆

In practice, you apply the same rules to relocatable blocks that you reserve space for and leave permanently locked as you apply to nonrelocatable blocks: Try not to allocate such blocks in the middle of your application's execution, and don't dispose of and reallocate such blocks in the middle of your application's execution.

After you lock relocatable blocks temporarily, you don't need to move them manually back into the middle area when you unlock them. Whenever the Memory Manager compacts the heap or moves another relocatable block to the top heap area, it brings all unlocked relocatable blocks at the bottom of that partition back into the middle area. When moving a block to the top area, be sure to call `MoveHHi` on the block and then lock the block, in that order.

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.

There is, however, an exception to the rule that you should not allocate nonrelocatable blocks in the middle of your application's execution. Sometimes you need to allocate a nonrelocatable block only temporarily. If between the times that you allocate and dispose of a nonrelocatable block, you allocate no additional nonrelocatable blocks and do not attempt to compact the heap, then you have done no harm. The temporary block cannot create a new gap because the Memory Manager places no other block over the temporary block.

Summary of Preventing Fragmentation

Avoiding heap fragmentation is not difficult. It simply requires that you follow a few rules as closely as possible. Remember that allocation of even a small nonrelocatable block in the middle of your heap can ruin a scheme to prevent fragmentation of the heap, because the Memory Manager does not move relocatable blocks around nonrelocatable blocks when you call `MoveHHi` or when it attempts to compact the heap.

If you adhere to the following rules, you are likely to avoid significant heap fragmentation:

- At the beginning of your application's execution, call the `MaxApplZone` procedure once and the `MoreMasters` procedure enough times so that the Memory Manager never needs to call it for you.
- Try to anticipate the maximum number of nonrelocatable blocks you will need and allocate them at the beginning of your application's execution.
- Avoid disposing of and then reallocating nonrelocatable blocks during your application's execution.
- When allocating relocatable blocks that you need to lock for long periods of time, use the `ReserveMem` procedure to reserve memory for them as close to the bottom of the heap as possible, and lock the blocks immediately after allocating them.
- If you plan to lock a relocatable block for a short period of time and allocate nonrelocatable blocks while it is locked, use the `MoveHHi` procedure to move the block to the top of the heap and then lock it. When the block no longer needs to be locked, unlock it.
- Remember that you need to lock a relocatable block only if you call a routine that could move or purge memory and you then use a dereferenced handle to the relocatable block, or if you want to use a dereferenced handle to the relocatable block at interrupt time.

Perhaps the most difficult restriction is to avoid disposing of and then reallocating nonrelocatable blocks in the middle of your application's execution. Some Toolbox routines require you to use nonrelocatable blocks, and it is not always easy to anticipate how many such blocks you will need. If you must allocate and dispose of blocks in the middle of your program's execution, you might want to place used blocks into a linked list of free blocks instead of disposing of them. If you know how many nonrelocatable blocks of a certain size your application is likely to need, you can add that many to the beginning of the list at the beginning of your application's execution. If you need a nonrelocatable block later, you can check the linked list for a block of the exact size instead of simply calling the `NewPtr` function.

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. Unfortunately, it is often easy during debugging to overlook situations that could leave pointers dangling, because pointers dangle only if the relocatable blocks that they reference actually move. Routines that can move or purge memory do not necessarily do so unless memory space is tight. Thus, if you improperly dereference a handle in a section of code, that code might still work properly most of the time. If, however, a dangling pointer does cause errors, they can be very difficult to trace.

This section describes a number of situations that can cause dangling pointers and suggests some ways to avoid them.

Compiler Dereferencing

Some of the most difficult dangling pointers to isolate are not caused by any explicit dereferencing on your part, but by implicit dereferencing on the part of the compiler. For example, suppose you use a handle called `myHandle` to access the fields of a record in a relocatable block. You might use Pascal's `WITH` statement to do so, as follows:

```
WITH myHandle^^ DO
   BEGIN
      ...
   END;
```

A compiler is likely to dereference `myHandle` so that it can access the fields of the record without double indirection. However, if the code between the `BEGIN` and `END` statements causes the Memory Manager to move or purge memory, you are likely to end up with a dangling pointer.

The easiest way to prevent dangling pointers is simply to lock the relocatable block whose data you want to read or write. Because the block is locked and cannot move,

the master pointer is guaranteed always to point to the beginning of the block's data. Listing 1-1 illustrates one way to avoid dangling pointers by locking a relocatable block.

Listing 1-1 Locking a block to avoid dangling pointers

```
VAR
   origState:  SignedByte;    {original attributes of handle}

origState := HGetState(Handle(myData));{get handle attributes}
MoveHHi(Handle(myData));               {move the handle high}
HLock(Handle(myData));                 {lock the handle}
WITH myData^^ DO                       {fill in window data}
   BEGIN
      editRec := TENew(gDestRect, gViewRect);
      vScroll := GetNewControl(rVScroll, myWindow);
      hScroll := GetNewControl(rHScroll, myWindow);
      fileRefNum := 0;
      windowDirty := FALSE;
   END;
HSetState(origState);                  {reset handle attributes}
```

The handle `myData` needs to be locked before the `WITH` statement because the functions `TENew` and `GetNewControl` allocate memory and hence might move the block whose handle is `myData`.

You should be careful to lock blocks only when necessary, because locked relocatable blocks can increase heap fragmentation and slow down your application unnecessarily. You should lock a handle only if you dereference it, directly or indirectly, and then use a copy of the original master pointer after calling a routine that could move or purge memory. When you no longer need to reference the block with the master pointer, you should unlock the handle. In Listing 1-1, the handle `myData` is never explicitly unlocked. Instead, the original attributes of the handle are saved by calling `HGetState` and later are restored by calling `HSetState`. This strategy is preferable to just calling `HLock` and `HUnlock`.

A compiler can generate hidden dereferencing, and hence potential dangling pointers, in other ways, for instance, by assigning the result of a function that might move or purge blocks to a field in a record referenced by a handle. Such problems are particularly common in code that manipulates linked data structures. For example, you might use this code to allocate a new element of a linked list:

```
myHandle^^.nextHandle := NewHandle(sizeof(myLinkedElement));
```

This can cause problems because your compiler could dereference `myHandle` before calling `NewHandle`. Therefore, you should either lock `myHandle` before performing the allocation, or use a temporary variable to allocate the new handle, as in the following code:

```
tempHandle := NewHandle(sizeof(myLinkedElement));
myHandle^^.nextHandle := tempHandle;
```

Passing fields of records as arguments to routines that might move or purge memory can cause similar problems, if the records are in relocatable blocks referred to with handles. Problems arise only when you pass a field by reference rather than by value. Pascal conventions call for all arguments larger than 4 bytes to be passed by reference. In Pascal, a variable is also passed by reference when the routine called requests a variable parameter. Both of the following lines of code could leave a pointer dangling:

```
TEUpdate(hTE^^.viewRect, hTE);
InvalRect(theControl^^.contrlRect);
```

These problems occur because a compiler may dereference a handle before calling the routine to which you pass the handle. Then, that routine may move memory before it uses the dereferenced handle, which might then be invalid. As before, you can solve these problems by locking the handles or using temporary variables.

Loading Code Segments

If you call an application-defined routine located in a code segment that is not currently in RAM, the Segment Manager might need to move memory when loading that code segment, thus jeopardizing any dereferenced handles you might be using. For example, suppose you call an application-defined procedure `ManipulateData`, which manipulates some data at an address passed to it in a variable parameter.

```
PROCEDURE MyRoutine;
BEGIN
   ...
   ManipulateData(myHandle^);
   ...
END;
```

You can create a dangling pointer if `ManipulateData` and `MyRoutine` are in different segments, and the segment containing `ManipulateData` is not loaded when `MyRoutine` is executed. You can do this because you've passed a dereferenced copy of `myHandle` as an argument to `ManipulateData`. If the Segment Manager must allocate a new relocatable block for the segment containing `ManipulateData`, it might move `myHandle` to do so. If so, the dereferenced handle would dangle. A similar problem can occur if you assign the result of a function in a nonresident code segment to a field in a record referred to by a handle.

You need to be careful even when passing a field in a record referenced by a handle to a routine in the same code segment as the caller, or when assigning the result of a function in the same code segment to such a field. If that routine could call a Toolbox routine that might move or purge memory, or call a routine in a different, nonresident code segment, then you could indirectly cause a pointer to dangle.

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. You must avoid empty, disposed, or fake handles as carefully as dangling pointers. Fortunately, it is generally easier to detect, and thus to avoid, invalid 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.)

If you accidentally use a handle to a block you have already disposed of, you can obtain unexpected results. In the best cases, your application will crash. In the worst cases, you will get garbled data. It might, however, be difficult to trace the cause of the garbled data, because your application can continue to run for quite a while before the problem begins to manifest itself.

You can avoid these problems quite easily by assigning the value `NIL` to the handle variable after you dispose of its associated block. By doing so, you indicate that the handle does not point anywhere in particular. If you subsequently attempt to operate on such a block, the Memory Manager will probably generate a `nilHandleErr` result code. If you want to make certain that a handle is not disposed of before operating on a relocatable block, you can test whether the value of the handle is `NIL`, as follows:

```
IF myHandle <> NIL THEN
   ...;           {handle is valid, so we can operate on it here}
```

Note

This test is useful only if you manually assign the value `NIL` to all disposed handles. The Memory Manager does not do that automatically. ◆

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. ◆

Once again, however, inadvertently using an empty handle can give unexpected results or lead to a system crash. In the Macintosh Operating System, NIL technically refers to memory location 0. But this memory location holds a value. If you doubly dereference an empty handle, you reference whatever data is found at that location, and you could obtain unexpected results that are difficult to trace.

You can check for empty handles much as you check for disposed handles. Assuming you set handles to NIL when you dispose of them, you can use the following code to determine whether a handle both points to a valid master pointer and references a nonempty relocatable block:

```
IF myHandle <> NIL THEN
   IF myHandle^ <> NIL THEN
      ...          {we can operate on the relocatable block here}
```

Note that because Pascal evaluates expressions completely, you need two IF-THEN statements rather than one compound statement in case the value of the handle itself is NIL. Most compilers, however, allow you to use "short-circuit" Boolean operators to minimize the evaluation of expressions. For example, if your compiler uses the operator & as a short-circuit operator for AND, you could rewrite the preceding code like this:

```
IF (myHandle <> NIL) & (myHandle^ <> NIL) THEN
   ...             {we can operate on the relocatable block here}
```

In this case, the second expression is evaluated only if the first expression evaluates to TRUE.

Note

The availability and syntax of short-circuit Boolean operators are compiler dependent. Check the documentation for your development system to see whether you can use such operators. ◆

It is useful during debugging to set memory location 0 to an odd number, such as $50FFC001. This causes the Operating System to crash immediately if you attempt to dereference an empty handle. This is useful, because you can immediately fix problems that might otherwise require extensive debugging.

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 Listing 1-2.

Listing 1-2 Creating a fake handle

```
FUNCTION MakeFakeHandle: Handle;    {DON'T USE THIS FUNCTION!}
CONST
   kMemoryLoc = $100;               {a random memory location}
VAR
   myHandle:   Handle;
   myPointer:  Ptr;
BEGIN
   myPointer := Ptr(kMemoryLoc);    {the address of some memory}
   myHandle := @myPointer;          {the address of a pointer}
   MakeFakeHandle := myHandle;
END;
```

▲ **WARNING**
The technique for creating a fake handle shown in Listing 1-2 is included for illustrative purposes only. Your application should never create fake handles. ▲

Remember that a real handle contains the address of a master pointer. The fake handle manufactured by the function `MakeFakeHandle` in Listing 1-2 contains an address that may or may not be the address of a master pointer. If it isn't the address of a master pointer, then you virtually guarantee chaotic results if you pass the fake handle to a system software routine that expects a real handle.

For example, suppose you pass a fake handle to the `MoveHHi` procedure. After allocating a new relocatable block high in the heap, `MoveHHi` is likely to copy the data from the original block to the new block by dereferencing the handle and using, supposedly, a master pointer. Because, however, the value of a fake handle probably isn't the address of a master pointer, `MoveHHi` copies invalid data. (Actually, it's unlikely that `MoveHHi` would ever get that far; probably it would run into problems when attempting to determine the size of the original block from the block header.)

Not all fake handles are as easy to spot as those created by the `MakeFakeHandle` function defined in Listing 1-2. You might, for instance, attempt to copy the data in an existing record (`myRecord`) into a new handle, as follows:

```
myHandle := NewHandle(SizeOf(myRecord));  {create a new handle}
myHandle^ := @myRecord;                   {DON'T DO THIS!}
```

The second line of code does *not* make `myHandle` a handle to the beginning of the `myRecord` record. Instead, it overwrites the master pointer with the address of that record, making `myHandle` a fake handle.

▲ **WARNING**

Never assign a value directly to a master pointer. ▲

A correct way to create a new handle to some existing data is to make a copy of the data using the `PtrToHand` function, as follows:

```
myErr := PtrToHand(@myRecord, myHandle, SizeOf(myRecord));
```

The Memory Manager provides a set of pointer- and handle-manipulation routines that can help you avoid creating fake handles. See the chapter "Memory Manager" in this book for details on those routines.

Low-Memory Conditions

It is particularly important to make sure that the amount of free space in your application heap never gets too low. For example, you should never deplete the available heap memory to the point that it becomes impossible to load required code segments. As you have seen, your application will crash if the Segment Manager is called to load a required code segment and there is not enough contiguous free memory to allocate a block of the appropriate size.

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. By making a block purgeable, you give the Memory Manager the freedom to release that space if heap memory becomes low. You can also help maximize the available heap memory by intelligently segmenting your application's executable code and by periodically unloading any unneeded segments. The standard way to do this is to unload every nonessential segment at the end of your application's main event loop. (See the chapter "Segment Manager" in *Inside Macintosh: Processes* for a complete discussion of code-segmentation techniques.)

Memory Cushions

These two measures—making blocks purgeable and unloading segments—help you only by releasing blocks that have already been allocated. It is even more important to make sure, *before* you attempt to allocate memory directly, that you don't deplete the available heap memory. 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.** You should not simply inspect the handle or pointer returned to you and make sure that its value isn't `NIL`, because you might have succeeded in allocating the space you requested but left the amount of free space dangerously low.

You also need to make sure that indirect memory allocation doesn't cut into the memory cushion. When, for example, you call `GetNewDialog`, the Dialog Manager might need to allocate space for a dialog record; it also needs to allocate heap space for the dialog item list and any other custom items in the dialog. Before calling `GetNewDialog`, therefore, you need to make sure that the amount of space left free after the call is greater than your memory cushion.

The execution of some system software routines requires significant amounts of memory in your heap. For example, some QuickDraw operations on regions can temporarily allocate fairly large amounts of space in your heap. Some of these system software routines, however, do little or no checking to see that your heap contains the required amount of free space. They either assume that they will get whatever memory they need or they simply issue a system error when they don't get the needed memory. In either case, the result is usually a system crash.

You can avoid these problems by making sure that there is always enough space in your heap to handle these hidden memory allocations. Experience has shown that 40 KB is a reasonably safe size for this memory cushion. If you can consistently maintain that amount of space free in your heap, you can be reasonably certain that system software routines will get the memory they need to operate. You also generally need a larger cushion (about 70 KB) when printing.

Memory Reserves

Unfortunately, there are times when you might need to use some of the memory in the cushion yourself. It is better, for instance, to dip into the memory cushion, if necessary, to save a user's document than to reject the request to save the document. Some actions your application performs should not be rejectable simply because they require it to reduce the amount of free space below a desired minimum.

Instead of relying on just the free memory of a memory cushion, you can allocate a **memory reserve,** some additional emergency storage that you release when free memory becomes low. The important difference between this memory reserve and the memory cushion is that the memory reserve is a block of allocated memory, which you release whenever you detect that essential tasks have dipped into the memory cushion.

That emergency memory reserve might provide enough memory to compensate for any essential tasks that you fail to anticipate. Because you allow essential tasks to dip into the memory cushion, the release itself of the memory reserve should not be a cause for alarm. Using this scheme, your application releases the memory reserve as a precautionary measure during ordinary operation. Ideally, however, the application should never actually deplete the memory cushion and use the memory reserve.

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 unpurgeable 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.

Using Memory

This section describes how you can use the Memory Manager to perform the most typical memory management tasks. In particular, this section shows how you can

- set up your application heap at application launch time
- determine how much free space is available in your application heap
- allocate and release blocks of memory in your heap
- define and install a grow-zone function

The techniques described in this section are designed to minimize fragmentation of your application heap and to ensure that your application always has sufficient memory to complete any essential operations. Many of these techniques incorporate the heap memory cushion and emergency memory reserve discussed in "Low-Memory Conditions," beginning on page 1-36.

Note

This section describes relatively simple memory-management techniques. Depending on the requirements of your application, you might want to manage your heap memory differently. ◆

Setting Up the Application Heap

When the Process Manager launches your application, it calls the Memory Manager to create and initialize a memory partition for your application. The Process Manager then

loads code segments into memory and sets up the stack, heap, and A5 world (including the jump table) for your application.

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.

Some applications, however, rely heavily on recursive programming techniques, in which one routine repeatedly calls itself or a small group of routines repeatedly call each other. In these applications, even routines with just a few local variables can cause stack overflow, because each time a routine calls itself, a new copy of that routine's parameters and variables is appended to the stack. The problem can become particularly acute if one or more of the local variables is a string, which can require up to 256 bytes of stack space.

You can help prevent your application from crashing because of insufficient stack space by expanding the size of your stack. If your application does not depend on recursion, you should do this only if you encounter system error 28 during testing. If your application does depend on recursion, you might consider expanding the stack so that your application can perform deeply nested recursive computations. In addition, some object-oriented languages (for example, C++) allocate space for objects on the stack. If you are using one of these languages, you might need to expand your stack.

Note

If you are programming in LISP or another language that depends extensively on recursion, your development system might allocate memory for local variables in the heap rather than on the stack. If so, expanding the size of the stack is not helpful. Consult your development system's documentation for details on how it allocates memory. ◆

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. (The size of the stack for a faceless background process is always 8 KB, whether Color QuickDraw is present or not.) You should never decrease the size of the stack, because future versions of system software might increase the default amount of space allocated for the stack. For the same reason, you should not set the stack to a predetermined absolute size or calculate a new absolute size for the stack based on the microprocessor's type. If you must modify the size of the stack, you should increase the stack size only by some relative amount that is sufficient to meet the increased stack requirements of your application. There is no maximum size to which the stack can grow.

Listing 1-3 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 1-3 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 (as described in the next section). If you call `IncreaseStackSize` after you call `MaxApplZone`, it has no effect, because the `SetApplLimit` procedure cannot change the `ApplLimit` global variable to a value lower than the current top of the heap.

Note

Some compilers add to the beginning of your application some default initialization code that automatically calls `MaxApplZone`. You might need to specify a compiler directive that turns off such default initialization if you want to increase the size of the stack. Consult your development system's documentation for details. ◆

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.

Another advantage to calling `MaxApplZone` is that doing so is likely to reduce the number of relocatable blocks that are purged by the Memory Manager. The Memory Manager expands your heap to fulfill a memory request only after it has exhausted other methods of obtaining the required amount of space, including compacting the heap and purging blocks marked as purgeable. By expanding the heap to its limit, you can prevent the Memory Manager from purging blocks that it otherwise would purge. This, together with the fact that your heap is expanded only once, can make memory allocation significantly faster.

Note

As indicated in the previous section, you should call `MaxApplZone` only after you have expanded the stack, if necessary. ◆

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.

If, when you allocate a relocatable block, there are no unused master pointers in your application heap, the Memory Manager automatically allocates a new block of master pointers. For several reasons, however, you should try to prevent the Memory Manager from calling `MoreMasters` for you. First, `MoreMasters` executes more slowly if it has to move relocatable blocks up in the heap to make room for the new nonrelocatable block of master pointers. When your application first starts running, there are no such blocks that might have to be moved. Second, the new nonrelocatable block of master pointers is likely to fragment your application heap. At any time the Memory Manager is forced to call `MoreMasters` for you, there are already at least 64 relocatable blocks allocated in your heap. Unless all or most of those blocks are locked high in the heap (an unlikely situation), the new nonrelocatable block of master pointers might be allocated above existing relocatable blocks. This increases heap fragmentation.

To prevent this fragmentation, you should call `MoreMasters` at the beginning of your application enough times to ensure that the Memory Manager never needs to call it for you. For example, if your application never allocates more than 300 relocatable blocks in its heap, then five calls to the `MoreMasters` should be enough. It's better to call `MoreMasters` too many times than too few, so if your application usually allocates about 100 relocatable blocks but sometimes might allocate 1000 in a particularly busy session, you should call `MoreMasters` enough times at the beginning of the program to cover the larger figure.

You can determine empirically how many times to call `MoreMasters` by using a low-level debugger. First, remove all the calls to `MoreMasters` from your code and then give your application a rigorous workout, opening and closing windows, dialog boxes, and desk accessories as much as any user would. Then, find out from your debugger how many times the system called `MoreMasters`. To do so, count the nonrelocatable

blocks of size $100 bytes (decimal 256, or 64 × 4). Because of Memory Manager size corrections, you should also count any nonrelocatable blocks of size $108, $10C, or $110 bytes. (You should also check to make sure that your application doesn't allocate other nonrelocatable blocks of those sizes. If it does, subtract the number it allocates from the total.) Finally, call `MoreMasters` at least that many times at the beginning of your application.

Listing 1-4 illustrates a typical sequence of steps to configure your application heap and stack. The `DoSetUpHeap` procedure defined there increases the size of the stack by 32 KB, expands the application heap to its new limit, and allocates five additional blocks of master pointers.

Listing 1-4 Setting up your application heap and stack

```
PROCEDURE DoSetUpHeap;
CONST
   kExtraStackSpace = $8000;              {32 KB}
   kMoreMasterCalls = 5;                  {for 320 master ptrs}
VAR
   count:   Integer;
BEGIN
   IncreaseStackSize(kExtraStackSpace);   {increase stack size}
   MaxApplZone;                           {extend heap to limit}
   FOR count := 1 TO kMoreMasterCalls DO
      MoreMasters;                        {64 more master ptrs}
END;
```

To reduce heap fragmentation, you should call `DoSetUpHeap` in a code segment that you never unload (possibly the main segment) rather than in a special initialization code segment. This is because `MoreMasters` allocates a nonrelocatable block. If you call `MoreMasters` from a code segment that is later purged, the new master pointer block is located above the purged space, thereby increasing fragmentation.

Determining the Amount of Free Memory

Because space in your heap is limited, you cannot usually honor every user request that would require your application to allocate memory. For example, every time the user opens a new window, you probably need to allocate a new window record and other associated data structures. If you allow the user to open windows endlessly, you risk running out of memory. This might adversely affect your application's ability to perform important operations such as saving existing data in a window.

It is important, therefore, to implement some scheme that prevents your application from using too much of its own heap. One way to do this is to maintain a memory cushion that can be used only to satisfy essential memory requests. Before allocating memory for any nonessential task, you need to ensure that the amount of memory that

remains free after the allocation exceeds the size of your memory cushion. You can do this by calling the function `IsMemoryAvailable` defined in Listing 1-5.

Listing 1-5 Determining whether allocating memory would deplete the memory cushion

```
FUNCTION IsMemoryAvailable (memRequest: LongInt): Boolean;
VAR
   total:   LongInt;    {total free memory if heap purged}
   contig:  LongInt;    {largest contiguous block if heap purged}
BEGIN
   PurgeSpace(total, contig);
   IsMemoryAvailable := ((memRequest + kMemCushion) < contig);
END;
```

The `IsMemoryAvailable` function calls the Memory Manager's `PurgeSpace` procedure to determine the size of the largest contiguous block that would be available if the application heap were purged; that size is returned in the `contig` parameter. If the size of the potential memory request together with the size of the memory cushion is less than the value returned in `contig`, `IsMemoryAvailable` is set to `TRUE`, indicating that it is safe to allocate the specified amount of memory; otherwise, `IsMemoryAvailable` returns `FALSE`.

Notice that the `IsMemoryAvailable` function does not itself cause the heap to be purged or compacted; the Memory Manager does so automatically when you actually attempt to allocate the memory.

Usually, the easiest way to determine how big to make your application's memory cushion is to experiment with various values. You should attempt to find the lowest value that allows your application to execute successfully no matter how hard you try to allocate memory to make the application crash. As an extra guarantee against your application's crashing, you might want to add some memory to this value. As indicated earlier in this chapter, 40 KB is a reasonable size for most applications.

```
CONST
   kMemCushion =  40 * 1024;           {size of memory cushion}
```

You should call the `IsMemoryAvailable` function before all nonessential memory requests, no matter how small. For example, suppose your application allocates a new, small relocatable block each time a user types a new line of text. That block might be small, but thousands of such blocks could take up a considerable amount of space. Therefore, you should check to see if there is sufficient memory available before allocating each one. (See Listing 1-6 on page 1-44 for an example of how to call `IsMemoryAvailable`.)

You should never, however, call the `IsMemoryAvailable` function before an essential memory request. When deciding how big to make the memory cushion for your application, you must make sure that essential requests can never deplete all of the cushion. Note that when you call the `IsMemoryAvailable` function for a nonessential

request, essential requests might have already dipped into the memory cushion. In that case, `IsMemoryAvailable` returns `FALSE` no matter how small the nonessential request is.

Some actions should never be rejectable. For example, you should guarantee that there is always enough memory free to save open documents, and to perform typical maintenance tasks such as updating windows. Other user actions are likely to be always rejectable. For example, because you cannot allow the user to create an endless number of documents, you should make the New Document and Open Document menu commands rejectable.

Although the decisions of which actions to make rejectable are usually obvious, modal and modeless boxes present special problems. If you want to make such dialog boxes available at all costs, you must ensure that you allocate a large enough memory cushion to handle the maximum number of these dialog boxes that the user could open at once. If you consider a certain dialog box (for instance, a spelling checker) nonessential, you must be prepared to inform the user that there is not enough memory to open it if memory space become low.

Allocating Blocks of Memory

As you have seen, a key element of the memory-management scheme presented in this chapter is to disallow any nonessential memory allocation requests that would deplete the memory cushion. In practice, this means that, before calling `NewHandle`, `NewPtr`, or another function that allocates memory, you should check that the amount of space remaining after the allocation, if successful, exceeds the size of the memory cushion.

An easy way to do this is never to allocate memory for nonessential tasks by calling `NewHandle` or `NewPtr` directly. Instead call a function such as `NewHandleCushion`, defined in Listing 1-6, or `NewPtrCushion`, defined in Listing 1-7.

Listing 1-6 Allocating relocatable blocks

```
FUNCTION NewHandleCushion (logicalSize: Size): Handle;
BEGIN
   IF NOT IsMemoryAvailable(logicalSize) THEN
      NewHandleCushion := NIL
   ELSE
      BEGIN
         SetGrowZone(NIL);          {remove grow-zone function}
         NewHandleCushion := NewHandleClear(logicalSize);
         SetGrowZone(@MyGrowZone);  {install grow-zone function}
      END;
END;
```

The `NewHandleCushion` function first calls `IsMemoryAvailable` to determine whether allocating the requested number of bytes would deplete the memory cushion.

If so, NewHandleCushion returns NIL to indicate that the request has failed. Otherwise, if there is indeed sufficient space for the new block, NewHandleCushion calls NewHandleClear to allocate the relocatable block. Before calling NewHandleClear, however, NewHandleCushion disables the grow-zone function for the application heap. This prevents the grow-zone function from releasing any emergency memory reserve your application might be maintaining. See "Defining a Grow-Zone Function" on page 1-48 for details on grow-zone functions.

You can define a function NewPtrCushion to handle allocation of nonrelocatable blocks, as shown in Listing 1-7.

Listing 1-7 Allocating nonrelocatable blocks

```
FUNCTION NewPtrCushion (logicalSize: Size): Handle;
BEGIN
   IF NOT IsMemoryAvailable(logicalSize) THEN
      NewPtrCushion := NIL
   ELSE
      BEGIN
         SetGrowZone(NIL);          {remove grow-zone function}
         NewPtrCushion := NewPtrClear(logicalSize);
         SetGrowZone(@MyGrowZone);  {install grow-zone function}
      END;
END;
```

Note

The functions NewHandleCushion and NewPtrCushion allocate prezeroed blocks in your application heap. You can easily modify those functions if you do not want the blocks prezeroed. ◆

Listing 1-8 illustrates a typical way to call NewPtrCushion.

Listing 1-8 Allocating a dialog record

```
FUNCTION GetDialog (dialogID: Integer): DialogPtr;
VAR
   myPtr: Ptr;                {storage for the dialog record}
BEGIN
   myPtr := NewPtrCushion(SizeOf(DialogRecord));
   IF MemError = noErr THEN
      GetDialog := GetNewDialog(dialogID, myPtr, WindowPtr(-1))
   ELSE
      GetDialog := NIL;       {can't get memory}
END;
```

When you allocate memory directly, you can later release it by calling the `DisposeHandle` and `DisposePtr` procedures. When you allocate memory indirectly by calling a Toolbox routine, there is always a corresponding Toolbox routine to release that memory. For example, the `DisposeWindow` procedure releases memory allocated with the `NewWindow` function. Be sure to use these special Toolbox routines instead of the generic Memory Manager routines when applicable.

Maintaining a Memory Reserve

A simple way to help ensure that your application always has enough memory available for essential operations is to maintain an emergency memory reserve. This **memory reserve** is a block of memory that your application uses only for essential operations and only when all other heap space has been allocated. This section illustrates one way to implement a memory reserve in your application.

To create and maintain an emergency memory reserve, you follow three distinct steps:

- When your application starts up, you need to allocate a block of reserve memory. Because you allocate the block, it is no longer free in the heap and does not enter into the free-space determination done by `IsMemoryAvailable`.
- When your application needs to fulfill an essential memory request and there isn't enough space in your heap to satisfy the request, you can release the reserve. This effectively ensures that you always have the memory you request, at least for essential operations. You can use a grow-zone function to release the reserve when necessary; see "Defining a Grow-Zone Function" on page 1-48 for details.
- Each time through your main event loop, you should check whether the reserve has been released. If it has, you should attempt to recover the reserve. If you cannot recover the reserve, you should warn the user that memory is critically short.

To refer to the emergency reserve, you can declare a global variable of type `Handle`.

```
VAR
   gEmergencyMemory: Handle;  {handle to emergency memory reserve}
```

Listing 1-9 defines a function that you can call early in your application's execution (before entering your main event loop) to create an emergency memory reserve. This function also installs the application-defined grow-zone procedure. See "Defining a Grow-Zone Function" on page 1-48 for a description of the grow-zone function.

Listing 1-9 Creating an emergency memory reserve

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

The `InitializeEmergencyMemory` procedure defined in Listing 1-9 simply allocates a relocatable block of a predefined size. That block is the emergency memory reserve. A reasonable size for the memory reserve is whatever size you use for the memory cushion. Once again, 40 KB is a good size for many applications.

```
CONST
   kEmergencyMemorySize =  40 * 1024;  {size of memory reserve}
```

When using a memory reserve, you need to change the `IsMemoryAvailable` function defined earlier in Listing 1-5. You need to make sure, when determining whether a nonessential memory allocation request should be honored, that the memory reserve has not been released. To check that the memory reserve is intact, use the function `IsEmergencyMemory` defined in Listing 1-10.

Listing 1-10 Checking the emergency memory reserve

```
FUNCTION IsEmergencyMemory: Boolean;
BEGIN
   IsEmergencyMemory :=
      (gEmergencyMemory <> NIL) & (gEmergencyMemory^ <> NIL);
END;
```

Then, you can replace the function `IsMemoryAvailable` defined in Listing 1-5 (page 1-43) by the version defined in Listing 1-11.

Listing 1-11 Determining whether allocating memory would deplete the memory cushion

```
FUNCTION IsMemoryAvailable (memRequest: LongInt): Boolean;
VAR
   total:   LongInt;    {total free memory if heap purged}
   contig:  LongInt;    {largest contiguous block if heap purged}
BEGIN
   IF NOT IsEmergencyMemory THEN {is emergency memory available?}
      IsMemoryAvailable := FALSE
   ELSE
   BEGIN
      PurgeSpace(total, contig);
      IsMemoryAvailable := ((memRequest + kMemCushion) < contig);
   END;
END;
```

As you can see, this is exactly like the earlier version except that it indicates that memory is not available if the memory reserve is not intact.

Once you have allocated the memory reserve early in your application's execution, it should be released only to honor essential memory requests when there is no other space available in your heap. You can install a simple grow-zone function that takes care of releasing the reserve at the proper moment. Each time through your main event loop, you can check whether the reserve is still intact; to do this, add these lines of code to your main event loop, before you make your event call:

```
IF NOT IsEmergencyMemory THEN
   RecoverEmergencyMemory;
```

The `RecoverEmergencyMemory` function, defined in Listing 1-12, simply attempts to reallocate the memory reserve.

Listing 1-12 Reallocating the emergency memory reserve

```
PROCEDURE RecoverEmergencyMemory;
BEGIN
   ReallocateHandle(gEmergencyMemory, kEmergencyMemorySize);
END;
```

If you are unable to reallocate the memory reserve, you might want to notify the user that because memory is in short supply, steps should be taken to save any important data and to free some memory.

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. For example, your grow-zone function might dispose of certain blocks or make some unpurgeable blocks purgeable. Your function should return the number of bytes, if any, it managed to free.

When the function returns, the Memory Manager once again purges and compacts the heap and tries again to allocate the requested amount of memory. If there is still insufficient memory, the Memory Manager calls your grow-zone function again, but only if the function returned a nonzero value when last called. This mechanism allows your grow-zone function to release memory gradually; if the amount it releases is not enough, the Memory Manager calls it again and gives it the opportunity to take more drastic measures.

Typically a grow-zone function frees space by calling the `EmptyHandle` procedure, which purges a relocatable block from the heap and sets the block's master pointer to `NIL`. This is preferable to disposing of the space (by calling the `DisposeHandle` procedure), because you are likely to want to reallocate the block.

The Memory Manager might designate a particular relocatable block in the heap as **protected;** your grow-zone function should not move or purge that block. You can determine which block, if any, the Memory Manager has protected by calling the `GZSaveHnd` function in your grow-zone function.

Listing 1-13 defines a very basic grow-zone function. The `MyGrowZone` function attempts to create space in the application heap simply by releasing the block of emergency memory. First, however, it checks that (1) the emergency memory hasn't already been released and (2) the emergency memory is not a protected block of memory (as it would be, for example, during an attempt to reallocate the emergency memory block). If either of these conditions isn't true, then `MyGrowZone` returns 0 to indicate that no memory was released.

Listing 1-13 A grow-zone function that releases emergency storage

```
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 `MyGrowZone` defined in Listing 1-13 saves the current value of the A5 register when it begins and then restores the previous value before it exits. This is necessary because your grow-zone function might be called at a time when the system is attempting to allocate memory and value in the A5 register is not correct. See the chapter "Memory Management Utilities" in this book for more information about saving and restoring the A5 register.

Note

You need to save and restore the A5 register only if your grow-zone function accesses your A5 world. (In Listing 1-13, the grow-zone function uses the global variable `gEmergencyMemory`.) ◆

Memory Management Reference

This section describes the routines used to illustrate the memory-management techniques presented earlier in this chapter. In particular, it describes the routines that allow you to manipulate blocks of memory in your application heap.

Note
For a complete description of all Memory Manager data types and routines, see the chapter "Memory Manager" in this book. ◆

Memory Management Routines

This section describes the routines you can use to set up your application's heap, allocate and dispose of relocatable and nonrelocatable blocks, manipulate those blocks, assess the availability of memory in your application's heap, free memory from the heap, and install a grow-zone function for your heap.

Note
The result codes listed for Memory Manager routines are usually not directly returned to your application. You need to call the `MemError` function (or, from assembly language, inspect the `MemErr` global variable) to get a routine's result code. ◆

You cannot call most Memory Manager routines at interrupt time for several reasons. You cannot allocate memory at interrupt time because the Memory Manager might already be handling a memory-allocation request and the heap might be in an inconsistent state. More generally, you cannot call at interrupt time any Memory Manager routine that returns its result code via the `MemError` function, even if that routine doesn't allocate or move memory. Resetting the `MemErr` global variable at interrupt time can lead to unexpected results if the interrupted code depends on the value of `MemErr`. Note that Memory Manager routines like `HLock` return their results via `MemError` and therefore should not be called in interrupt code.

Setting Up the Application Heap

The Operating System automatically initializes your application's heap when your application is launched. To help prevent heap fragmentation, you should call the procedures in this section before you allocate any blocks of memory in your heap.

Use the `MaxApplZone` procedure to extend the application heap zone to the application heap limit so that the Memory Manager does not do so gradually as memory requests require. Use the `MoreMasters` procedure to preallocate enough blocks of master pointers so that the Memory Manager never needs to allocate new master pointer blocks for you.

MaxApplZone

To help ensure that you can use as much of the application heap zone as possible, call the `MaxApplZone` procedure. Call this once near the beginning of your program, after you have expanded your stack.

```
PROCEDURE MaxApplZone;
```

DESCRIPTION

The `MaxApplZone` procedure expands the application heap zone to the application heap limit. If you do not call `MaxApplZone`, the application heap zone grows as necessary to fulfill memory requests. The `MaxApplZone` procedure does not purge any blocks currently in the zone. If the zone already extends to the limit, `MaxApplZone` does nothing.

It is a good idea to call `MaxApplZone` once at the beginning of your program if you intend to maintain an effectively partitioned heap. If you do not call `MaxApplZone` and then call `MoveHHi` to move relocatable blocks to the top of the heap zone before locking them, the heap zone could later grow beyond these locked blocks to fulfill a memory request. If the Memory Manager were to allocate a nonrelocatable block in this new space, your heap would be fragmented.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for `MaxApplZone` are

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error

MoreMasters

Call the `MoreMasters` procedure several times at the beginning of your program to prevent the Memory Manager from running out of master pointers in the middle of application execution. If it does run out, it allocates more, possibly causing heap fragmentation.

```
PROCEDURE MoreMasters;
```

DESCRIPTION

The `MoreMasters` procedure allocates another block of master pointers in the current heap zone. In the application heap, a block of master pointers consists of 64 master pointers, and in the system heap, a block consists of 32 master pointers. (These values, however, might change in future versions of system software.) When you initialize additional heap zones, you can specify the number of master pointers you want to have in a block of master pointers.

The Memory Manager automatically calls `MoreMasters` once for every new heap zone, including the application heap zone.

You should call `MoreMasters` at the beginning of your program enough times to ensure that the Memory Manager never needs to call it for you. For example, if your application never allocates more than 300 relocatable blocks in its heap zone, then five calls to the `MoreMasters` should be enough. It's better to call `MoreMasters` too many times than too few. For instance, if your application usually allocates about 100 relocatable blocks but might allocate 1000 in a particularly busy session, call `MoreMasters` enough times at the beginning of the program to accommodate times of greater memory use.

If you are forced to call `MoreMasters` so many times that it causes a significant slowdown, you could change the `moreMast` field of the zone header to the total number of master pointers you need and then call `MoreMasters` just once. Afterward, be sure to restore the `moreMast` field to its original value.

SPECIAL CONSIDERATIONS

Because `MoreMasters` allocates memory, you should not call it at interrupt time.

The calls to `MoreMasters` at the beginning of your application should be in the main code segment of your application or in a segment that the main segment never unloads.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for `MoreMasters` are

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough memory

GetApplLimit

Use the `GetApplLimit` function to get the application heap limit, beyond which the application heap cannot expand.

```
FUNCTION GetApplLimit: Ptr;
```

DESCRIPTION

The `GetApplLimit` function returns the current application heap limit. The Memory Manager expands the application heap only up to the byte preceding this limit.

Nothing prevents the stack from growing below the application limit. If the Operating System detects that the stack has crashed into the heap, it generates a system error. To avoid this, use `GetApplLimit` and the `SetApplLimit` procedure to set the application limit low enough so that a growing stack does not encounter the heap.

Note
The `GetApplLimit` function does not indicate the amount of memory available to your application. ◆

ASSEMBLY-LANGUAGE INFORMATION

The global variable `ApplLimit` contains the current application heap limit.

SetApplLimit

Use the `SetApplLimit` procedure to set the application heap limit, beyond which the application heap cannot expand.

```
PROCEDURE SetApplLimit (zoneLimit: Ptr);
```

`zoneLimit` A pointer to a byte in memory demarcating the upper boundary of the application heap zone. The zone can grow to include the byte preceding `zoneLimit` in memory, but no further.

DESCRIPTION

The `SetApplLimit` procedure sets the current application heap limit to `zoneLimit`. The Memory Manager then can expand the application heap only up to the byte

preceding the application limit. If the zone already extends beyond the specified limit, the Memory Manager does not cut it back but does prevent it from growing further.

Note

The `zoneLimit` parameter is not a byte count, but an absolute byte in memory. Thus, you should use the `SetApplLimit` procedure only with a value obtained from the Memory Manager functions `GetApplLimit` or `ApplicationZone`. ◆

You cannot change the limit of zones other than the application heap zone.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `SetApplLimit` are

Registers on entry

A0 Pointer to desired new zone limit

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough memory

SEE ALSO

To use `SetApplLimit` to expand the default size of the stack, see the discussion in "Changing the Size of the Stack" on page 1-39.

Allocating and Releasing Relocatable Blocks of Memory

You can use the `NewHandle` function to allocate a relocatable block of memory. If you want to allocate new blocks of memory with their bits precleared to 0, you can use the `NewHandleClear` function.

▲ **WARNING**

You should not call any of these memory-allocation routines at interrupt time. ▲

You can use the `DisposeHandle` procedure to free relocatable blocks of memory you have allocated.

NewHandle

You can use the `NewHandle` function to allocate a relocatable memory block of a specified size.

```
FUNCTION NewHandle (logicalSize: Size): Handle;
```

`logicalSize`
The requested size (in bytes) of the relocatable block.

DESCRIPTION

The `NewHandle` function attempts to allocate a new relocatable block in the current heap zone with a logical size of `logicalSize` bytes and then return a handle to the block. The new block is unlocked and unpurgeable. If `NewHandle` cannot allocate a block of the requested size, it returns `NIL`.

▲ **WARNING**
Do not try to manufacture your own handles without this function by simply assigning the address of a variable of type `Ptr` to a variable of type `Handle`. The resulting "fake handle" would not reference a relocatable block and could cause a system crash. ▲

The `NewHandle` function pursues all available avenues to create a block of the requested size, including compacting the heap zone, increasing its size, and purging blocks from it. If all of these techniques fail and the heap zone has a grow-zone function installed, `NewHandle` calls the function. Then `NewHandle` tries again to free the necessary amount of memory, once more compacting and purging the heap zone if necessary. If memory still cannot be allocated, `NewHandle` calls the grow-zone function again, unless that function had returned 0, in which case `NewHandle` gives up and returns `NIL`.

SPECIAL CONSIDERATIONS

Because `NewHandle` allocates memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `NewHandle` are

Registers on entry

A0 Number of logical bytes requested

Registers on exit

A0 Address of the new block's master pointer or `NIL`

D0 Result code

If you want to clear the bytes of a block of memory to 0 when you allocate it with the `NewHandle` function, set bit 9 of the routine trap word. You can usually do this by supplying the word `CLEAR` as the second argument to the routine macro, as follows:

```
_NewHandle ,CLEAR
```

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough memory in heap zone

SEE ALSO

If you allocate a relocatable block that you plan to lock for long periods of time, you can prevent heap fragmentation by allocating the block as low as possible in the heap zone. To do this, see the description of the `ReserveMem` procedure on page 1-70.

If you plan to lock a relocatable block for short periods of time, you might want to move it to the top of the heap zone to prevent heap fragmentation. For more information, see the description of the `MoveHHi` procedure on page 1-71.

NewHandleClear

You can use the `NewHandleClear` function to allocate prezeroed memory in a relocatable block of a specified size.

```
FUNCTION NewHandleClear (logicalSize: Size): Handle;
```

`logicalSize`
The requested size (in bytes) of the relocatable block. The `NewHandleClear` function sets each of these bytes to 0.

DESCRIPTION

The `NewHandleClear` function works much as the `NewHandle` function does but sets all bytes in the new block to 0 instead of leaving the contents of the block undefined.

Currently, `NewHandleClear` clears the block one byte at a time. For a large block, it might be faster to clear the block manually a long word at a time.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough memory in heap zone

DisposeHandle

When you are completely done with a relocatable block, call the `DisposeHandle` procedure to free it and its master pointer for other uses.

```
PROCEDURE DisposeHandle (h: Handle);
```

`h` A handle to a relocatable block.

DESCRIPTION

The `DisposeHandle` procedure releases the memory occupied by the relocatable block whose handle is `h`. It also frees the handle's master pointer for other uses.

▲ **WARNING**
After a call to `DisposeHandle`, all handles to the released block become invalid and should not be used again. Any subsequent calls to `DisposeHandle` using an invalid handle might damage the master pointer list. ▲

Do not use `DisposeHandle` to dispose of a handle obtained from the Resource Manager (for example, by a previous call to `GetResource`); use `ReleaseResource` instead. If, however, you have called `DetachResource` on a resource handle, you should dispose of the storage by calling `DisposeHandle`.

SPECIAL CONSIDERATIONS

Because `DisposeHandle` purges memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `DisposeHandle` are

Registers on entry

A0 Handle to the relocatable block to be disposed of

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`memWZErr`	–111	Attempt to operate on a free block

Allocating and Releasing Nonrelocatable Blocks of Memory

You can use the `NewPtr` function to allocate a nonrelocatable block of memory. If you want to allocate new blocks of memory with their bits precleared to 0, you can use the `NewPtrClear` function.

▲ **WARNING**
You should not call any of these memory-allocation routines at interrupt time. ▲

You can use the `DisposePtr` procedure to free nonrelocatable blocks of memory you have allocated.

NewPtr

You can use the `NewPtr` function to allocate a nonrelocatable block of memory of a specified size.

```
FUNCTION NewPtr (logicalSize: Size): Ptr;
```

`logicalSize`
The requested size (in bytes) of the nonrelocatable block.

DESCRIPTION

The `NewPtr` function attempts to allocate, in the current heap zone, a nonrelocatable block with a logical size of `logicalSize` bytes and then return a pointer to the block. If the requested number of bytes cannot be allocated, `NewPtr` returns `NIL`.

The `NewPtr` function attempts to reserve space as low in the heap zone as possible for the new block. If it is able to reserve the requested amount of space, `NewPtr` allocates the nonrelocatable block in the gap `ReserveMem` creates. Otherwise, `NewPtr` returns `NIL` and generates a `memFullErr` error.

SPECIAL CONSIDERATIONS

Because `NewPtr` allocates memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `NewPtr` are

Registers on entry

A0 Number of logical bytes requested

Registers on exit

A0 Address of the new block or `NIL`

D0 Result code

If you want to clear the bytes of a block of memory to 0 when you allocate it with the `NewPtr` function, set bit 9 of the routine trap word. You can usually do this by supplying the word `CLEAR` as the second argument to the routine macro, as follows:

```
_NewPtr ,CLEAR
```

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough memory

NewPtrClear

You can use the `NewPtrClear` function to allocate prezeroed memory in a nonrelocatable block of a specified size.

```
FUNCTION NewPtrClear (logicalSize: Size): Ptr;
```

`logicalSize`
The requested size (in bytes) of the nonrelocatable block.

DESCRIPTION

The `NewPtrClear` function works much as the `NewPtr` function does, but sets all bytes in the new block to 0 instead of leaving the contents of the block undefined.

Currently, `NewPtrClear` clears the block one byte at a time. For a large block, it might be faster to clear the block manually a long word at a time.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough memory

DisposePtr

When you are completely done with a nonrelocatable block, call the `DisposePtr` procedure to free it for other uses.

```
PROCEDURE DisposePtr (p: Ptr);
```

`p` A pointer to the nonrelocatable block you want to dispose of.

DESCRIPTION

The `DisposePtr` procedure releases the memory occupied by the nonrelocatable block specified by `p`.

▲ **WARNING**
After a call to `DisposePtr`, all pointers to the released block become invalid and should not be used again. Any subsequent use of a pointer to the released block might cause a system error. ▲

SPECIAL CONSIDERATIONS

Because `DisposePtr` purges memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `DisposePtr` are

Registers on entry

A0 Pointer to the nonrelocatable block to be disposed of

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`memWZErr`	–111	Attempt to operate on a free block

Setting the Properties of Relocatable Blocks

A relocatable block can be either locked or unlocked and either purgeable or unpurgeable. In addition, it can have its resource bit either set or cleared. To determine the state of any of these properties, use the `HGetState` function. To change these

properties, use the `HLock`, `HUnlock`, `HPurge`, `HNoPurge`, `HSetRBit`, and `HClrRBit` procedures. To restore these properties, use the `HSetState` procedure.

▲ **WARNING**
Be sure to use these procedures to get and set the properties of relocatable blocks. In particular, do not rely on the structure of master pointers, because their structure in 24-bit mode is different from their structure in 32-bit mode. ▲

HGetState

You can use the `HGetState` function to get the current properties of a relocatable block (perhaps so that you can change and then later restore those properties).

```
FUNCTION HGetState (h: Handle): SignedByte;
```

`h` A handle to a relocatable block.

DESCRIPTION

The `HGetState` function returns a signed byte containing the flags of the master pointer for the given handle. You can save this byte, change the state of any of the flags, and then restore their original states by passing the byte to the `HSetState` procedure, described next.

You can use bit-manipulation functions on the returned signed byte to determine the value of a given attribute. Currently the following bits are used:

Bit	Meaning
0–4	Reserved
5	Set if relocatable block is a resource
6	Set if relocatable block is purgeable
7	Set if relocatable block is locked

If an error occurs during an attempt to get the state flags of the specified relocatable block, `HGetState` returns the low-order byte of the result code as its function result. For example, if the handle `h` points to a master pointer whose value is `NIL`, then the signed byte returned by `HGetState` will contain the value –109.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HGetState` are

Registers on entry

A0 Handle whose properties you want to get

Registers on exit

D0 Byte containing flags

RESULT CODES

`noErr`	0	No error
`nilHandleErr`	–109	`NIL` master pointer
`memWZErr`	–111	Attempt to operate on a free block

HSetState

You can use the `HSetState` procedure to restore properties of a block after a call to `HGetState`.

```
PROCEDURE HSetState (h: Handle; flags: SignedByte);
```

`h`	A handle to a relocatable block.
`flags`	A signed byte specifying the properties to which you want to set the relocatable block.

DESCRIPTION

The `HSetState` procedure restores to the handle `h` the properties specified in the `flags` signed byte. See the description of the `HGetState` function for a list of the currently used bits in that byte. Because additional bits of the `flags` byte could become significant in future versions of system software, use `HSetState` only with a byte returned by `HGetState`. If you need to set two or three properties of a relocatable block at once, it is better to use the procedures that set individual properties than to manipulate the bits returned by `HGetState` and then call `HSetState`.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HSetState` are

Registers on entry

A0 Handle whose properties you want to set

D0 Byte containing flags indicating the handle's new properties

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`nilHandleErr`	–109	`NIL` master pointer
`memWZErr`	–111	Attempt to operate on a free block

HLock

You can use the `HLock` procedure to lock a relocatable block so that it does not move in the heap. If you plan to dereference a handle and then allocate, move, or purge memory (or call a routine that does so), then you should lock the handle before using the dereferenced handle.

```
PROCEDURE HLock (h: Handle);
```

`h` A handle to a relocatable block.

DESCRIPTION

The `HLock` procedure locks the relocatable block to which `h` is a handle, preventing it from being moved within its heap zone. If the block is already locked, `HLock` does nothing.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HLock` are

Registers on entry

A0 Handle to lock

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`nilHandleErr`	–109	`NIL` master pointer
`memWZErr`	–111	Attempt to operate on a free block

SEE ALSO

If you plan to lock a relocatable block for long periods of time, you can prevent fragmentation by ensuring that the block is as low as possible in the heap zone. To do this, see the description of the `ReserveMem` procedure on page 1-70.

If you plan to lock a relocatable block for short periods of time, you can prevent heap fragmentation by moving the block to the top of the heap zone before locking. For more information, see the description of the `MoveHHi` procedure on page 1-71.

HUnlock

You can use the `HUnlock` procedure to unlock a relocatable block so that it is free to move in its heap zone.

```
PROCEDURE HUnlock (h: Handle);
```

`h` A handle to a relocatable block.

DESCRIPTION

The `HUnlock` procedure unlocks the relocatable block to which `h` is a handle, allowing it to be moved within its heap zone. If the block is already unlocked, `HUnlock` does nothing.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HUnlock` are

Registers on entry

A0 Handle to unlock

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`nilHandleErr`	–109	`NIL` master pointer
`memWZErr`	–111	Attempt to operate on a free block

HPurge

You can use the `HPurge` procedure to mark a relocatable block so that it can be purged if a memory request cannot be fulfilled after compaction.

```
PROCEDURE HPurge (h: Handle);
```

`h` A handle to a relocatable block.

DESCRIPTION

The `HPurge` procedure makes the relocatable block to which `h` is a handle purgeable. If the block is already purgeable, `HPurge` does nothing.

The Memory Manager might purge the block when it needs to purge the heap zone containing the block to satisfy a memory request. A direct call to the `PurgeMem` procedure or the `MaxMem` function would also purge blocks marked as purgeable.

Once you mark a relocatable block as purgeable, you should make sure that handles to the block are not empty before you access the block. If they are empty, you must reallocate space for the block and recopy the block's data from another source, such as a resource file, before using the information in the block.

If the block to which `h` is a handle is locked, `HPurge` does not unlock the block but does mark it as purgeable. If you later call `HUnlock` on `h`, the block is subject to purging.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HPurge` are

Registers on entry

A0 Handle to make purgeable

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`nilHandleErr`	–109	`NIL` master pointer
`memWZErr`	–111	Attempt to operate on a free block

SEE ALSO

If the Memory Manager has purged a block, you can reallocate space for it by using the `ReallocateHandle` procedure, described on page 1-68.

You can immediately free the space taken by a handle without disposing of it by calling `EmptyHandle`. This procedure, described on page 1-67, does not require that the block be purgeable.

HNoPurge

You can use the `HNoPurge` procedure to mark a relocatable block so that it cannot be purged.

```
PROCEDURE HNoPurge (h: Handle);
```

`h` A handle to a relocatable block.

DESCRIPTION

The `HNoPurge` procedure makes the relocatable block to which `h` is a handle unpurgeable. If the block is already unpurgeable, `HNoPurge` does nothing.

The `HNoPurge` procedure does not reallocate memory for a handle if it has already been purged.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HNoPurge` are

Registers on entry

A0 Handle to make unpurgeable

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`nilHandleErr`	–109	`NIL` master pointer
`memWZErr`	–111	Attempt to operate on a free block

SEE ALSO

If you want to reallocate memory for a relocatable block that has already been purged, you can use the `ReallocateHandle` procedure, described in the next section, "Managing Relocatable Blocks."

Managing Relocatable Blocks

The Memory Manager provides routines that allow you to purge and later reallocate space for relocatable blocks and control where in their heap zone relocatable blocks are located.

To free the memory taken up by a relocatable block without releasing the master pointer to the block for other uses, use the `EmptyHandle` procedure. To reallocate space for a handle that you have emptied or the Memory Manager has purged, use the `ReallocateHandle` procedure.

To ensure that a relocatable block that you plan to lock for short or long periods of time does not cause heap fragmentation, use the `MoveHHi` and the `ReserveMem` procedures, respectively.

EmptyHandle

The `EmptyHandle` procedure allows you to free memory taken by a relocatable block without freeing the relocatable block's master pointer for other uses.

```
PROCEDURE EmptyHandle (h: Handle);
```

`h` A handle to a relocatable block.

DESCRIPTION

The `EmptyHandle` procedure purges the relocatable block whose handle is `h` and sets the handle's master pointer to `NIL`. The block whose handle is `h` must be unlocked but need not be purgeable.

Note

If there are multiple handles to the relocatable block, then calling the `EmptyHandle` procedure empties them all, because all of the handles share a common master pointer. When you later use `ReallocateHandle` to reallocate space for the block, the master pointer is updated, and all of the handles reference the new block correctly. ◆

SPECIAL CONSIDERATIONS

Because `EmptyHandle` purges memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `EmptyHandle` are

Registers on entry

A0 Handle to relocatable block

Registers on exit

A0 Handle to relocatable block

D0 Result code

RESULT CODES

`noErr`	0	No error
`memWZErr`	–111	Attempt to operate on a free block
`memPurErr`	–112	Attempt to purge a locked block

SEE ALSO

To free the memory taken up by a relocatable block and release the block's master pointer for other uses, use the `DisposeHandle` procedure, described on page 1-57.

ReallocateHandle

To recover space for a relocatable block that you have emptied or the Memory Manager has purged, use the `ReallocateHandle` procedure.

```
PROCEDURE ReallocateHandle (h: Handle; logicalSize: Size);
```

h A handle to a relocatable block.

logicalSize
The desired new logical size (in bytes) of the relocatable block.

DESCRIPTION

The ReallocateHandle procedure allocates a new relocatable block with a logical size of logicalSize bytes. It updates the handle h by setting its master pointer to point to the new block. The new block is unlocked and unpurgeable.

Usually you use ReallocateHandle to reallocate space for a block that you have emptied or the Memory Manager has purged. If the handle references an existing block, ReallocateHandle releases that block before creating a new one.

Note

To reallocate space for a resource that has been purged, you should call LoadResource, not ReallocateHandle. ◆

If many handles reference a single purged, relocatable block, you need to call ReallocateHandle on just one of them.

In case of an error, ReallocateHandle neither allocates a new block nor changes the master pointer to which handle h points.

SPECIAL CONSIDERATIONS

Because ReallocateHandle might purge and allocate memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for ReallocateHandle are

Registers on entry

A0 Handle for new relocatable block

D0 Desired logical size, in bytes, of new block

Registers on exit

D0 Result code

RESULT CODES

noErr	0	No error
memROZErr	–99	Heap zone is read-only
memFullErr	–108	Not enough memory
memWZErr	–111	Attempt to operate on a free block
memPurErr	–112	Attempt to purge a locked block

ReserveMem

Use the `ReserveMem` procedure when you allocate a relocatable block that you intend to lock for long periods of time. This helps prevent heap fragmentation because it reserves space for the block as close to the bottom of the heap as possible. Consistent use of `ReserveMem` for this purpose ensures that all locked, relocatable blocks and nonrelocatable blocks are together at the bottom of the heap zone and thus do not prevent unlocked relocatable blocks from moving about the zone.

```
PROCEDURE ReserveMem (cbNeeded: Size);
```

`cbNeeded` The number of bytes to reserve near the bottom of the heap.

DESCRIPTION

The `ReserveMem` procedure attempts to create free space for a block of `cbNeeded` contiguous logical bytes at the lowest possible position in the current heap zone. It pursues every available means of placing the block as close as possible to the bottom of the zone, including moving other relocatable blocks upward, expanding the zone (if possible), and purging blocks from it.

Because `ReserveMem` does not actually allocate the block, you must combine calls to `ReserveMem` with calls to the `NewHandle` function.

Do not use the `ReserveMem` procedure for a relocatable block you intend to lock for only a short period of time. If you do so and then allocate a nonrelocatable block above it, the relocatable block becomes trapped under the nonrelocatable block when you unlock that relocatable block.

Note
It isn't necessary to call `ReserveMem` to reserve space for a nonrelocatable block, because the `NewPtr` function calls it automatically. Also, you do not need to call `ReserveMem` to reserve memory before you load a locked resource into memory, because the Resource Manager calls `ReserveMem` automatically. ◆

SPECIAL CONSIDERATIONS

Because the `ReserveMem` procedure could move and purge memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `ReserveMem` are

Registers on entry

D0 Number of bytes to reserve

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough memory

MoveHHi

If you plan to lock a relocatable block for a short period of time, use the `MoveHHi` procedure, which moves the block to the top of the heap and thus helps prevent heap fragmentation.

```
PROCEDURE MoveHHi (h: Handle);
```

`h` A handle to a relocatable block.

DESCRIPTION

The `MoveHHi` procedure attempts to move the relocatable block referenced by the handle `h` upward until it reaches a nonrelocatable block, a locked relocatable block, or the top of the heap.

▲ **WARNING**
If you call `MoveHHi` to move a handle to a resource that has its `resChanged` bit set, the Resource Manager updates the resource by using the `WriteResource` procedure to write the contents of the block to disk. If you want to avoid this behavior, call the Resource Manager procedure `SetResPurge(FALSE)` before you call `MoveHHi`, and then call `SetResPurge(TRUE)` to restore the default setting. ▲

By using the `MoveHHi` procedure on relocatable blocks you plan to allocate for short periods of time, you help prevent islands of immovable memory from accumulating in (and thus fragmenting) the heap.

Do not use the `MoveHHi` procedure to move blocks you plan to lock for long periods of time. The `MoveHHi` procedure moves such blocks to the top of the heap, perhaps preventing other blocks already at the top of the heap from moving down once they are unlocked. Instead, use the `ReserveMem` procedure before allocating such blocks, thus keeping them in the bottom partition of the heap, where they do not prevent relocatable blocks from moving.

If you frequently lock a block for short periods of time and find that calling `MoveHHi` each time slows down your application, you might consider leaving the block always locked and calling the `ReserveMem` procedure before allocating it.

Once you move a block to the top of the heap, be sure to lock it if you do not want the Memory Manager to move it back to the middle partition as soon as it can. (The `MoveHHi` procedure cannot move locked blocks; be sure to lock blocks after, not before, calling `MoveHHi`.)

Note

Using the `MoveHHi` procedure without taking other precautionary measures to prevent heap fragmentation is useless, because even one small nonrelocatable or locked relocatable block in the middle of the heap might prevent `MoveHHi` from moving blocks to the top of the heap. ◆

SPECIAL CONSIDERATIONS

Because the `MoveHHi` procedure moves memory, you should not call it at interrupt time.

Don't call `MoveHHi` on blocks in the system heap. Don't call `MoveHHi` from a desk accessory.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `MoveHHi` are

Registers on entry

A0 Handle to move

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`nilHandleErr`	–109	`NIL` master pointer
`memLockedErr`	–117	Block is locked

HLockHi

You can use the `HLockHi` procedure to move a relocatable block to the top of the heap and lock it.

```
PROCEDURE HLockHi (h: Handle);
```

`h` A handle to a relocatable block.

DESCRIPTION

The `HLockHi` procedure attempts to move the relocatable block referenced by the handle `h` upward until it reaches a nonrelocatable block, a locked relocatable block, or the top of the heap. Then `HLockHi` locks the block.

The `HLockHi` procedure is simply a convenient replacement for the pair of procedures `MoveHHi` and `HLock`.

SPECIAL CONSIDERATIONS

Because the `HLockHi` procedure moves memory, you should not call it at interrupt time.

Don't call `HLockHi` on blocks in the system heap. Don't call `HLockHi` from a desk accessory.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HLockHi` are

Registers on entry

A0 Handle to move and lock

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`nilHandleErr`	–109	`NIL` master pointer
`memWZErr`	–111	Attempt to operate on a free block
`memLockedErr`	–117	Block is locked

Manipulating Blocks of Memory

The Memory Manager provides a routine for copying blocks of memory referenced by pointers. To copy a block of memory to a nonrelocatable block, you can use the `BlockMove` procedure.

BlockMove

To copy a sequence of bytes from one location in memory to another, you can use the `BlockMove` procedure.

```
PROCEDURE BlockMove (sourcePtr, destPtr: Ptr; byteCount: Size);
```

`sourcePtr` The address of the first byte to copy.

`destPtr` The address of the first byte to copy to.

`byteCount` The number of bytes to copy. If the value of `byteCount` is 0, `BlockMove` does nothing.

DESCRIPTION

The `BlockMove` procedure moves a block of `byteCount` consecutive bytes from the address designated by `sourcePtr` to that designated by `destPtr`. It updates no pointers.

The `BlockMove` procedure works correctly even if the source and destination blocks overlap.

SPECIAL CONSIDERATIONS

You can safely call `BlockMove` at interrupt time. Even though it moves memory, `BlockMove` does not move relocatable blocks, but simply copies bytes.

The `BlockMove` procedure currently flushes the processor caches whenever the number of bytes to be moved is greater than 12. This behavior can adversely affect your application's performance. You might want to avoid calling `BlockMove` to move small amounts of data in memory if there is no possibility of moving stale data or instructions. For more information about stale data and instructions, see the discussion of the processor caches in the chapter "Memory Management Utilities" in this book.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `BlockMove` are

Registers on entry

A0 Pointer to source

A1 Pointer to destination

D0 Number of bytes to copy

Registers on exit

D0 Result code

RESULT CODE

noErr	0	No error

Assessing Memory Conditions

The Memory Manager provides routines to test how much memory is available. To determine the total amount of free space in the current heap zone or the size of the maximum block that could be obtained after a purge of the heap, call the PurgeSpace function.

To find out whether a Memory Manager operation finished successfully, use the MemError function.

PurgeSpace

Use the PurgeSpace procedure to determine the total amount of free memory and the size of the largest allocatable block after a purge of the heap.

```
PROCEDURE PurgeSpace (VAR total: LongInt; VAR contig: LongInt);
```

total	On exit, the total amount of free memory in the current heap zone if it were purged.
contig	On exit, the size of the largest contiguous block of free memory in the current heap zone if it were purged.

DESCRIPTION

The PurgeSpace procedure returns, in the total parameter, the total amount of space (in bytes) that could be obtained after a general purge of the current heap zone; this amount includes space that is already free. In the contig parameter, PurgeSpace returns the size of the largest allocatable block in the current heap zone that could be obtained after a purge of the zone.

The PurgeSpace procedure does not actually purge the current heap zone.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for PurgeSpace are

Registers on exit

A0	Maximum number of contiguous bytes after purge
D0	Total free memory after purge

RESULT CODES

noErr	0	No error

MemError

To find out whether your application's last direct call to a Memory Manager routine executed successfully, use the `MemError` function.

```
FUNCTION MemError: OSErr;
```

DESCRIPTION

The `MemError` function returns the result code produced by the last Memory Manager routine your application called directly.

This function is useful during application debugging. You might also use the function as one part of a memory-management scheme to identify instances in which the Memory Manager rejects overly large memory requests by returning the error code `memFullErr`.

▲ **WARNING**
Do not rely on the `MemError` function as the only component of a memory-management scheme. For example, suppose you call `NewHandle` or `NewPtr` and receive the result code `noErr`, indicating that the Memory Manager was able to allocate sufficient memory. In this case, you have no guarantee that the allocation did not deplete your application's memory reserves to levels so low that simple operations might cause your application to crash. Instead of relying on `MemError`, check before making a memory request that there is enough memory both to fulfill the request and to support essential operations. ▲

ASSEMBLY-LANGUAGE INFORMATION

Because most Memory Manager routines return a result code in register D0, you do not ordinarily need to call the `MemError` function if you program in assembly language. See the description of an individual routine to find out whether it returns a result code in register D0. If not, you can examine the global variable `MemErr`. When `MemError` returns, register D0 contains the result code.

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Error in parameter list
`memROZErr`	–99	Operation on a read-only zone
`memFullErr`	–108	Not enough memory
`nilHandleErr`	–109	`NIL` master pointer
`memWZErr`	–111	Attempt to operate on a free block
`memPurErr`	–112	Attempt to purge a locked block
`memBCErr`	–115	Block check failed
`memLockedErr`	–117	Block is locked

Grow-Zone Operations

You can implement a grow-zone function that the Memory Manager calls when it cannot fulfill a memory request. You should use the grow-zone function only as a last resort to free memory when all else fails.

The `SetGrowZone` procedure specifies which function the Memory Manager should use for the current zone. The grow-zone function should call the `GZSaveHnd` function to receive a handle to a relocatable block that the grow-zone function must not move or purge.

SetGrowZone

To specify a grow-zone function for the current heap zone, pass a pointer to that function to the `SetGrowZone` procedure. Ordinarily, you call this procedure early in the execution of your application.

If you initialize your own heap zones besides the application and system zones, you can alternatively specify a grow-zone function as a parameter to the `InitZone` procedure.

```
PROCEDURE SetGrowZone (growZone: ProcPtr);
```

`growZone` A pointer to the grow-zone function.

DESCRIPTION

The `SetGrowZone` procedure sets the current heap zone's grow-zone function as designated by the `growZone` parameter. A `NIL` parameter value removes any grow-zone function the zone might previously have had.

The Memory Manager calls the grow-zone function only after exhausting all other avenues of satisfying a memory request, including compacting the zone, increasing its size (if it is the original application zone and is not yet at its maximum size), and purging blocks from it.

See "Grow-Zone Functions" on page 1-80 for a complete description of a grow-zone function.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `SetGrowZone` are

Registers on entry

A0 Pointer to new grow-zone function

Registers on exit

D0 Result code

RESULT CODES

noErr	0	No error

SEE ALSO

See "Defining a Grow-Zone Function" on page 1-48 for a description of a grow-zone function.

GZSaveHnd

Your grow-zone function must call the GZSaveHnd function to obtain a handle to a protected relocatable block that the grow-zone function must not move, purge, or delete.

```
FUNCTION GZSaveHnd: Handle;
```

DESCRIPTION

The GZSaveHnd function returns a handle to a relocatable block that the grow-zone function must not move, purge, or delete. It returns NIL if there is no such block. The returned handle is a handle to the block of memory being manipulated by the Memory Manager at the time that the grow-zone function is called.

ASSEMBLY-LANGUAGE INFORMATION

You can find the same handle in the global variable GZRootHnd.

Setting and Restoring the A5 Register

Any code that runs asynchronously or as a callback routine and that accesses the calling application's A5 world must ensure that the A5 register correctly points to the boundary between the application parameters and the application global variables. To accomplish this, you can call the SetCurrentA5 function at the beginning of any asynchronous or callback code that isn't executed at interrupt time. If the code is executed at interrupt time, you must use the SetA5 function to set the value of the A5 register. (You determine this value at noninterrupt time by calling SetCurrentA5.) Then you must restore the A5 register to its previous value before the interrupt code returns.

SetCurrentA5

You can use the SetCurrentA5 function to get the current value of the system global variable CurrentA5.

```
FUNCTION SetCurrentA5: LongInt;
```

DESCRIPTION

The SetCurrentA5 function does two things: First, it gets the current value in the A5 register and returns it to your application. Second, SetCurrentA5 sets register A5 to the value of the low-memory global variable CurrentA5. This variable points to the boundary between the parameters and global variables of the current application.

SPECIAL CONSIDERATIONS

You cannot reliably call SetCurrentA5 in code that is executed at interrupt time unless you first guarantee that your application is the current process (for example, by calling the Process Manager function GetCurrentProcess). In general, you should call SetCurrentA5 at noninterrupt time and then pass the returned value to the interrupt code.

ASSEMBLY-LANGUAGE INFORMATION

You can access the value of the current application's A5 register with the low-memory global variable CurrentA5.

SetA5

In interrupt code that accesses application global variables, use the SetA5 function first to restore a value previously saved using SetCurrentA5, and then, at the end of the code, to restore the A5 register to the value it had before the first call to SetA5.

```
FUNCTION SetA5 (newA5: LongInt): LongInt;
```

newA5 The value to which the A5 register is to be changed.

DESCRIPTION

The SetA5 function performs two tasks: it returns the address in the A5 register when the function is called, and it sets the A5 register to the address specified in newA5.

Application-Defined Routines

The techniques illustrated in this chapter use only one application-defined routine, a grow-zone function.

Grow-Zone Functions

The Memory Manager calls your application's grow-zone function whenever it cannot find enough contiguous memory to satisfy a memory allocation request and has exhausted other means of obtaining the space.

MyGrowZone

A grow-zone function should have the following form:

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

`cbNeeded` The physical size, in bytes, of the needed block, including the block header. The grow-zone function should attempt to create a free block of at least this size.

DESCRIPTION

Whenever the Memory Manager has exhausted all available means of creating space within your application heap—including purging, compacting, and (if possible) expanding the heap—it calls your application-defined grow-zone function. The grow-zone function can do whatever is necessary to create free space in the heap. Typically, a grow-zone function marks some unneeded blocks as purgeable or releases an emergency memory reserve maintained by your application.

The grow-zone function should return a nonzero value equal to the number of bytes of memory it has freed, or zero if it is unable to free any. When the function returns a nonzero value, the Memory Manager once again purges and compacts the heap zone 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.

The Memory Manager might designate a particular relocatable block in the heap as protected; your grow-zone function should not move or purge that block. You can determine which block, if any, the Memory Manager has protected by calling the `GZSaveHnd` function in your grow-zone function.

Remember that a grow-zone function is called while the Memory Manager is attempting to allocate memory. As a result, your grow-zone function should not allocate memory itself or perform any other actions that might indirectly cause memory to be allocated (such as calling routines in unloaded code segments or displaying dialog boxes).

You install a grow-zone function by passing its address to the `InitZone` procedure when you create a new heap zone or by calling the `SetGrowZone` procedure at any other time.

SPECIAL CONSIDERATIONS

Your grow-zone function might be called at a time when the system is attempting to allocate memory and the value in the A5 register is not correct. If your function accesses your application's A5 world or makes any trap calls, you need to set up and later restore the A5 register by calling `SetCurrentA5` and `SetA5`.

Because of the optimizations performed by some compilers, the actual work of the grow-zone function and the setting and restoring of the A5 register might have to be placed in separate procedures.

SEE ALSO

See "Defining a Grow-Zone Function" on page 1-48 for a definition of a sample grow-zone function.

Summary of Memory Management

Pascal Summary

Data Types

```
TYPE
   SignedByte           = -128..127;      {arbitrary byte of memory}
   Byte                 = 0..255;         {unsigned, arbitrary byte}
   Ptr                  = ^SignedByte;    {pointer to nonrelocatable block}
   Handle               = ^Ptr;           {handle to relocatable block}

   ProcPtr              = Ptr;            {procedure pointer}

   Size                 = LongInt;        {size, in bytes, of block}
```

Memory Management Routines

Setting Up the Application Heap

```
PROCEDURE MaxApplZone;
PROCEDURE MoreMasters;
FUNCTION GetApplLimit       : Ptr;
PROCEDURE SetApplLimit      (zoneLimit: Ptr);
```

Allocating and Releasing Relocatable Blocks of Memory

```
FUNCTION NewHandle          (logicalSize: Size): Handle;
FUNCTION NewHandleClear     (logicalSize: Size): Handle;
PROCEDURE DisposeHandle     (h: Handle);
```

Allocating and Releasing Nonrelocatable Blocks of Memory

```
FUNCTION NewPtr             (logicalSize: Size): Ptr;
FUNCTION NewPtrClear        (logicalSize: Size): Ptr;
PROCEDURE DisposePtr        (p: Ptr);
```

Setting the Properties of Relocatable Blocks

```
FUNCTION HGetState          (h: Handle): SignedByte;
PROCEDURE HSetState         (h: Handle; flags: SignedByte);
PROCEDURE HLock             (h: Handle);
PROCEDURE HUnlock           (h: Handle);
PROCEDURE HPurge            (h: Handle);
PROCEDURE HNoPurge          (h: Handle);
```

Managing Relocatable Blocks

```
PROCEDURE EmptyHandle       (h: Handle);
PROCEDURE ReallocateHandle  (h: Handle; logicalSize: Size);
PROCEDURE ReserveMem        (cbNeeded: Size);
PROCEDURE MoveHHi           (h: Handle);
PROCEDURE HLockHi           (h: Handle);
```

Manipulating Blocks of Memory

```
PROCEDURE BlockMove         (sourcePtr, destPtr: Ptr; byteCount: Size);
```

Assessing Memory Conditions

```
PROCEDURE PurgeSpace        (VAR total: LongInt; VAR contig: LongInt);
FUNCTION MemError:          : OSErr;
```

Grow-Zone Operations

```
PROCEDURE SetGrowZone       (growZone: ProcPtr);
FUNCTION GZSaveHnd          : Handle;
```

Setting and Restoring the A5 Register

```
FUNCTION SetCurrentA5       : LongInt;
FUNCTION SetA5              (newA5: LongInt) : LongInt;
```

Application-Defined Routines

Grow-Zone Functions

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

C Summary

Data Types

```
typedef char SignedByte;                /*arbitrary byte of memory*/
typedef unsigned char Byte;             /*unsigned, arbitrary byte*/
typedef char *Ptr;                      /*pointer to nonrelocatable block*/
typedef Ptr *Handle;                    /*handle to relocatable block*/

typedef long (*ProcPtr)();              /*procedure pointer*/
typedef long Size;                      /*size in bytes of block*/
```

Memory Management Routines

Setting Up the Application Heap

```
pascal void MaxApplZone     (void);
pascal void MoreMasters     (void);
#define GetApplLimit()      (* (Ptr*) 0x0130)
pascal void SetApplLimit    (void *zoneLimit);
```

Allocating and Releasing Relocatable Blocks of Memory

```
pascal Handle NewHandle     (Size byteCount);
pascal Handle NewHandleClear(Size byteCount);
pascal void DisposeHandle   (Handle h);
```

Allocating and Releasing Nonrelocatable Blocks of Memory

```
pascal Ptr NewPtr           (Size byteCount);
pascal Ptr NewPtrClear      (Size byteCount);
pascal void DisposePtr      (Ptr p);
```

Setting the Properties of Relocatable Blocks

```
pascal char HGetState       (Handle h);
pascal void HSetState       (Handle h, char flags);
pascal void HLock           (Handle h);
pascal void HUnlock         (Handle h);
pascal void HPurge          (Handle h);
pascal void HNoPurge        (Handle h);
```

Managing Relocatable Blocks

```
pascal void EmptyHandle     (Handle h);
pascal void ReallocateHandle(Handle h, Size byteCount);
pascal void ReserveMem      (Size cbNeeded);
pascal void MoveHHi         (Handle h);
pascal void HLockHi         (Handle h);
```

Manipulating Blocks of Memory

```
pascal void BlockMove       (const void *srcPtr, void *destPtr,
                             Size byteCount);
```

Assessing Memory Conditions

```
pascal void PurgeSpace      (long *total, long *contig);
#define MemError()          (* (OSErr*) 0x0220)
```

Grow-Zone Operations

```
pascal void SetGrowZone     (GrowZoneProcPtr growZone);
#define GZSaveHnd()         (* (Handle*) 0x0328)
```

Setting and Restoring the A5 Register

```
long SetCurrentA5           (void);
long SetA5                  (long newA5);
```

Application-Defined Routines

Grow-Zone Functions

```
pascal long MyGrowZone      (Size cbNeeded);
```

Assembly-Language Summary

Global Variables

`ApplLimit`	long	The application heap limit, beyond which the heap cannot expand.
`ApplZone`	long	A pointer to the original application heap zone.
`BufPtr`	long	Address of highest byte of allocatable memory.
`CurrentA5`	long	Address of the boundary between the application global variables and the application parameters of the current application.
`GZRootHnd`	long	A handle to a block that the grow-zone function must not move.

Result Codes

`noErr`	0	No error
`paramErr`	–50	Error in parameter list
`memROZErr`	–99	Heap zone is read-only
`memFullErr`	–108	Not enough memory
`nilHandleErr`	–109	`NIL` master pointer
`memWZErr`	–111	Attempt to operate on a free block
`memPurErr`	–112	Attempt to purge a locked block
`memBCErr`	–115	Block check failed
`memLockedErr`	–117	Block is locked