This chapter describes how your application can use the Memory Manager to manage memory both in its own partition and outside its partition. Ordinarily, you allocate memory in your application heap only. You might, however, occasionally need to access memory outside of your application partition, or you might want to create additional heap zones within your application partition.

You need to read this chapter if you want to use Memory Manager routines other than those described in the chapter "Introduction to Memory Management" in this book. That chapter shows how to use the Memory Manager and other system software components to perform the most common memory-manipulation operations while avoiding heap fragmentation and low memory situations. This chapter addresses a number of other important memory-related issues.

This chapter begins with a description of areas of memory that are outside your application's partition and their typical uses. Then it describes how you can

- allocate temporary memory
- allocate memory in and install code into the system heap
- read and change the values of system global variables
- allocate high memory during the startup process
- create additional heap zones within your application's partition
- install a purge-warning procedure for a heap zone

This chapter also addresses some advanced topics that are generally of use only to developers of very specialized applications or memory utilities. These advanced topics include

- how the Memory Manager organizes heap zones
- how the Memory Manager organizes memory blocks

To use this chapter, you should be familiar with ordinary use of the Memory Manager and other system software components that allow you to manage memory, as described in the chapter "Introduction to Memory Management" earlier in this book.

The "Memory Manager Reference" and "Summary of the Memory Manager" sections in this chapter provide a complete reference and summary of the constants, data types, and routines provided by the Memory Manager.

About the Memory Manager

The Memory Manager is the part of the Macintosh Operating System that controls the dynamic **allocation** of memory space. Ordinarily, you need to access information only within your own application's heap, stack, and A5 world. Occasionally, however, you might need to use the Memory Manager to allocate temporary memory outside of your application's partition or to initialize new heap zones within your application partition. You might also need to read a system global variable to obtain information about the environment in which your application is executing.

The Memory Manager provides a large number of routines that you can use to perform various operations on blocks within your application partition. You can use the Memory Manager to

- set up your application partition
- allocate and release both relocatable and nonrelocatable blocks in your application heap
- copy data from nonrelocatable blocks to relocatable blocks, and vice versa
- determine how much space is free in your heap
- determine the location of the top of your stack
- determine the size of a memory block and, if necessary, change that size
- change the properties of relocatable blocks
- install or remove a grow-zone function for your heap
- obtain the result code of the most recent Memory Manager routine executed

The Memory Manager also provides routines that you can use to access areas of memory outside your application partition. You can use the Memory Manager to

- allocate memory outside your partition that is currently unused by any open application or by the Operating System
- allocate memory in the system heap

This section describes the areas of memory that lie outside your application partition. It also describes multiple heap zones.

Temporary Memory

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

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

Your application should use temporary memory only for occasional short-term purposes that could be accomplished in less space, though perhaps less efficiently. 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 use the large buffer to copy data from the source file into the destination file. If, however, the request for temporary memory fails, you can instead use a smaller buffer within your application heap. Although the use of a smaller buffer might prolong the copy 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, if two or more applications use all available memory outside the system partition, then a request by any of them for some temporary memory would fail.

Another strategy for using temporary memory is to use it, when possible, for all nonessential memory requests. For example, you could allocate window records and any associated window data using temporary memory. This scheme allows you to keep your application partition relatively small (because you don't need space for nonessential tasks) but assumes that users will not fill up the available memory with other applications.

Multiple Heap Zones

A **heap zone** is a heap (that is, an area in which you can dynamically allocate and release memory on demand) together with a zone header and a zone trailer. The **zone header** is an area of memory that contains essential information about the heap, such as the number of free bytes in the heap and the addresses of the heap's grow-zone function and purge-warning procedure. The **zone trailer** is just a minimum-sized block placed as a marker at the end of the heap zone. (See "Heap Zones" on page 2-19 for a complete description of zone headers and trailers.)

When your application is executing, there exist at least two heap zones: your application's heap zone (created when your application was launched) and the system heap zone (created when the system was started up). The **system heap zone** is the heap zone that contains the system heap. Your **application heap zone** (also known as the **original application heap zone**) is the heap zone initially provided by the Memory Manager for use by your application and any system software routines your application calls.

Ordinarily, you allocate and release blocks of memory in the **current heap zone,** which by default is your application heap zone. Unless you change the current heap zone (for example, by calling the `InitZone` or `SetZone` procedures), you do not need to worry about which is the current zone; all blocks that you access are taken from the current heap zone, that is, your application heap zone.

Occasionally, however, you might need to allocate memory in the system heap zone. System software uses the system heap to store information it needs. Although, in general, you should not allocate memory in the system heap, there are several valid reasons for doing so. First, if you are implementing a system extension, the extension can use the system heap to store information. Second, if you want the Time Manager or Vertical Retrace Manager to execute some interrupt code when your application is not the current application, you might in certain cases need to store the task record and the task code in the system heap. Third, if you write interrupt code that itself uses heap memory, you should either place that memory in the system heap or hold it in real RAM to prevent page faults at interrupt time, as discussed in the chapter "Virtual Memory Manager" in this book.

You can create additional heap zones for your application's own use by calling the `InitZone` procedure. If you do maintain more than one heap zone, you can find out which heap zone is the current one at any time by calling the `GetZone` function, and you can switch zones by calling the `SetZone` procedure. Almost all Memory Manager operations implicitly apply to the current heap zone. To refer to the system heap zone or to the (original) application heap zone, you can call the functions `SystemZone` or `ApplicationZone`. To find out which zone a particular block resides in, you can call the `HandleZone` function (if the block is relocatable) or the `PtrZone` function (if it's nonrelocatable).

▲ **WARNING**
Be sure, when calling routines that access blocks, that the zone in which the block is located is the current zone. If, for example, you attempt to release an empty resource in the system zone when the current zone is not the system zone, the Operating System might incorrectly update the list of free master pointers in your partition. ▲

Once you have created a heap zone, it remains fixed in size and location. For this reason, it usually makes more sense to use the undivided application heap zone for all of your memory-allocation needs. You might, however, choose to initialize an additional heap zone in circumstances like these:

- If you are implementing a software development environment and want to launch applications within the development environment's partition, you can initialize a heap zone for the launched application to use as its heap zone.
- If you want to avoid heap fragmentation but cannot prevent allocation of small nonrelocatable blocks in the middle of your program's execution, you could, soon after your application starts up, allocate a small heap zone to hold the nonrelocatable blocks you allocate during execution.
- If you need to resize a particular handle quite often, you can minimize the resizing time by creating a heap zone whose size is set to the maximum size the handle will ever be assigned. Because there is only one relocatable block in the new heap zone, the resizing is likely to happen more quickly than if that block were in the original heap zone (where other relocatable blocks in the zone might need to be moved).

Before deciding to create additional heap zones, however, make sure that you really need to. Maintaining multiple heap zones requires a considerable amount of extra work. You must always make sure to allocate or release memory in the correct zone, and you must monitor memory conditions in each zone so that your application doesn't run out of memory.

The System Global Variables

Just as the Toolbox stores information about your drawing environment in a set of QuickDraw global variables within your application partition, the Operating System and Toolbox store information about the entire multiple-application environment in a set of **system global variables,** also called low-memory global variables. The system global variables are stored in the lowest part of the physical RAM, in the system partition.

Most system global variables are intended for use by system software only, and you should never need to read or write them directly. Current versions of system software contain functions that return values equivalent to most of the important system global variables. Use those routines whenever they are available. However, you might occasionally need to access the value of a system global variable to maintain compatibility with previous versions of system software, or you might need to access a system global variable whose value no equivalent function returns.

The MPW interface file `SysEqu.p` defines the memory locations at which system global variables are stored in the latest version of system software. For example, `SysEqu.p` contains lines like these:

```
CONST
   RndSeed        = $156;  {random number seed (long)}
   Ticks          = $16A;  {ticks since last boot (unsigned long)}
   DeskHook       = $A6C;  {hook for painting desktop (pointer)}
   MBarHeight     = $BAA;  {height of menu bar (integer)}
```

You can use these memory locations to examine the value of one of these variables. See "Reading and Writing System Global Variables" on page 2-8 for instructions on reading and writing the values of system global variables from a high-level language.

You should avoid relying on the value of a system global variable whenever possible. The meanings of many global variables have changed in the past and will change again in the future. Using the system global variables documented in *Inside Macintosh* is fairly safe, but you risk incompatibility with future versions of system software if you attempt to access global variables defined in the interface files but not explicitly documented.

Even when *Inside Macintosh* does document a particular system global variable, you should use any available routines to access that variable's value instead of examining it directly. For example, you should use the `TickCount` function to find the number of ticks since startup instead of examining the `Ticks` global variable directly.

IMPORTANT
You should read or write the value of a system global variable only when that variable is documented in *Inside Macintosh* and when there is no alternate method of reading or writing the information you need. ▲

Using the Memory Manager

This section discusses the techniques you can use both to deal with memory outside of your application's partition and to manipulate your own application's partition.

You can use the techniques in this section to

- read and write the values of system global variables when there is no Toolbox routine that would accomplish the work for you

- check for the availability of temporary memory and use it to speed operations that depend on memory buffers
- allocate memory in the system heap
- install code into the system heap
- allocate memory at the high end of the available RAM from within a system extension during the startup process
- initialize new heap zones within your application heap zone, on your application's stack, or in the application global variables area
- install a purge-warning procedure for your application heap zone

Reading and Writing System Global Variables

In general, you should avoid relying on the values of system global variables whenever possible. However, you might occasionally need to access the value of one of these variables. Because the actual values associated with global variables in MPW's `SysEqu.p` interface file are memory locations, you can access the value of a low-memory variable simply by dereferencing a memory location.

Many system global variables are process-independent, but some are process-specific. The Operating System swaps the values of the process-specific variables as it switches processes. If you write interrupt code that reads low memory, that code could execute at a time when another process's system global variables are installed. Therefore, before reading low memory from interrupt code, you should call the Process Manager to ensure that your process is the current process. If it is not, you should not rely on the value of system global variables that could conceivably be process-specific.

Note
No available documentation distinguishes process-specific from process-independent system global variables. ◆

The routine defined in Listing 2-1 illustrates how you can read a system global variable, in this case the system global variable `BufPtr`, which gives the address of the highest byte of allocatable memory.

Listing 2-1 Reading the value of a system global variable

```
FUNCTION FindHighestByte: LongInt;
TYPE
   LongPtr = ^LongInt;
BEGIN
   FindHighestByte := LongPtr(BufPtr)^;
END;
```

In Pascal, the main technique for reading system global variables is to define a new data type that points to the variable type you want to read. In this example, the address is

stored as a long integer. Thus, the memory location `BufPtr` is really a pointer to a long integer. Because of Pascal's strict typing rules, you must cast the low-memory address into a pointer to a long integer. Then, you can dereference the pointer and return the long integer itself as the function result.

You can use a similar technique to change the value of a system global variable. For example, suppose you are writing an extension that displays a window at startup time. To maintain compatibility with pre-Macintosh II systems, you need to clear the system global variable named `DeskHook`. This global variable holds a `ProcPtr` that references a procedure called by system software to paint the desktop. If the value of the pointer is `NIL`, the system software uses the standard desktop pattern. If you do not set `DeskHook` to `NIL`, the system software might attempt to use whatever random data it contains to call an updating procedure when you move or close your window. The procedure defined in Listing 2-2 sets `DeskHook` to `NIL`.

Listing 2-2 Changing the value of a system global variable

```
PROCEDURE ClearDeskHook;
TYPE
   ProcPtrPtr = ^ProcPtr;                {pointer to ProcPtr}
VAR
   deskHookProc: ProcPtrPtr;
BEGIN
   deskHookProc := ProcPtrPtr(DeskHook);  {initialize variable}
   deskHookProc^ := NIL;                  {clear DeskHook proc}
END;
```

You can use a similar technique to change the value of any other documented system global variable.

Extending an Application's Memory

Rather than using your application's `'SIZE'` resource to specify a preferred partition size that is large enough to contain the largest possible application heap, you should specify a smaller but adequate partition size. When you need more memory for temporary use, you can use a set of Memory Manager routines for the allocation of temporary memory.

By using the routines for allocating temporary memory, your application can request some additional memory for occasional short-term needs. For example, the Finder uses these temporary-memory routines to secure buffer space for use during file copy operations. Any available memory (that is, memory currently unallocated to any application's partition) is dedicated to this purpose. The Finder releases this memory as soon as the copy is completed, thus making the memory available to other applications or to the Operating System for launching new applications.

Because the requested amount of memory might not be available, you cannot be sure that every request for temporary memory will be honored. Thus, you should make sure that your application will work even if your request for temporary memory is denied. For example, if the Finder cannot allocate a large temporary copy buffer, it uses a reserved small copy buffer from within its own heap zone, prolonging the copying but performing it nonetheless.

Temporary memory is taken from RAM that is reserved for (but not yet used by) other applications. Thus, if you use too much temporary memory or hold temporary memory for long periods of time, you might prevent the user from being able to launch other applications. In certain circumstances, however, you can hold temporary memory indefinitely. For example, if the temporary memory is used for open files and the user can free that memory simply by closing those files, it is safe to hold onto that memory as long as necessary.

Temporary memory is tracked (or monitored) for each application, and so you must use it only for code that is running on an application's behalf. Moreover, the Operating System frees all temporary memory allocated to an application when the application quits or crashes. As a result, you should not use temporary memory for VBL tasks, Time Manager tasks, or other procedures that should continue to be executed after your application quits. Similarly, it is wise not to use temporary memory for an interprocess buffer (that is, a buffer whose address is passed to another application in a high-level event) because the originating application could crash, quit, or be terminated, thereby causing the temporary memory to be released before (or even while) the receiving application uses that memory.

Although you can usually perform ordinary Memory Manager operations on temporary memory, there are two restrictions. First, you must never lock temporary memory across calls to `GetNextEvent` or `WaitNextEvent`. Second, although you can determine the zone from which temporary memory is generated (using the `HandleZone` function), you should not use this information to make new blocks or perform heap operations on your own.

Allocating Temporary Memory

You can request a block of memory for temporary use by calling the Memory Manager's `TempNewHandle` function. This function attempts to allocate a new relocatable block of the specified size for temporary use. For example, to request a block that is one-quarter megabyte in size, you might issue this command:

```
myHandle := TempNewHandle($40000, myErr); {request temp memory}
```

If the routine succeeds, it returns a handle to the block of memory. The block of memory returned by a successful call to `TempNewHandle` is initially unlocked. If an error occurs and `TempNewHandle` fails, it returns a `NIL` handle. You should always check for `NIL` handles before using any temporary memory. If you detect a `NIL` handle, the second parameter (in this example, `myErr`) contains the result code from the function.

Instead of asking for a specific amount of memory and then checking the returned handle to find out whether it was allocated, you might prefer to determine beforehand how much temporary memory is available. There are two functions that return information on the amount of free memory available for temporary allocation. The first is the `TempFreeMem` function, which you can use as follows:

```
memFree := TempFreeMem;    {find amount of free temporary memory}
```

The result is a long integer containing the amount, in bytes, of free memory available for temporary allocation. It usually isn't possible to allocate a block of this size because of fragmentation. Consequently, you'll probably want to use the second function, `TempMaxMem`, to determine the size of the largest contiguous block of space available. To allocate that block, you can write

```
mySize := TempMaxMem(grow);
myHandle := TempNewHandle(mySize, myErr);
```

The `TempMaxMem` function returns the size, in bytes, of the largest contiguous free block available for temporary allocation. (The `TempMaxMem` function is analogous to the `MaxMem` function.) The `grow` parameter is a variable parameter of type `Size`; after the function returns, it always contains 0, because the temporary memory does not come from the application's heap. Even when you use `TempMaxMem` to determine the size of the available memory, you should check that the handle returned by `TempNewHandle` is not `NIL`.

Determining the Features of Temporary Memory

Only computers running system software version 7.0 and later can use temporary memory as described in this chapter. For this reason, you should always check that the routines are available and that they have the features you require before calling them.

Note
The temporary-memory routines are available in some earlier system software versions when MultiFinder is running. However, the handles to blocks of temporary memory are neither tracked nor real. ◆

The `Gestalt` function includes a selector to determine whether the temporary-memory routines are present in the operating environment and, if they are, whether the temporary-memory handles are tracked and whether they are real. If temporary-memory handles are not tracked, you must release temporary memory before your next call to `GetNextEvent` or `WaitNextEvent`. If temporary-memory handles are not real, then you cannot use normal Memory Manager routines such as `HLock` to manipulate them.

To determine whether the temporary-memory routines are implemented, you can check the value returned by the `TempMemCallsAvailable` function, defined in Listing 2-3.

Listing 2-3 Determining whether temporary-memory routines are available

```
FUNCTION TempMemCallsAvailable: Boolean;
VAR
   myErr:   OSErr;              {Gestalt result code}
   myRsp:   LongInt;            {response returned by Gestalt}
BEGIN
   TempMemCallsAvailable := FALSE;
   myErr := Gestalt(gestaltOSAttr, myRsp);
   IF myErr <> noErr THEN
      DoError(myErr)          {Gestalt failed}
   ELSE                       {check bit for temp mem support}
      TempMemCallsAvailable :=
            BAND(myRsp, gestaltTempMemSupport) <> 0;
END;
```

You can use similar code to determine whether temporary-memory handles are real and whether the temporary memory is tracked.

Using the System Heap

The system heap is used to store most of the information needed by the Operating System and other system software components. As a result, it is ideal for storing information needed by a system extension (which by definition extends the capabilities of system software). You might also need to use the system heap to store a task record and the code for an interrupt task that should continue to be executed when your application is not the current application.

Allocating blocks in the system heap is straightforward. Most ordinary Memory Manager routines have counterparts that allocate memory in the system heap zone instead of the current heap zone. For example, the counterpart of the `NewPtr` function is the `NewPtrSys` function. The following line of code allocates a new nonrelocatable block of memory in the system heap to store a Time Manager task record:

```
myTaskPtr := QElemPtr(NewPtrSys(SizeOf(TMTask)));
```

Alternatively, you can change the current zone and use ordinary Memory Manager operations, as follows:

```
SetZone(SystemZone);
myTaskPtr := QElemPtr(NewPtr(SizeOf(TMTask)));
...
SetZone(ApplicationZone);
```

You might also need to store the interrupt code itself in the system heap. For example, when an application that installed a vertical retrace task with the `VInstall` function is in the background, the Vertical Retrace Manager executes the task only if the `vblAddr` field of the task record points to a routine in the system heap.

Unfortunately, manually copying a routine into the system heap is difficult in Pascal. The easiest way to install code into the system heap is to place the code into a separate stand-alone code resource in your application's resource fork. You should set the system heap bit and the locked bit of the code resource's attributes. Then, when you need to use the code, you must load the resource from the resource file and cast the resource handle's master pointer into a procedure pointer (a variable of type `ProcPtr`), as follows:

```
myProcHandle := GetResource(kProcType, kProcID);
IF myProcHandle <> NIL THEN
   myTaskPtr^.vblAddr := ProcPtr(myProcHandle^);
```

Because the resource is locked in memory, you don't have to worry about creating a dangling pointer when you dereference a handle to the resource. If you want the code to remain in the system heap after the user quits your application, you can call the Resource Manager procedure `DetachResource` so that closing your application's resource fork does not destroy the resource data. Note, however, that if you do so and your application crashes, the code still remains in the system heap.

Once you have loaded a code resource into memory and created a `ProcPtr` that references the entry point of the code resource, you can use that `ProcPtr` just as you can use any such variable. For example, you could assign the value of the variable to the `vblAddr` field of a vertical retrace task record (as shown just above). If you are programming in assembly language, you can then call the code directly. To call the routine from a high-level language such as Pascal, you'll need to use some inline assembly-language code. Listing 2-4 defines a routine that you can use to execute a procedure by address.

Listing 2-4 Calling a procedure by address

```
PROCEDURE CallByAddress (aRoutine: ProcPtr);
   INLINE   $205F,            {MOVE.L (SP)+,A0}
            $4ED0;            {JMP (A0)}
```

Allocating Memory at Startup Time

If you are implementing a system extension, you might need to allocate memory at startup time. As explained in the previous section, an ideal place to allocate such memory is in the system heap. To allocate memory in the system heap under system software version 7.0 and later, you merely need to call the appropriate Memory Manager routines, and the system heap expands dynamically to meet your request. In earlier versions of system software, you must use a `'sysz'` resource to indicate how much the Operating System should increase the size of the system zone.

Alternatively, however, you can allocate blocks in high memory. The global variable `BufPtr` always references the highest byte in memory that might become part of an application partition. You can lower the value of `BufPtr` and then use the memory between the old and new values of `BufPtr`.

Note
In general, if you are implementing a system extension, you should allocate memory in the system heap instead of high memory. In this way, you avoid the problems associated with lowering the value of `BufPtr` too far (described in the following paragraphs) and ensure that the extension is not paged out if virtual memory is operating. ◆

Lowering the value of `BufPtr` too far can be dangerous for several reasons. In 128K ROM Macintosh computers running system software version 4.1, you must avoid lowering the value of `BufPtr` so that it points in the system startup blocks. The highest byte of these blocks can always be found relative to the global variable `MemTop`, at `MemTop DIV 2 + 1024`.

In later versions of the Macintosh system software, the system startup blocks were no longer barriers to `BufPtr`, but new barriers arose, including Macintosh IIci video storage, for example. To maintain compatibility with extensions that rely on the ability to lower `BufPtr` relative to `MemTop`, the system software simply adjusts `MemTop` so that the formula still holds. Thus, at startup, the `MemTop` global variable currently does not reference any memory location in particular. Instead, it holds a value that guarantees that the formula allowing you to lower `BufPtr` as low as `MemTop DIV 2 + 1024` but no further still holds.

Beginning in system software version 7.0, the Operating System can detect excessive lowering of `BufPtr`, but only after the fact. When the Operating System does detect that the value of `BufPtr` has fallen too low, it generates an out-of-memory system error.

▲ **WARNING**
Although the above formula has been true since system software version 4.1, a bug in the Macintosh IIci and later ROMs made it invalid in certain versions of system software 6.x. ▲

Because there is no calling interface for lowering `BufPtr`, you must do it manually, by changing the value of the system variable, as explained in "Reading and Writing System Global Variables" on page 2-8. To obtain the value of the `MemTop` global variable, you can use the `TopMem` function.

Creating Heap Zones

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

Note
Most applications do not need to create heap zones. ◆

To create a new heap zone in the application heap, you must allocate nonrelocatable blocks in your application heap to hold new subzones of the application heap. In addition to being able to create subzones of the application zone, you can create subzones of any other zone to which you have access, including a zone that is itself a subzone of another zone.

You create a heap zone by calling the `InitZone` procedure, which takes four parameters. The first parameter specifies a grow-zone function for the new zone, or `NIL` if you do not want the zone to have a grow-zone function. The second parameter specifies the number of new master pointers that you want each block of master pointers in the zone to contain. The `InitZone` procedure allocates one such block to start with, and you can allocate more by calling the `MoreMasters` procedure. The third and fourth parameters specify, respectively, the first byte beyond the end of the new zone and the first byte of the zone.

When initializing a zone with the `InitZone` procedure, make sure that you are subdividing the current zone. When `InitZone` returns, the new zone becomes current. Thus, if you subdivide the application zone into several subzones, you must call `SetZone(ApplicationZone)` before you create the second and each of the subsequent subzones. Listing 2-5 shows a technique for creating a single subzone of the original application zone, assuming that the application zone is the current zone. The technique for subdividing subzones is similar.

Listing 2-5 Creating a subzone of the original application heap zone

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

To create a subzone in the system heap zone, you can call `SetZone(SystemZone)` at the beginning of the procedure in Listing 2-5. You might find this technique useful if you are implementing a system extension but want to manage your extension's memory much as you manage memory in an application. Instead of simply allocating blocks in the system heap, you can make your zone current whenever your extension is executed. Then, you can call regular Memory Manager routines to allocate memory in your subzone of the system heap, and you can compact and purge your subzone without compacting and purging the entire system heap zone.

When you allocate memory for a subzone, you must allocate that memory in a nonrelocatable block (as in Listing 2-5) or in a locked relocatable block. If you create a subzone within an unlocked relocatable block, the Memory Manager might move your entire subzone during memory operations in the zone containing your subzone. If so, any references to nonrelocatable blocks that you allocated in the subzone would become invalid. Even handles to relocatable blocks in the subzone would no longer be valid, because the Memory Manager does not update the handles' master pointers correctly. This happens because the Memory Manager views a subzone of another zone as a single block. If that subzone is a relocatable block, the Memory Manager updates only that block's master pointer when moving it, and does not update the block's contents (that is, the blocks allocated within the subzone).

If you use a block of temporary memory as a heap zone, you must lock the temporary memory immediately after allocating it. Then, you can pass to `InitZone` a dereferenced copy of a handle to the temporary memory. If you find (after a call to the `Gestalt` function) that temporary memory handles are not real, then you must dispose of the new zone before any calls to `GetNextEvent` or `WaitNextEvent`. You must dispose of the new zone because you cannot lock a handle to temporary memory across event calls if the handle is not real.

Once you have created a subzone as a nonrelocatable block or a locked relocatable block, you can allocate both relocatable and nonrelocatable blocks within it. Although the Memory Manager can move such relocatable blocks only within the subzone, it correctly updates those blocks' master pointers, which are also in the subzone.

Installing a Purge-Warning Procedure

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

Note
Most applications don't need to install a purge-warning procedure. This capability is provided primarily for applications that require greater control over their heap. Examples are applications that maintain purgeable handles containing important data and applications that for any other reason need notification when a block is about to be purged. ◆

When your purge-warning procedure is called, the Memory Manager passes it a handle to the block about to be purged. In your procedure, you can test the handle to determine whether it contains data that needs to be saved; if so, you can save the data (possibly by writing it to some open file). Listing 2-6 defines a very simple purge-warning procedure.

Listing 2-6 A purge-warning procedure

```
PROCEDURE MyPurgeProc (h: Handle);
VAR
   theA5:   LongInt;             {value of A5 when procedure is called}
BEGIN
   theA5 := SetCurrentA5;        {remember current value of A5; install ours}
   IF BAND(HGetState(h), $20) = 0 THEN
      BEGIN                      {if the handle isn't a resource handle}
         IF InSaveList(h) THEN
            WriteData(h);        {save the data in the block}
      END;
   theA5 := SetA5(theA5);        {restore previous value of A5}
END;
```

The `MyPurgeProc` procedure defined in Listing 2-6 inspects the handle's properties (using `HGetState`) to see whether its resource bit is clear. If so, the procedure next determines whether the handle is contained in an application-maintained list of handles whose data should be saved before purging. If the handle is in that list, the purge-warning procedure writes its data to disk. (The file into which the data is written should already be open at the time the procedure is called, because opening a file might cause memory to move.)

Note that `MyPurgeProc` sets up the A5 register with the application's A5 value upon entry and restores it to its previous value before exiting. This is necessary because you cannot rely on the A5 register within a purge-warning procedure.

▲ **WARNING**
Because of the optimizations performed by some compilers, the actual work of the purge-warning procedure and the setting and restoring of the A5 register might have to be placed in separate procedures. See the chapter "Vertical Retrace Manager" in *Inside Macintosh: Processes* for an illustration of how you can do this. ▲

To install a purge-warning procedure, you need to install the address of the procedure into the `purgeProc` field of your application's heap zone header. Listing 2-7 illustrates one way to do this.

Listing 2-7 Installing a purge-warning procedure

```
PROCEDURE InstallPurgeProc;
VAR
   myZone:  THz;
BEGIN
   myZone := GetZone;                      {find the current zone header}
   gPrevProc := myZone^.purgeProc;         {remember previous procedure}
   myZone^.purgeProc := @MyPurgeProc;      {install new procedure}
END;
```

The `InstallPurgeProc` procedure defined in Listing 2-7 first obtains the address of the current heap zone by calling the `GetZone` function. Then it saves the address of any existing purge-warning procedure in the global variable `gPrevProc`. Finally, `InstallPurgeProc` installs the new procedure by putting its address directly into the `purgeProc` field of the zone header. (For more information on zone headers, see "Heap Zones" on page 2-19.)

Keep in mind that the Memory Manager calls your purge-warning procedure each time it decides to purge any purgeable block, and it might call your procedure far more often than you would expect. Your purge-warning procedure might be passed handles not only to blocks that you explicitly mark as purgeable (by calling `HPurge`), but also to resources whose purgeable attribute is set. (In general, applications don't need to take any action on handles that belong to the Resource Manager.) Because of the potentially large number of times your purge-warning procedure might be called, it should be able to determine quickly whether a handle that is about to be purged needs additional processing.

Remember that a purge-warning procedure is called during the execution of some Memory Manager routine. As a result, your procedure cannot cause memory to be moved or purged. In addition, it should not dispose of the handle it is passed or change the purge status of the handle. See "Purge-Warning Procedures" on page 2-90 for a complete description of the limitations on purge-warning procedures.

▲ **WARNING**

If your application calls the Resource Manager procedure `SetResPurge` with the parameter `TRUE` (to have the Resource Manager automatically save any modified resources that are about to be purged), you should avoid using a purge-warning procedure. This is because the Resource Manager installs its own purge-warning procedure when you call `SetResPurge` in this way. If you must install your own purge-warning procedure, you should remove your procedure, call `SetResPurge`, then reinstall your procedure as shown in Listing 2-7. You then need to make sure that your procedure calls the Resource Manager's purge-warning procedure (which is saved in the global variable `gPrevProc`) before exiting. Most applications do not need to call `SetResPurge` at all. ▲

If your application does call `SetResPurge(TRUE)`, you should use the version of `MyPurgeProc` defined in Listing 2-8. It is just like the version defined in Listing 2-6 except that it calls the Resource Manager's purge-warning procedure before exiting.

Listing 2-8 A purge-warning procedure that calls the Resource Manager's procedure

```
PROCEDURE MyPurgeProc (h: Handle);
VAR
   theA5:   LongInt;            {value of A5 when procedure is called}
BEGIN
   theA5 := SetCurrentA5;       {remember current value of A5; install ours}
   IF BAND(HGetState(h), $20) = 0 THEN
      BEGIN                     {if the handle isn't a resource handle}
         IF InSaveList(h) THEN
            WriteData(h);       {save the data in the block}
      END
   ELSE IF gPrevProc <> NIL THEN
      CallByAddress(gPrevProc);
   theA5 := SetA5(theA5);       {restore previous value of A5}
END;
```

See Listing 2-4 on page 2-13 for a definition of the procedure `CallByAddress`.

Organization of Memory

This section describes the organization of heap zones and block headers. In general, you do not need to know how the Memory Manager organizes heap zones or block headers if your application simply allocates and releases blocks of memory. The information described in this section is used by the Memory Manager for its own purposes. Developers of some specialized applications and utilities might, however, need to know exactly how zones and block headers are organized. This information is also sometimes useful for debugging.

▲ **WARNING**
This section is provided primarily for informational purposes. The organization and size of heap zones and block headers is subject to change in future system software versions. ▲

Heap Zones

Except for temporary memory blocks, all relocatable and nonrelocatable blocks exist within heap zones. A heap zone consists of a zone header, a zone trailer block, and usable bytes in between. The header contains all of the information the

Memory Manager needs about that heap zone; the trailer is just a minimum-sized free block placed as a marker at the end of the zone.

In Pascal, a heap zone is defined as a **zone record** of type `Zone`. The zone record contains all of the fields of the zone header. A heap zone is always referred to with a **zone pointer** of data type `THz`.

▲ **WARNING**
The fields of the zone header are for the Memory Manager's own internal use. You can examine the contents of the zone's fields, but in general it doesn't make sense for your application to try to change them. The only fields of the zone record that you can safely modify directly are the `moreMast` and `purgeProc` fields. ▲

```
TYPE Zone =
RECORD
   bkLim:         Ptr;          {first usable byte after zone}
   purgePtr:      Ptr;          {used internally}
   hFstFree:      Ptr;          {first free master pointer}
   zcbFree:       LongInt;      {number of free bytes in zone}
   gzProc:        ProcPtr;      {grow-zone function}
   moreMast:      Integer;      {num. of master ptrs to allocate}
   flags:         Integer;      {used internally}
   cntRel:        Integer;      {reserved}
   maxRel:        Integer;      {reserved}
   cntNRel:       Integer;      {reserved}
   maxNRel:       Integer;      {reserved}
   cntEmpty:      Integer;      {reserved}
   cntHandles:    Integer;      {reserved}
   minCBFree:     LongInt;      {reserved}
   purgeProc:     ProcPtr;      {purge-warning procedure}
   sparePtr:      Ptr;          {used internally}
   allocPtr:      Ptr;          {used internally}
   heapData:      Integer;      {first usable byte in zone}
END;
THz = ^Zone;                    {zone pointer}
```

Field descriptions

`bkLim` A pointer to the byte *following* the last byte of usable space in the zone.

`purgePtr` Used internally.

`hFstFree` A pointer to the first free master pointer in the zone. All master pointers that are allocated but not currently in use are linked together into a list. The `hFstFree` field references the head node of this list. The Memory Manager updates this list every time it allocates a new relocatable block or releases one, so that the list contains all unused master pointers. If the Memory Manager needs

	a new master pointer but this field is set to `NIL`, it allocates a new nonrelocatable block of master pointers. You can check the value of this field to see whether allocating a relocatable block would cause a new block of master pointers to be allocated.
`zcbFree`	The number of free bytes remaining in the zone. As blocks are allocated and released, the Memory Manager adjusts this field accordingly. You can use the `FreeMem` function to determine the value of this field for the current heap zone.
`gzProc`	A pointer to a grow-zone function that system software uses to maintain control over the heap. The system's grow-zone function subsequently calls the grow-zone function you specify for your heap, if any. You can change a heap zone's grow-zone function at any time but should do so only by calling the `InitZone` or `SetGrowZone` procedures. Note that in current versions of system software, this field does not contain a pointer to the grow-zone function that your application defines.
`moreMast`	The number of master pointers the Memory Manager should allocate at a time. The Memory Manager allocates this many automatically when a heap zone is initialized. By default, master pointers are allocated 32 at a time for the system heap zone and 64 at a time for the application heap zone, but this might change in future versions of system software.
`flags`	Used internally.
`cntRel`	Reserved.
`maxRel`	Reserved.
`cntNRel`	Reserved.
`maxNRel`	Reserved.
`cntEmpty`	Reserved.
`cntHandles`	Reserved.
`minCBFree`	Reserved.
`purgeProc`	A pointer to the zone's purge-warning procedure, or `NIL` if there is none. The Memory Manager calls this procedure before it purges a block from the zone. Note that whenever you call the Resource Manager procedure `SetResPurge` with the parameter set to `TRUE`, the Resource Manager installs its own purge-warning procedure, overriding any purge-warning procedure you have specified here.
`sparePtr`	Used internally.
`allocPtr`	Used internally.
`heapData`	A dummy field marking the beginning of the zone's usable memory space. The integer in this field has no significance in itself; it is just the first 2 bytes in the block header of the first block in the zone. For example, if `myZone` is a zone pointer, then `@(myZone^.heapData)` is the address of the first usable byte in the zone, and `myZone^.bkLim` is a pointer to the byte following the last usable byte in the zone.

The structure of a heap zone is the same in both 24-bit and 32-bit addressing modes. The use of several of the fields that are reserved or used internally, however, may differ in 24-bit and 32-bit heap zones.

Block Headers

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

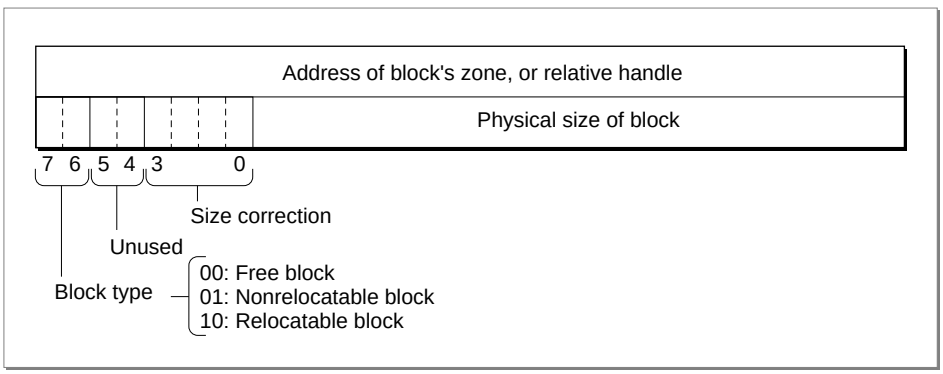
There are two reasons that a block might contain such unused bytes:

- The Memory Manager allocates space only in even numbers of bytes. (This practice guarantees that both the contents and the address of a master pointer are even.) If a block's logical size is odd, an extra, unused byte is added at the end to make the physical size an even number. On computers containing the MC68020, MC68030, or MC68040 microprocessor, blocks are padded to 4-byte boundaries.
- The minimum number of bytes in a block is 12. This minimum applies to all blocks, free as well as allocated. If allocating the required number of bytes from a free block would leave a fragment of fewer than 12 free bytes, the leftover bytes are included unused at the end of the newly allocated block instead of being returned to free storage.

There is no Pascal record type defining the structure of block headers because you shouldn't normally need to access them directly. In addition, the structure of a block header depends on whether the block is located in a 24-bit or 32-bit zone.

In a 24-bit zone, a block header consists of 8 bytes, which together make up two long words, as shown in Figure 2-1.

Figure 2-1 A block header in a 24-bit zone



In the first long word, the low-order 3 bytes contain the block's physical size in bytes. Adding this number to the block's address gives the address of the next block in the zone. The first byte of the block header is a **tag byte** that provides other information on the block. The bits in the tag byte have these meanings:

Bit	Meaning
0–3	The block's size correction
4–5	Reserved
6–7	The block type

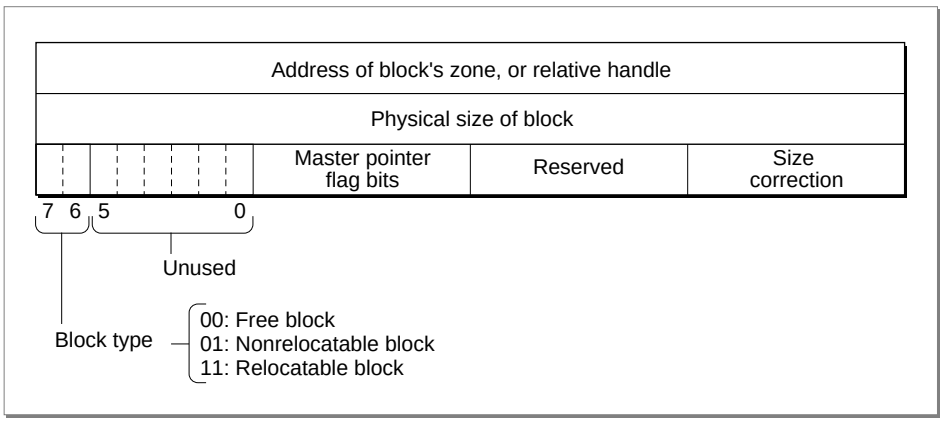
In the tag byte, the high-order 2 bits determine whether a block is free (binary 00), relocatable (binary 10), or nonrelocatable (binary 01). The low-order 4 bits contain a block's **size correction,** the number of unused bytes at the end of the block, beyond the end of the block's contents. This correction is equal to the difference between the block's logical and physical sizes, excluding the 8 bytes of overhead for the block header, as in the following formula:

```
physicalSize = logicalSize + sizeCorrection + 8
```

The contents of the second long word (4 bytes) in the 24-bit block header depend on the type of block. For relocatable blocks, the second long word contains the block's **relative handle:** a pointer to the block's master pointer, expressed as an offset relative to the start of the heap zone rather than as an absolute memory address. Adding the relative handle to the zone pointer produces a true handle for this block. For nonrelocatable blocks, the second long word of the header is just a pointer to the block's zone. For free blocks, the contents of these 4 bytes are undefined.

In a 32-bit zone, a block header consists of 12 bytes, which together make up three long words, as shown in Figure 2-2.

Figure 2-2 A block header in a 32-bit zone



The first byte of the block header is a tag byte that indicates the type of the block. The bits in the tag byte have these meanings:

Bit	Meaning
0–5	Reserved
6–7	The block type

In the tag byte, the high-order 2 bits determine whether a block is free (binary 00), relocatable (binary 10), or nonrelocatable (binary 01).

The second byte in the block header contains the master pointer flag bits, if the block is a relocatable block. Otherwise, this byte is undefined. The bits in this byte have these meanings:

Bit	Meaning
0–4	Reserved
5	If set, block contains resource data
6	If set, block is purgeable
7	If set, block is locked

The low-order byte of the high-order long word contains the block's size correction. This correction is equal to the difference between the block's logical and physical sizes, excluding the 12 bytes of overhead for the block header, as follows:

```
physicalSize = logicalSize + sizeCorrection + 12
```

The second long word in the 32-bit block header contains the block's physical size, and the third long word contains the block's relative handle. These fields have the same meaning as the corresponding fields in the 24-bit block header.

Memory Manager Reference

This section describes the data types and routines provided by the Memory Manager. It describes the general-purpose data types the Memory Manager defines and all routines that relate to manipulating blocks of memory or managing memory in the application heap zone. This section also describes the data structures and routines that allow your application to allocate temporary memory and to use multiple heap zones.

Data Types

This section discusses the general-purpose data types defined by the Memory Manager. Most of these types are used throughout the system software.

The Memory Manager uses pointers and handles to reference nonrelocatable and relocatable blocks, respectively. The data types `Ptr` and `Handle` define pointers and handles as follows:

```
TYPE
   SignedByte     = –128..127;
   Byte           = 0..255;
   Ptr            = ^SignedByte;
   Handle         = ^Ptr;
```

The `SignedByte` type stands for an arbitrary byte in memory, just to give `Ptr` and `Handle` something to point to. The `Byte` type is an alternative definition that treats byte-length data as an unsigned rather than a signed quantity.

Many other data types also use the concept of pointers and handles. For example, the Macintosh system software stores strings in arrays of up to 255 characters, with the first byte of the array storing the length of the string. Some Toolbox routines allow you to pass such a string directly; others require that you pass a pointer or handle to a string. The following type definitions define character strings:

```
TYPE
   Str255         = STRING[255];
   StringPtr      = ^Str255;
   StringHandle   = ^StringPtr;
```

Some Toolbox routines allow you to execute code after a certain amount of time elapses or after a certain condition is met. Any such routine requires you to pass the address of the routine containing the code to be executed so that it knows what routine to call when the time has elapsed or the condition has been met. You use the data type `ProcPtr` to define a pointer to a procedure or function.

```
TYPE ProcPtr = Ptr;
```

For example, after the declarations

```
VAR
   aProcPtr: ProcPtr;

PROCEDURE MyProc;
BEGIN
            ...
END;
```

you can make `aProcPtr` reference the `MyProc` procedure by using the `@` operator, as follows:

```
aProcPtr := @MyProc;
```

With the @ operator, you can assign procedures and functions to variables of type `ProcPtr`, embed them in data structures, and pass them as arguments to other routines. Notice, however, that the data type `ProcPtr` technically points to an arbitrary byte, not an actual routine. As a result, there's no direct way in Pascal to access the underlying routine via this pointer in order to call it. (See Listing 2-4 on page 2-13 for some assembly-language code you can use to do so.) The routines in the Operating System and Toolbox, which are written in assembly language, can however, call routines designated by pointers of type `ProcPtr`.

Note
You can't use the @ operator to reference procedures or functions whose declarations are nested within other routines. ◆

The Memory Manager uses the `Size` data type to refer to the size, in bytes, of memory blocks. For example, when specifying how large a relocatable block you want to allocate, you pass a parameter of type `Size`. The `Size` data type is also defined as a long integer.

```
TYPE Size      = LongInt;
```

Memory Manager Routines

This section describes the routines provided by the Memory Manager. You can use these routines to set up your application's partition, 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. The Memory Manager also provides routines that allow you to allocate temporary memory and manipulate heap zones.

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

SEE ALSO

If you initialize a new zone, you can specify the number of master pointers that a master pointer block should contain. See the description of the `InitZone` procedure on page 2-86 for details.

Allocating and Releasing Relocatable Blocks of Memory

You can use the `NewHandle` function to allocate a relocatable block of memory, or the `NewEmptyHandle` function to allocate handles for which you do not yet need blocks of memory. If you want to allocate new blocks of memory in the system heap or with their bits precleared to 0, you can use the functions `NewHandleSys`, `NewHandleClear`, and `NewHandleSysClear`.

▲ **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

You can specify that the `NewHandle` function apply to the system heap zone instead of the current zone by setting bit 10 of the routine trap word. In most development systems, you can do this by supplying the word `SYS` as the second argument to the routine macro, as follows:

```
_NewHandle ,SYS
```

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
```

You can combine `SYS` and `CLEAR` in the same macro call, but `SYS` must come first.

```
_NewHandle ,SYS,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 2-55.

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 2-56.

NewHandleSys

You can use the `NewHandleSys` function to allocate a relocatable block of memory of a specified size in the system heap.

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

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

DESCRIPTION

The `NewHandleSys` function works much as the `NewHandle` function does, but attempts to allocate the requested block in the system heap zone instead of in the current heap zone. If it cannot, it returns `NIL`.

RESULT CODES

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

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

NewHandleSysClear

You can use the `NewHandleSysClear` function to allocate, in the system heap, prezeroed memory in a relocatable block of a specified size.

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

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

DESCRIPTION

The `NewHandleSysClear` function works much as the `NewHandleClear` function does, but attempts to allocate the requested block in the system heap zone instead of in the current heap zone. `NewHandleSysClear` sets all bytes in the new block to 0 instead of leaving the contents of the block undefined.

RESULT CODES

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

NewEmptyHandle

If you want to initialize a handle but not allocate any space for it, use the `NewEmptyHandle` function. The Resource Manager uses this function extensively, but you probably won't need to use it.

```
FUNCTION NewEmptyHandle: Handle;
```


DESCRIPTION

The `NewEmptyHandle` function initializes a new handle by allocating a master pointer for it, but it does not allocate any memory for the handle to control. `NewEmptyHandle` sets the handle's master pointer to `NIL`.

SPECIAL CONSIDERATIONS

Because `NewEmptyHandle` might need to call the `MoreMasters` procedure to allocate new master pointers, it might allocate memory. Thus, you should not call `NewEmptyHandle` at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for `NewEmptyHandle` are

Registers on exit

A0 Address of the new block's master pointer

D0 Result code

You can specify that the `NewEmptyHandle` function apply to the system heap zone instead of the current zone. To do so, set bit 10 of the routine trap word. In most development systems, you can do this by supplying the word `SYS` as the second argument to the routine macro, as follows:

```
_NewEmptyHandle ,SYS
```

RESULT CODES

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

SEE ALSO

When you want to allocate memory for the empty handle, use the `ReallocateHandle` procedure, described on page 2-52.

NewEmptyHandleSys

If you want to initialize a handle in the system heap but not allocate any space for it, use the `NewEmptyHandleSys` function. The Resource Manager uses this function extensively, but you probably won't need to use it.

```
FUNCTION NewEmptyHandleSys: Handle;
```

DESCRIPTION

The `NewEmptyHandleSys` function initializes a new handle in the system heap by allocating a master pointer for it, but it does not allocate any memory for the handle to control. `NewEmptyHandleSys` sets the handle's master pointer to `NIL`.

SPECIAL CONSIDERATIONS

Because `NewEmptyHandleSys` might need to call the `MoreMasters` procedure to allocate new master pointers, it might allocate memory. Thus, you should not call `NewEmptyHandleSys` at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for `NewEmptyHandleSys` are

Registers on exit

A0	Address of the new block's master pointer
D0	Result code

RESULT CODES

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

SEE ALSO

When you want to allocate memory for the empty handle, use the `ReallocateHandle` procedure, described on page 2-52.

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 in the system heap or with their bits precleared to 0, you can use the `NewPtrSys`, `NewPtrClear`, and `NewPtrSysClear` functions.

▲ **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

You can specify that the `NewPtr` function apply to the system heap zone instead of the current zone. To do so, set bit 10 of the routine trap word. In most development systems, you can do this by supplying the word `SYS` as the second argument to the routine macro, as follows:

```
_NewPtr ,SYS
```

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
```

You can combine SYS and CLEAR in the same macro call, but SYS must come first.

```
_NewPtr ,SYS,CLEAR
```

RESULT CODES

noErr	0	No error
memFullErr	–108	Not enough memory

NewPtrSys

You can use the NewPtrSys function to allocate a nonrelocatable block of memory of a specified size in the system heap.

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

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

DESCRIPTION

The NewPtrSys function works much as the NewPtr function does, but attempts to allocate the requested block in the system heap zone instead of in the current heap zone.

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

NewPtrSysClear

You can use the `NewPtrSysClear` function to allocate, in the system heap, prezeroed memory in a nonrelocatable block of a specified size.

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

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

DESCRIPTION

The `NewPtrSysClear` function works much as the `NewPtr` function does, but attempts to allocate the requested block in the system heap zone instead of in the current heap zone. Also, it sets all bytes in the new block to 0 instead of leaving the contents of the block undefined.

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

Changing the Sizes of Relocatable and Nonrelocatable Blocks

You can use the `GetHandleSize` function and the `SetHandleSize` procedure to find out and change the logical size of a relocatable block, and you can use the `GetPtrSize` function and the `SetPtrSize` procedure to find out and change the logical size of a nonrelocatable block.

GetHandleSize

You can use the `GetHandleSize` function to find out the logical size of the relocatable block corresponding to a handle.

```
FUNCTION GetHandleSize (h: Handle): Size;
```

`h` A handle to a relocatable block.

DESCRIPTION

The `GetHandleSize` function returns the logical size, in bytes, of the relocatable block whose handle is `h`. In case of an error, `GetHandleSize` returns 0.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `GetHandleSize` are

Registers on entry

A0 Handle to the relocatable block

Registers on exit

D0 If `>=0`, number of bytes in relocatable block

If `<0`, result code

The trap dispatcher sets the condition codes before returning from a trap by testing the low-order word of register D0 with a `TST.W` instruction. Because the block size returned in D0 by `_GetHandleSize` is a full 32-bit long word, the word-length test sets the condition codes incorrectly in this case. To branch on the contents of D0, use your own `TST.L` instruction on return from the trap to test the full 32 bits of the register.

SPECIAL CONSIDERATIONS

You shouldn't call `GetHandleSize` at interrupt time because the heap might be in an inconsistent state.

RESULT CODES

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

SetHandleSize

You can use the `SetHandleSize` procedure to change the logical size of the relocatable block corresponding to a handle.

```
PROCEDURE SetHandleSize (h: Handle; newSize: Size);
```

`h` A handle to a relocatable block.

`newSize` The desired new logical size, in bytes, of the relocatable block.

DESCRIPTION

The `SetHandleSize` procedure attempts to change the logical size of the relocatable block whose handle is `h`. The new logical size is specified by `newSize`.

`SetHandleSize` might need to move the relocatable block to obtain enough space for the resized block. Thus, for best results you should unlock a block before resizing it.

An attempt to increase the size of a locked block might fail, because of blocks above and below it that are either nonrelocatable or locked. You should be prepared for this possibility.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `SetHandleSize` are

Registers on entry

A0 Handle to the relocatable block

D0 Desired new size of relocatable block

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough memory
`nilHandleErr`	–109	`NIL` master pointer
`memWZErr`	–111	Attempt to operate on a free block

SEE ALSO

Instead of using the `SetHandleSize` procedure to set the size of a handle to 0, you can use the `EmptyHandle` procedure, described on page 2-51.

GetPtrSize

You can use the `GetPtrSize` function to find out the logical size of the nonrelocatable block corresponding to a pointer.

```
FUNCTION GetPtrSize (p: Ptr): Size;
```

`p` A pointer to a nonrelocatable block.

DESCRIPTION

The `GetPtrSize` function returns the logical size, in bytes, of the nonrelocatable block pointed to by `p`. In case of an error, `GetPtrSize` returns 0.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `GetPtrSize` are

Registers on entry

A0 Pointer to the nonrelocatable block

Registers on exit

D0 If `>=0`, number of bytes in nonrelocatable block

If `<0`, result code

The trap dispatcher sets the condition codes before returning from a trap by testing the low-order word of register D0 with a `TST.W` instruction. Because the block size returned in D0 by `_GetPtrSize` is a full 32-bit long word, the word-length test sets the condition codes incorrectly in this case. To branch on the contents of D0, use your own `TST.L` instruction on return from the trap to test the full 32 bits of the register.

RESULT CODES

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

SetPtrSize

You can use the `SetPtrSize` procedure to change the logical size of the nonrelocatable block corresponding to a pointer.

```
PROCEDURE SetPtrSize (p: Ptr; newSize: Size);
```

`p`	A pointer to a nonrelocatable block.
`newSize`	The desired new logical size, in bytes, of the nonrelocatable block.

DESCRIPTION

The `SetPtrSize` procedure attempts to change the logical size of the nonrelocatable block pointed to by `p`. The new logical size is specified by `newSize`.

An attempt to increase the size of a nonrelocatable block might fail because of a block above it that is either nonrelocatable or locked. You should be prepared for this possibility.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `SetPtrSize` are

Registers on entry

A0 Pointer to the nonrelocatable block

D0 Desired new size of nonrelocatable block

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough memory
`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 using the routines described on page 2-45 through page 2-50, 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 2-55.

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 2-56.

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 2-52.

You can immediately free the space taken by a handle without disposing of it by calling `EmptyHandle`. This procedure, described on page 2-51, 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 on page 2-52.

HSetRBit

You can use the `HSetRBit` procedure to set the resource flag of a relocatable block. The Resource Manager uses this routine extensively, but you should never need to use it.

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

`h` A handle to a relocatable block.

DESCRIPTION

The `HSetRBit` procedure sets the resource flag of the relocatable block to which `h` is a handle. It does nothing if the flag is already set.

▲ **WARNING**
When the resource flag is set, the Resource Manager identifies the associated relocatable block as belonging to a resource. This can cause problems if that block wasn't actually read from a resource. ▲

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HSetRBit` are

Registers on entry

A0 Handle whose resource flag you want to set

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

HClrRBit

You can use the `HClrRBit` procedure to clear the resource flag of a relocatable block. The Resource Manager uses this routine extensively, but you probably won't need to use it.

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

`h` A handle to a relocatable block.

DESCRIPTION

The `HClrRBit` procedure clears the resource flag of a relocatable block. It does nothing if the flag is already cleared.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HClrRBit` are

Registers on entry

A0 Handle whose resource flag you want to clear

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

To disassociate the data in a resource handle from the resource file, you should use the Resource Manager procedure `DetachResource` instead of this procedure.

Managing Relocatable Blocks

The Memory Manager provides routines that allow you to purge and later reallocate space for relocatable blocks, recreate handles to relocatable blocks if you have access to their master pointers, 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.

If, because you have dereferenced a handle, you no longer have access to it but do have access to its master pointer, you can use the `RecoverHandle` function to recreate the handle.

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 purge all of the blocks in a heap zone that are marked purgeable, use the `PurgeMem` procedure, described on page 2-73.

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 2-34.

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

SEE ALSO

Because `ReallocateHandle` releases any existing relocatable block referenced by the handle `h` before allocating a new one, it does not provide an efficient technique for resizing relocatable blocks. To do that, use the `SetHandleSize` procedure, described on page 2-40.

RecoverHandle

The Memory Manager does not allow you to change relocatable blocks into nonrelocatable blocks, or vice-versa. However, if you no longer have access to a handle but still have access to its master pointer, you can use the `RecoverHandle` function to recreate a handle to the relocatable block referenced by the master pointer.

```
FUNCTION RecoverHandle (p: Ptr): Handle;
```

`p` The master pointer to a relocatable block.

DESCRIPTION

The `RecoverHandle` function returns a handle to the relocatable block pointed to by `p`. If `p` doesn't point to a valid block, the results of `RecoverHandle` are undefined.

SPECIAL CONSIDERATIONS

Even though `RecoverHandle` does not move or purge memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `RecoverHandle` are

Registers on entry

A0 Master pointer

Registers on exit

A0 Handle to master pointer's relocatable block

D0 Unchanged

Unlike most other Memory Manager routines, `RecoverHandle` does not return a result code in register D0; the previous contents of D0 are preserved unchanged. The result code is, however, returned by `MemError`.

The `RecoverHandle` function looks only in the current heap zone for the relocatable block pointed to by the parameter `p`. If you want to use the `RecoverHandle` function to recover a handle for a relocatable block in the system heap, set bit 10 of the routine trap word. In most development systems, you can do this by supplying the word `SYS` as the second argument to the routine macro, as follows:

```
_RecoverHandle ,SYS
```

RESULT CODES

`noErr`	0	No error
`memBCErr`	–115	Block check failed

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

The `ReserveMem` procedure reserves memory in the current heap zone. If you want to reserve memory in the system heap zone rather than in the current heap zone, set bit 10 of the routine trap word. In most development systems, you can do this by supplying the word `SYS` as the second argument to the routine macro, as follows:

```
_ResrvMem ,SYS
```

RESULT CODES

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

ReserveMemSys

If you plan to lock a relocatable block for long periods of time in the system heap zone, use the `ReserveMemSys` procedure to reserve space for the block as low in the system heap as possible.

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

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

DESCRIPTION

The `ReserveMemSys` procedure works much as the `ReserveMem` procedure does, but reserves memory in the system heap zone rather than in the current heap zone.

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 three routines for copying blocks of memory referenced by pointers. To copy a block of memory to a nonrelocatable block, use the `BlockMove` procedure. To copy to a new relocatable block, use the `PtrToHand` function. To copy to an existing relocatable block, use the `PtrToXHand` function. If you want to use any of these routines to copy memory you access with a handle, you must first dereference and lock the handle. A fourth routine, `HandToHand`, allows you to copy information from one handle to another.

To concatenate blocks of memory, you can use the `HandAndHand` and `PtrAndHand` functions.

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

PtrToHand

To copy data referenced by a pointer to a new relocatable block, use the `PtrToHand` function.

```
FUNCTION PtrToHand (srcPtr: Ptr; VAR dstHndl: Handle;
                    size: LongInt): OSErr;
```

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

`dstHndl` A handle for which you have not yet allocated any memory. The `PtrToHand` function allocates memory for the handle and copies `size` bytes beginning at `srcPtr` into it.

`size` The number of bytes to copy.

DESCRIPTION

The `PtrToHand` function returns, in `dstHndl`, a newly created handle to a copy of the number of bytes specified by the `size` parameter, beginning at the location specified by `srcPtr`. The `dstHndl` parameter must be a handle variable that is not empty and is not a handle to an allocated block of size 0.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `PtrToHand` are

Registers on entry

A0 Pointer to source

D0 Number of bytes to copy

Registers on exit

A0 Destination handle

D0 Result code

RESULT CODES

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

SEE ALSO

You can use the `PtrToHand` function to copy data from one handle to a new handle if you dereference and lock the source handle. However, if you want to copy all of the data from one handle to another, the `HandToHand` function (described on page 2-62) is more efficient.

PtrToXHand

To copy data referenced by a pointer to an already existing relocatable block, use the `PtrToXHand` function.

```
FUNCTION PtrToXHand (srcPtr: Ptr; dstHndl: Handle; size: LongInt):
                     OSErr;
```

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

`dstHndl`	A handle to an already existing relocatable block to which to copy `size` bytes, beginning at `srcPtr`.
`size`	The number of bytes to copy.

DESCRIPTION

The `PtrToXHand` function makes the existing handle, specified by `dstHndl`, a handle to a copy of the number of bytes specified by the `size` parameter, beginning at the location specified by `srcPtr`.

SPECIAL CONSIDERATIONS

Because `PtrToXHand` affects memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `PtrToXHand` are

Registers on entry

A0	Pointer to source
A1	Handle to destination
D0	Number of bytes to copy

Registers on exit

A0	Handle to destination
D0	Result code

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough memory
`nilHandleErr`	–109	`NIL` master pointer
`memWZErr`	–111	Attempt to operate on a free block

HandToHand

Use the `HandToHand` function to copy all of the data from one relocatable block to a new relocatable block.

```
FUNCTION HandToHand (VAR theHndl: Handle): OSErr;
```

`theHndl`	On entry, a handle to the relocatable block whose data is to be copied. On exit, a handle to a new relocatable block whose data duplicates that of the original.

DESCRIPTION

The `HandToHand` function attempts to copy the information in the relocatable block to which `theHndl` is a handle; if successful, `HandToHand` returns a handle to the new relocatable block in `theHndl`. The new relocatable block is created in the same heap zone as the original block (which might not be the current heap zone).

Because `HandToHand` replaces its input parameter with the new handle, you should retain the original value of the input parameter somewhere else, or you won't be able to access it. Here is an example:

```
VAR
   original, copy: Handle;
   myErr: OSErr;
...
   copy := original;          {both handles access same block}
   myErr := HandToHand(copy); {copy now points to copy of block}
```

SPECIAL CONSIDERATIONS

If successful in creating a new relocatable block, the `HandToHand` function does not duplicate the properties of the original block. The new block is unlocked, unpurgeable, and not a resource. You might need to call `HLock`, `HPurge`, or `HSetRBit` (or the combination of `HGetState` and `HSetState`) to adjust the properties of the new block.

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

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HandToHand` are

Registers on entry

A0 Handle to original data

Registers on exit

A0 Handle to copy of data

D0 Result code

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough memory
`nilHandleErr`	–109	`NIL` master pointer
`memWZErr`	–111	Attempt to operate on a free block

SEE ALSO

If you want to copy only part of a relocatable block into a new relocatable block, use the `PtrToHand` function, described on page 2-60, after locking and dereferencing a handle to the relocatable block to be copied.

HandAndHand

Use the `HandAndHand` function to concatenate two relocatable blocks.

```
FUNCTION HandAndHand (aHndl, bHndl: Handle): OSErr;
```

`aHndl` A handle to the first relocatable block, whose contents do not change but are concatenated to the end of the second relocatable block.

`bHndl` A handle to the second relocatable block, whose size the Memory Manager expands so that it can concatenate the information from `aHndl` to the end of the contents of this block.

DESCRIPTION

The `HandAndHand` function concatenates the information from the relocatable block to which `aHndl` is a handle onto the end of the relocatable block to which `bHndl` is a handle. The `aHndl` variable remains unchanged.

▲ **WARNING**
The `HandAndHand` function dereferences the handle `aHndl`. You must call the `HLock` procedure to lock the block before calling `HandAndHand`. Afterward, you can call the `HUnlock` procedure to unlock it. Alternatively, you can save the block's original state by calling the `HGetState` function, lock the block by calling `HLock`, and then restore the original settings by calling `HSetState`. ▲

SPECIAL CONSIDERATIONS

Because `HandAndHand` moves memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HandAndHand` are

Registers on entry

A0 Handle to be concatenated

A1 Handle to contain itself, data from A0's handle

Registers on exit

A0 Handle to concatenated data

D0 Result code

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough memory
`nilHandleErr`	–109	`NIL` master pointer
`memWZErr`	–111	Attempt to operate on a free block

PtrAndHand

Use the `PtrAndHand` function to concatenate part or all of a memory block to the end of a relocatable block.

```
FUNCTION PtrAndHand (pntr: Ptr; hndl: Handle; size: LongInt):
                     OSErr;
```

`pntr` A pointer to the beginning of the data that the Memory Manager is to concatenate onto the end of the relocatable block.

`hndl` A handle to the relocatable block, whose size the Memory Manager expands so that it can concatenate the information from `pntr` onto the end of this block.

`size` The number of bytes of the block referenced by `pntr` to be copied.

DESCRIPTION

The `PtrAndHand` function takes the number of bytes specified by the `size` parameter, beginning at the location specified by `pntr`, and concatenates them onto the end of the relocatable block to which `hndl` is a handle.

The contents of the source block remain unchanged.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `PtrAndHand` are

Registers on entry

A0 Pointer to data to copy

A1 Handle to relocatable block at whose end the copied data concatenated

A2 Number of bytes to concatenate

Registers on exit

A0 Handle to now-concatenated relocatable block

D0 Result code

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough memory
`nilHandleErr`	–109	`NIL` master pointer
`memWZErr`	–111	Attempt to operate on a free block

Assessing Memory Conditions

The Memory Manager provides four routines to test how much memory is available, one routine used after memory operations to determine if an error occurred, and one routine to determine the location in memory of the top of your application's partition.

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 compacting the heap, use the `FreeMem` and `MaxBlock` functions, respectively. To determine what those values would be after a purge of the heap zone, call the `PurgeSpace` procedure. Finally, to find out how much your stack can grow before it collides with the heap, use the `StackSpace` function.

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

FreeMem

By calling the `FreeMem` function, you can find out the total amount of free space, in bytes, in the current heap zone.

```
FUNCTION FreeMem: LongInt;
```

DESCRIPTION

The `FreeMem` function returns the total amount of free space (in bytes) in the current heap zone. Note that it usually isn't possible to allocate a block of that size, because of heap fragmentation due to nonrelocatable or locked blocks.

SPECIAL CONSIDERATIONS

Even though `FreeMem` does not move or purge memory, you should not call it at interrupt time because the heap might be in an inconsistent state.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for `FreeMem` are

Registers on exit

D0 Number of bytes available in heap zone

The `FreeMem` function reports the number of free bytes in the current heap zone. If you want to know how many bytes are available in the system heap zone rather than in the current heap zone, set bit 10 of the routine trap word. In most development systems, you can do this by supplying the word `SYS` as the second argument to the routine macro, as follows:

```
_FreeMem ,SYS
```

RESULT CODES

`noErr`	0	No error

FreeMemSys

To determine how much free space remains in the system heap zone, use the `FreeMemSys` function.

```
FUNCTION FreeMemSys: LongInt;
```

DESCRIPTION

The `FreeMemSys` function works much as the `FreeMem` function does, but returns the total amount of free memory in the system heap zone instead of in the current heap zone.

RESULT CODES

`noErr`	0	No error

MaxBlock

Use the `MaxBlock` function to determine the size of the largest block you could allocate in the current heap zone after a compaction.

```
FUNCTION MaxBlock: LongInt;
```

DESCRIPTION

The `MaxBlock` function returns the maximum contiguous space, in bytes, that you could obtain after compacting the current heap zone. `MaxBlock` does not actually do the compaction.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for `MaxBlock` are

Registers on exit

D0 Size of largest allocatable block

If you want to know the size of the largest allocatable block in the system heap zone, rather than in the current heap zone, set bit 10 of the routine trap word. In most development systems, you can do this by supplying the word `SYS` as the second argument to the routine macro, as follows:

```
_MaxBlock ,SYS
```

RESULT CODES

`noErr`	0	No error

MaxBlockSys

Use the `MaxBlockSys` function to determine the size of the largest block you could allocate in the system heap after a compaction.

```
FUNCTION MaxBlockSys: LongInt;
```

DESCRIPTION

The `MaxBlockSys` function works much as the `MaxBlock` function does, but returns the maximum contiguous space, in bytes, that you could obtain after compacting the system heap. `MaxBlockSys` does not actually do the compaction.

RESULT CODES

`noErr`	0	No error

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

If you want to test the system heap zone instead of the current zone, set bit 10 of the routine trap word. In most development systems, you can do this by supplying the word `SYS` as the second argument to the routine macro, as follows:

```
_PurgeSpace ,SYS
```

RESULT CODES

`noErr`	0	No error

StackSpace

Use the `StackSpace` function to find out how much space there is between the bottom of the stack and the top of the application heap.

```
FUNCTION StackSpace: LongInt;
```

DESCRIPTION

The `StackSpace` function returns the current amount of stack space (in bytes) between the current stack pointer and the application heap at the instant of return from the trap.

SPECIAL CONSIDERATIONS

Ordinarily, you determine the maximum amount of stack space you need before you ship your application. In general, therefore, this routine is useful only during debugging to determine how big to make the stack. However, if your application calls a recursive function that conceivably could call itself many times, that function should keep track of the stack space and take appropriate action if it becomes too low.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for `StackSpace` are

Registers on exit

D0 Number of bytes between stack and heap

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

Freeing Memory

The Memory Manager compacts and purges the heap whenever necessary to satisfy requests for memory. You can also compact or purge the heap manually. To compact the current heap zone manually, use the `CompactMem` function. To purge it manually, use the `PurgeMem` procedure. To do both at once, use the `MaxMem` function. To perform the same operations on the system heap zone, use the `CompactMemSys` function, the `PurgeMemSys` procedure, and the `MaxMemSys` function.

Note

Most applications don't need to call the routines described in this section. Normally you should let the Memory Manager compact or purge your application heap. ◆

CompactMem

The Memory Manager compacts the heap for you when you make a memory request that it can't fill. However, you can use the `CompactMem` function to compact the current heap zone manually.

```
FUNCTION CompactMem (cbNeeded: Size): Size;
```

`cbNeeded` The size, in bytes, of the block for which `CompactMem` should attempt to make room.

DESCRIPTION

The `CompactMem` function compacts the current heap zone by moving unlocked, relocatable blocks down until they encounter nonrelocatable blocks or locked, relocatable blocks, but not by purging blocks. It continues compacting until it either finds a contiguous block of at least `cbNeeded` free bytes or has compacted the entire zone.

The `CompactMem` function returns the size, in bytes, of the largest contiguous free block for which it could make room, but it does not actually allocate that block.

To compact the entire heap zone, call `CompactMem(maxSize)`. The Memory Manager defines the constant `maxSize` for the largest contiguous block possible in the 24-bit Memory Manager:

```
CONST
   maxSize           = $800000;        {maximum size of a block}
```

SPECIAL CONSIDERATIONS

Because `CompactMem` moves memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `CompactMem` are

Registers on entry

D0 Size of block to make room for

Registers on exit

D0 Size of largest allocatable block

The `CompactMem` function compacts the current heap zone. If you want to compact the system heap zone rather than the current heap zone, set bit 10 of the routine trap word. In most development systems, you can do this by supplying the word `SYS` as the second argument to the routine macro, as follows:

```
_CompactMem ,SYS
```

RESULT CODES

`noErr`	0	No error

CompactMemSys

You can use the `CompactMemSys` function to compact the system heap zone manually.

```
FUNCTION CompactMemSys (cbNeeded: Size): Size;
```

`cbNeeded` The size in bytes of the block for which `CompactMemSys` should attempt to make room.

DESCRIPTION

The `CompactMemSys` function works much as the `CompactMem` function does, but compacts the system heap instead of the current heap.

RESULT CODES

`noErr`	0	No error

PurgeMem

The Memory Manager purges the heap for you when you make a memory request that it can't fill. However, you can use the `PurgeMem` procedure to purge the current heap zone manually.

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

`cbNeeded` The size, in bytes, of the block for which `PurgeMem` should attempt to make room.

DESCRIPTION

The `PurgeMem` procedure sequentially purges blocks from the current heap zone until it either allocates a contiguous block of at least `cbNeeded` free bytes or has purged the entire zone. If it purges the entire zone without creating a contiguous block of at least `cbNeeded` free bytes, `PurgeMem` generates a `memFullErr`.

The `PurgeMem` procedure purges only relocatable, unlocked, purgeable blocks.

The `PurgeMem` procedure does not actually attempt to allocate a block of `cbNeeded` bytes.

To purge the entire heap zone, call `PurgeMem(maxSize)`.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `PurgeMem` are

Registers on entry

D0 Size of block to make room for

Registers on exit

D0 Result code

The `PurgeMem` procedure purges the current heap zone. If you want to purge the system heap zone rather than the current heap zone, set bit 10 of the routine trap word. In most development systems, you can do this by supplying the word `SYS` as the second argument to the routine macro, as follows:

```
_PurgeMem ,SYS
```

RESULT CODES

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

PurgeMemSys

You can use the `PurgeMemSys` procedure to purge the system heap manually.

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

`cbNeeded` The size, in bytes, of the block for which `PurgeMemSys` should attempt to make room.

DESCRIPTION

The `PurgeMemSys` procedure works much as the `PurgeMem` procedure does, but purges the system heap instead of the current heap.

RESULT CODES

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

MaxMem

Use the `MaxMem` function to compact and purge the current heap zone.

```
FUNCTION MaxMem (VAR grow: Size): Size;
```

`grow` On exit, the maximum number of bytes by which the current heap zone can grow. After a call to `MaxApplZone`, `MaxMem` always returns 0 in this parameter.

DESCRIPTION

The `MaxMem` function compacts the current heap zone and purges all relocatable, unlocked, and purgeable blocks from the zone. It returns the size, in bytes, of the largest contiguous free block in the zone after the compacting and purging. If the current zone is the original application zone, the `grow` parameter is set to the maximum number of bytes by which the zone can grow. For any other heap zone, `grow` is set to 0. `MaxMem` doesn't actually expand the zone or call the zone's grow-zone function.

SPECIAL CONSIDERATIONS

Because `MaxMem` moves and purges memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for `MaxMem` are

Registers on exit

A0 Number of bytes zone can grow

D0 Size in bytes of largest allocatable block

The `MaxMem` function compacts the current heap zone. If you want to compact and purge the system heap zone rather than the current heap zone, set bit 10 of the routine trap word. In most development systems, you can do this by supplying the word `SYS` as the second argument to the routine macro, as follows:

```
_MaxMem ,SYS
```

RESULT CODES

`noErr`	0	No error

MaxMemSys

You can use the `MaxMemSys` function to purge and compact the system heap zone manually.

```
FUNCTION MaxMemSys (VAR grow: Size): Size;
```

`grow` On exit, the `MaxMemSys` function sets this parameter to 0. Ignore this parameter.

DESCRIPTION

The `MaxMemSys` function works much as the `MaxMem` function does, but compacts and purges the system heap instead of the current heap. It returns the size, in bytes, of the largest block you can allocate in the system heap.

RESULT CODES

`noErr`	0	No error

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. For explanations of how grow-zone functions work and an example of a memory-management scheme that uses a grow-zone function, see the discussion of low-memory conditions in the chapter "Introduction to Memory Management" in this book.

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 2-89 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

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`.

Allocating Temporary Memory

In system software version 7.0 and later, you can manipulate temporary memory with three routines that are counterparts to other Memory Manager routines. The `TempNewHandle` function allocates a new block of relocatable memory, the `TempFreeMem` function returns the total amount of free memory available for temporary

allocation, and the `TempMaxMem` function compacts the heap zone and returns the size of the largest contiguous block available for temporary allocation.

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

TempNewHandle

To allocate a new relocatable block of temporary memory, call the `TempNewHandle` function after making sure that there is enough free space to satisfy the request.

```
FUNCTION TempNewHandle (logicalSize: Size;
                        VAR resultCode: OSErr): Handle;
```

`logicalSize`
The requested logical size, in bytes, of the new temporary block of memory.

`resultCode`
On exit, the result code from the function call.

DESCRIPTION

The `TempNewHandle` function returns a handle to a block of size `logicalSize`. If it cannot allocate a block of that size, the function returns `NIL`. Before you use the returned handle, make sure its value is not `NIL`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `TempNewHandle` are

Trap macro	Selector
`_OSDispatch`	$001D

SPECIAL CONSIDERATIONS

Because `TempNewHandle` might allocate memory, you should not call it at interrupt time.

Note that `TempNewHandle` returns its result code in a parameter, not through `MemError`.

RESULT CODES

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

TempFreeMem

To find out the total amount of memory available for temporary allocation, use the `TempFreeMem` function.

```
FUNCTION TempFreeMem: LongInt;
```

DESCRIPTION

The `TempFreeMem` function returns the total amount of free temporary memory that you could allocate by calling `TempNewHandle`. The returned value is the total number of free bytes. Because these bytes might be dispersed throughout memory, it is ordinarily not possible to allocate a single relocatable block of that size.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `TempFreeMem` are

Trap macro	Selector
`_OSDispatch`	$0018

SPECIAL CONSIDERATIONS

Even though `TempFreeMem` does not move or purge memory, you should not call it at interrupt time.

TempMaxMem

To find the size of the largest contiguous block available for temporary allocation, use the `TempMaxMem` function.

```
FUNCTION TempMaxMem (VAR grow: Size): Size;
```

`grow` On exit, this parameter always contains 0 after the function call because temporary memory does not come from the application's heap zone, and only that zone can grow. Ignore this parameter.

DESCRIPTION

The `TempMaxMem` function compacts the current heap zone and returns the size of the largest contiguous block available for temporary allocation.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `TempMaxMem` are

Trap macro	Selector
`_OSDispatch`	$0015

SPECIAL CONSIDERATIONS

Because `TempMaxMem` could move memory, you should not call it at interrupt time.

Accessing Heap Zones

The majority of applications, which allocate memory in their application heap zone only, do not need to use any of the routines in this section. The few applications that do allocate memory in zones other than the application heap zone can use the `GetZone` function and the `SetZone` procedure to get and set the current zone, the `ApplicationZone` and `SystemZone` functions to obtain pointers to the application and system zones, and the `HandleZone` and `PtrZone` functions to find the zones in which relocatable and nonrelocatable blocks lie.

GetZone

To find which zone is current, use the `GetZone` function.

```
FUNCTION GetZone: THz;
```

DESCRIPTION

The `GetZone` function returns a pointer to the current heap zone.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for `GetZone` are

Registers on exit	
A0	Pointer to current heap zone
D0	Result code

The global variable `TheZone` contains a pointer to the current heap zone.

RESULT CODES

`noErr`	0	No error

SetZone

To change the current heap zone, you can use the `SetZone` procedure.

```
PROCEDURE SetZone (hz: THz);
```

`hz` A pointer to the heap zone to make current.

DESCRIPTION

The `SetZone` procedure makes the zone to which `hz` points the current heap zone. Often, you use the `SetZone` procedure in conjunction with one of the `ApplicationZone`, `SystemZone`, `HandleZone`, and `PtrZone` functions. For example, the code `SetZone(SystemZone)` makes the system heap zone current.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `SetZone` are

Registers on entry

A0 Pointer to new current heap zone

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error

ApplicationZone

To obtain a pointer to the application heap zone, you can use the `ApplicationZone` function.

```
FUNCTION ApplicationZone: THz;
```

DESCRIPTION

The `ApplicationZone` function returns a pointer to the original application heap zone.

ASSEMBLY-LANGUAGE INFORMATION

The global variable `ApplZone` contains a pointer to the original application heap zone.

SystemZone

To obtain a pointer to the system heap zone, you can use the `SystemZone` function.

```
FUNCTION SystemZone: THz;
```

DESCRIPTION

The `SystemZone` function returns a pointer to the system heap zone.

ASSEMBLY-LANGUAGE INFORMATION

The global variable `SysZone` contains a pointer to the system heap zone.

HandleZone

If you need to know which heap zone contains a particular relocatable block, you can use the `HandleZone` function.

```
FUNCTION HandleZone (h: Handle): THz;
```

`h` A handle to a relocatable block.

DESCRIPTION

The `HandleZone` function returns a pointer to the heap zone containing the relocatable block whose handle is `h`. In case of an error, the result returned by `HandleZone` is undefined and should be ignored.

IMPORTANT

If the handle `h` is empty (that is, if it points to a `NIL` master pointer), `HandleZone` returns a pointer to the heap zone that contains the master pointer. ▲

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `HandleZone` are

Registers on entry

A0 Handle whose zone is to be found

Registers on exit

A0 Pointer to handle's heap zone

D0 Result code

RESULT CODES

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

PtrZone

If you have allocated a nonrelocatable block and need to know in which zone it lies, you can use the `PtrZone` function.

```
FUNCTION PtrZone (p: Ptr): THz;
```

`p` A pointer to a nonrelocatable block.

DESCRIPTION

The `PtrZone` function returns a pointer to the heap zone containing the nonrelocatable block pointed to by `p`.

In case of an error, the result returned by `PtrZone` is undefined and should be ignored.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `PtrZone` are

Registers on entry

A0 Pointer whose zone is to be found

Registers on exit

A0 Pointer to heap zone of nonrelocatable block

D0 Result code

RESULT CODES

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

Manipulating Heap Zones

The Memory Manager provides several routines for initializing and resizing heap zones.

To obtain information about the current application partition, applications can call the `GetApplLimit` function and the `TopMem` function. If your application uses the stack extensively, you might want to ensure that the stack is set to at least some minimum size, at the expense of the heap. To do so, use the `SetApplLimit` procedure to change the application heap limit before you call the `MaxApplZone` procedure.

To initialize a new heap zone, use the `InitZone` procedure. The Operating System automatically initializes the application zone by calling the `SetApplBase` procedure, which subsequently calls the `InitApplZone` procedure.

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

TopMem

To find out the location of the top of an application's partition, you can use the `TopMem` function, which exhibits special behavior during the startup process.

```
FUNCTION TopMem: Ptr;
```

DESCRIPTION

Except during the startup process, the `TopMem` function returns a pointer to the byte at the top of an application's partition, directly above the jump table. The function does this to maintain compatibility with programs that check `TopMem` to find out how much memory is installed in a computer. To obtain this information, you can currently use the `Gestalt` function.

The function exhibits special behavior at startup time, and the value it returns controls the amount by which an extension can lower the value of the global variable `BufPtr` at startup time. If you are writing a system extension, you should not lower the value of `BufPtr` by more than `MemTop DIV 2 + 1024`. If you do lower `BufPtr` too far, the startup process generates an out-of-memory system error.

You should never need to call `TopMem` except during the startup process.

ASSEMBLY-LANGUAGE INFORMATION

The `TopMem` function returns the value of the `MemTop` global variable.

InitZone

If you want to use heap zones other than the original application heap zone, a temporary memory zone, or the system heap zone, you can use the `InitZone` procedure to initialize a new heap zone.

```
PROCEDURE InitZone (pGrowZone: ProcPtr; cMoreMasters: Integer;
                    limitPtr, startPtr: Ptr);
```

`pGrowZone` A pointer to a grow-zone function for the new heap zone. If you do not want the new zone to have a grow-zone function, set this parameter to `NIL`.

`cMoreMasters`
The number of master pointers that should be allocated at a time for the new zone. The Memory Manager allocates this number initially, and, if it needs to allocate more later, allocates them in increments of this same number.

`limitPtr` The first byte beyond the end of the zone.

`startPtr` The first byte of the new zone.

DESCRIPTION

The `InitZone` procedure creates a new heap zone, initializes its header and trailer, and makes it the current zone. Although the new zone occupies memory addresses from `startPtr` through `limitPtr–1`, the new zone includes a zone header and a zone trailer. In addition, the new zone contains a block header for the master pointer block and 4 bytes for each master pointer. If you need to create a zone with some specific number of usable bytes, see "Organization of Memory," beginning on page 2-19, for details on the sizes of the zone header, zone trailer, and block header.

Note
The sizes of zones and block headers may change in future system software versions. You should ensure that your zones are large enough to accommodate a reasonable increase in the sizes of those structures. ◆

SPECIAL CONSIDERATIONS

Because `InitZone` changes the current zone, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `InitZone` are

Registers on entry

A0 Pointer to parameter block

Registers on exit

D0 Result code

The parameter block whose address is passed in register A0 has no Pascal type definition. It has this structure:

Parameter block

→	`startPtr`	`Ptr`	The first byte of the new zone.
→	`limitPtr`	`Ptr`	The first byte beyond the new zone.
→	`cMoreMasters`	`Integer`	The number of master pointers to be allocated at a time.
→	`pGrowZone`	`ProcPtr`	A pointer to the new zone's grow-zone function, or `NIL` if none.

RESULT CODES

`noErr`	0	No error

InitApplZone

The Process Manager calls the `InitApplZone` procedure indirectly when it starts up your application. You should never need to call it. It is documented for completeness only.

```
PROCEDURE InitApplZone;
```

DESCRIPTION

The `InitApplZone` procedure initializes the application heap zone and makes it the current zone. The Memory Manager discards the contents of any previous application zone and discards all previously existing blocks in that zone. The procedure sets the zone's grow-zone function to `NIL`.

▲ **WARNING**
Reinitializing the application zone from within a running program is dangerous, because the application's code itself normally resides in the application zone. To do so safely, you must make sure that the code containing the `InitApplZone` call is not in the application zone. ▲

SPECIAL CONSIDERATIONS

You should not call `InitApplZone` at all, but, if you must, be sure not to call it at interrupt time because it could purge and allocate memory.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for `InitApplZone` are

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error

SetApplBase

The Process Manager calls the `SetApplBase` procedure when it starts up your application. You should never need to call it. It is documented for completeness only.

```
PROCEDURE SetApplBase (startPtr: Ptr);
```

`startPtr` The starting address for the application heap zone to be initialized.

DESCRIPTION

The `SetApplBase` procedure sets the starting address of the application heap zone for the application being initialized to the address designated by `startPtr`, and then calls the `InitApplZone` procedure.

▲ **WARNING**
Like `InitApplZone`, `SetApplBase` is a potentially dangerous operation, because the program's code itself normally resides in the application heap zone. To do so safely, you must make sure that the code containing the `SetApplBase` call is not in the application zone. ▲

SPECIAL CONSIDERATIONS

You should not call `SetApplBase` at all, but, if you must, be sure not to call it at interrupt time because it affects memory.

ASSEMBLY-LANGUAGE INFORMATION

The registers on exit for `SetApplBase` are

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error

Application-Defined Routines

The Memory Manager provides a means for you to intervene in its otherwise automatic operations by allowing you to define a grow-zone function and a purge-warning procedure.

Note
Many applications use a grow-zone function as part of a general strategy to prevent low-memory situations. Most applications, however, do not need to use purge-warning procedures. ◆

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`. See the chapter "Memory Management Utilities" in this book for a description of these two functions.

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 the chapter "Introduction to Memory Management" in this book for a definition of a sample grow-zone function.

Purge-Warning Procedures

The Memory Manager calls your application's purge-warning procedure whenever it is about to purge a relocatable block from your application heap.

MyPurgeProc

A purge-warning procedure should have the following form:

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

`h` A handle to the block that is about to be purged.

DESCRIPTION

Whenever the Memory Manager needs to purge a block from the application heap, it first calls any application-defined purge-warning procedure that you have installed. The purge-warning procedure can, if necessary, save the contents of that block or otherwise respond to the warning.

Your purge-warning procedure is called during a memory-allocation request. As a result, you should not call any routines that might cause memory to be moved or purged. In particular, if you save the data of the block in a file, the file should already be open when your purge-warning procedure is called, and you should write the data synchronously.

You should not dispose of or change the purgeable status of the block whose handle is passed to your procedure.

To install a purge-warning procedure, you need to assign its address to the `purgeProc` field of the associated zone header.

Note
If you call the Resource Manager procedure `SetResPurge` with the parameter `TRUE`, any existing purge-warning procedure is replaced by a purge-warning procedure installed by the Resource Manager. You can execute both warning procedures by calling `SetResPurge`, saving the existing value of the `purgeProc` field of the zone header, and then reinstalling your purge-warning procedure. Your purge-warning procedure should call the Resource Manager's purge-warning procedure internally. ◆

SPECIAL CONSIDERATIONS

Your purge-warning procedure 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 purge-warning procedure and the setting and restoring of the A5 register might have to be placed in separate procedures.

Your purge-warning procedure is called for *every* handle that is about to be purged (not necessarily for every purgeable handle in your heap, however). Your procedure should be able to determine quickly whether the handle it is passed is one whose associated data needs to be saved or otherwise processed.

SEE ALSO

See "Installing a Purge-Warning Procedure" on page 2-16 for a definition of a sample purge-warning procedure and for instructions on installing the procedure.

Summary of the Memory Manager

Pascal Summary

Constants

```
CONST
   {Gestalt constants}
   gestaltOSAttr              = 'os  ';   {O/S attributes}
   gestaltTempMemSupport      = 4;        {temp memory support present}
   gestaltRealTempMemory      = 5;        {temp memory handles are real}
   gestaltTempMemTracked      = 6;        {temp memory handles tracked}

   maxSize                    = $800000;  {maximum size of a block}
```

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}

   Str255               = STRING[255];    {Pascal string}
   StringPtr            = ^Str255;
   StringHandle         = ^StringPtr;

   ProcPtr              = Ptr;            {procedure pointer}

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

```
Zone =
RECORD
   bkLim:         Ptr;              {first usable byte after zone}
   purgePtr:      Ptr;              {used internally}
   hFstFree:      Ptr;              {first free master pointer}
   zcbFree:       LongInt;          {number of free bytes}
   gzProc:        ProcPtr;          {grow-zone function}
   moreMast:      Integer;          {number of master ptrs to allocate}
   flags:         Integer;          {used internally}
   cntRel:        Integer;          {reserved}
   maxRel:        Integer;          {reserved}
   cntNRel:       Integer;          {reserved}
   maxNRel:       Integer;          {reserved}
   cntEmpty:      Integer;          {reserved}
   cntHandles:    Integer;          {reserved}
   minCBFree:     LongInt;          {reserved}
   purgeProc:     ProcPtr;          {purge-warning procedure}
   sparePtr:      Ptr;              {used internally}
   allocPtr:      Ptr;              {used internally}
   heapData:      Integer;          {first usable byte in zone}
END;

THz = ^Zone;                        {zone pointer}
```

Memory Manager Routines

Setting Up the Application Heap

```
PROCEDURE MaxApplZone;
PROCEDURE MoreMasters;
```

Allocating and Releasing Relocatable Blocks of Memory

```
FUNCTION NewHandle          (logicalSize: Size): Handle;
FUNCTION NewHandleSys       (logicalSize: Size): Handle;
FUNCTION NewHandleClear     (logicalSize: Size): Handle;
FUNCTION NewHandleSysClear  (logicalSize: Size): Handle;
FUNCTION NewEmptyHandle     : Handle;
FUNCTION NewEmptyHandleSys  : Handle;
PROCEDURE DisposeHandle     (h: Handle);
```

Allocating and Releasing Nonrelocatable Blocks of Memory

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

Changing the Sizes of Relocatable and Nonrelocatable Blocks

```
FUNCTION GetHandleSize      (h: Handle): Size;
PROCEDURE SetHandleSize     (h: Handle; newSize: Size);
FUNCTION GetPtrSize         (p: Ptr): Size;
PROCEDURE SetPtrSize        (p: Ptr; newSize: Size);
```

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);
PROCEDURE HSetRBit          (h: Handle);
PROCEDURE HClrRBit          (h: Handle);
```

Managing Relocatable Blocks

```
PROCEDURE EmptyHandle       (h: Handle);
PROCEDURE ReallocateHandle  (h: Handle; logicalSize: Size);
FUNCTION RecoverHandle      (p: Ptr): Handle;
PROCEDURE ReserveMem        (cbNeeded: Size);
PROCEDURE ReserveMemSys     (cbNeeded: Size);
PROCEDURE MoveHHi           (h: Handle);
PROCEDURE HLockHi           (h: Handle);
```

Manipulating Blocks of Memory

```
PROCEDURE BlockMove         (sourcePtr, destPtr: Ptr; byteCount: Size);
FUNCTION PtrToHand          (srcPtr: Ptr; VAR dstHndl: Handle;
                             size: LongInt): OSErr;
FUNCTION PtrToXHand         (srcPtr: Ptr; dstHndl: Handle; size: LongInt):
                             OSErr;
```

```
FUNCTION HandToHand         (VAR theHndl: Handle): OSErr;
FUNCTION HandAndHand        (aHndl, bHndl: Handle): OSErr;
FUNCTION PtrAndHand         (pntr: Ptr; hndl: Handle; size: LongInt): OSErr;
```

Assessing Memory Conditions

```
FUNCTION FreeMem            : LongInt;
FUNCTION FreeMemSys         : LongInt;
FUNCTION MaxBlock           : LongInt;
FUNCTION MaxBlockSys        : LongInt;
PROCEDURE PurgeSpace        (VAR total: LongInt; VAR contig: LongInt);
FUNCTION StackSpace         : LongInt;
FUNCTION MemError           : OSErr;
```

Freeing Memory

```
FUNCTION CompactMem         (cbNeeded: Size): Size;
FUNCTION CompactMemSys      (cbNeeded: Size): Size;
PROCEDURE PurgeMem          (cbNeeded: Size);
PROCEDURE PurgeMemSys       (cbNeeded: Size);
FUNCTION MaxMem             (VAR grow: Size): Size;
FUNCTION MaxMemSys          (VAR grow: Size): Size;
```

Grow-Zone Operations

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

Allocating Temporary Memory

```
FUNCTION TempNewHandle      (logicalSize: Size; VAR resultCode: OSErr):
                             Handle;
FUNCTION TempFreeMem        : LongInt;
FUNCTION TempMaxMem         (VAR grow: Size): Size;
```

Accessing Heap Zones

```
FUNCTION GetZone            : THz;
PROCEDURE SetZone           (hz: THz);
FUNCTION ApplicationZone    : THz;
FUNCTION SystemZone         : THz;
FUNCTION HandleZone         (h: Handle): THz;
FUNCTION PtrZone            (p: Ptr): THz;
```

Manipulating Heap Zones

```
FUNCTION GetApplLimit       : Ptr;
PROCEDURE SetApplLimit      (zoneLimit: Ptr);
FUNCTION TopMem             : Ptr;
PROCEDURE InitZone          (pGrowZone: ProcPtr; cMoreMasters: Integer;
                             limitPtr, startPtr: Ptr);
PROCEDURE InitApplZone;
PROCEDURE SetApplBase       (startPtr: Ptr);
```

Application-Defined Routines

Grow-Zone Functions

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

Purge-Warning Procedures

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

C Summary

Constants

```
/*Gestalt constants*/
#define gestaltOSAttr            'os  ';     /*O/S attributes*/
#define gestaltTempMemSupport    4;          /*temp memory support present*/
#define gestaltRealTempMemory    5;          /*temp memory handles are real*/
#define gestaltTempMemTracked    6;          /*temp memory handles tracked*/

#define maxSize                  0x800000;   /*maximum size of a block*/
```

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 unsigned char Str255[256];     /*Pascal string*/
typedef unsigned char *StringPtr;
typedef unsigned char **StringHandle;

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

struct Zone {
   Ptr               bkLim;            /*first usable byte after zone*/
   Ptr               purgePtr;         /*used internally*/
   Ptr               hFstFree;         /*first free master pointer*/
   long              zcbFree;          /*number of free bytes*/
   GrowZoneProcPtr   gzProc;           /*grow-zone function*/
   short             moreMast;         /*number of master ptrs to allocate*/
   short             flags;            /*used internally*/
   short             cntRel;           /*reserved*/
   short             maxRel;           /*reserved*/
   short             cntNRel;          /*reserved*/
   short             maxNRel;          /*reserved*/
   short             cntEmpty;         /*reserved*/
   short             cntHandles;       /*reserved*/
   long              minCBFree;        /*reserved*/
   ProcPtr           purgeProc;        /*purge-warning procedure*/
   Ptr               sparePtr;         /*used internally*/
   Ptr               allocPtr;         /*used internally*/
   short             heapData;         /*first usable byte in zone*/
};
typedef struct Zone Zone;
typedef Zone *THz;                     /*zone pointer*/
```

Memory Manager Routines

Setting Up the Application Heap

```
pascal void MaxApplZone     (void);
pascal void MoreMasters     (void);
```

Allocating and Releasing Relocatable Blocks of Memory

```
pascal Handle NewHandle     (Size byteCount);
pascal Handle NewHandleSys  (Size byteCount);
pascal Handle NewHandleClear(Size byteCount);
```

```
pascal Handle NewHandleSysClear
                            (Size byteCount);
pascal Handle NewEmptyHandle(void);
pascal Handle NewEmptyHandleSys
                            (void);
pascal void DisposeHandle   (Handle h);
```

Allocating and Releasing Nonrelocatable Blocks of Memory

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

Changing the Sizes of Relocatable and Nonrelocatable Blocks

```
pascal Size GetHandleSize   (Handle h);
pascal void SetHandleSize   (Handle h, Size newSize);
pascal Size GetPtrSize      (Ptr p);
pascal void SetPtrSize      (Ptr p, Size newSize);
```

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);
pascal void HSetRBit        (Handle h);
pascal void HClrRBit        (Handle h);
```

Managing Relocatable Blocks

```
pascal void EmptyHandle     (Handle h);
pascal void ReallocateHandle(Handle h, Size byteCount);
pascal Handle RecoverHandle (Ptr p);
pascal void ReserveMem      (Size cbNeeded);
pascal void ReserveMemSys   (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);
pascal OSErr PtrToHand      (Ptr srcPtr, Handle *dstHndl, long size);
pascal OSErr PtrToXHand     (Ptr srcPtr, Handle dstHndl, long size);
pascal OSErr HandToHand     (Handle *theHndl);
pascal OSErr HandAndHand    (Handle hand1, Handle hand2);
pascal OSErr PtrAndHand     (Ptr ptr1, Handle hand2, long size);
```

Assessing Memory Conditions

```
pascal long FreeMem         (void);
pascal long FreeMemSys      (void);
pascal long MaxBlock        (void);
pascal long MaxBlockSys     (void);
pascal void PurgeSpace      (long *total, long *contig);
pascal long StackSpace      (void);
#define MemError()          (* (OSErr*) 0x0220)
```

Freeing Memory

```
pascal Size CompactMem      (Size cbNeeded);
pascal Size CompactMemSys   (Size cbNeeded);
pascal void PurgeMem        (Size cbNeeded);
pascal void PurgeMemSys     (Size cbNeeded);
pascal Size MaxMem          (Size *grow);
pascal Size MaxMemSys       (Size *grow);
```

Grow-Zone Operations

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

Allocating Temporary Memory

```
pascal Handle TempNewHandle (Size logicalSize, OSErr *resultCode);
pascal long TempFreeMem     (void);
pascal Size TempMaxMem      (Size *grow);
```

Accessing Heap Zones

```
pascal THz GetZone          (void);
pascal void SetZone         (THz hz);
#define ApplicationZone()   (* (THz*) 0x02AA)
#define SystemZone()        (* (THz*) 0x02A6)
pascal THz HandleZone       (Handle h);
pascal THz PtrZone          (Ptr p);
```

Manipulating Heap Zones

```
#define GetApplLimit()      (* (Ptr*) 0x0130)
pascal void SetApplLimit    (void *zoneLimit);
#define TopMem()            (* (Ptr*) 0x0108)
pascal void InitZone        (GrowZoneProcPtr pgrowZone, short cmoreMasters,
                             void *limitPtr, void *startPtr);
pascal void InitApplZone    (void);
pascal void SetApplBase     (void *startPtr);
```

Application-Defined Routines

Grow-Zone Functions

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

Purge-Warning Procedures

```
pascal void MyPurgeProc     (Handle h);
```

Assembly-Language Summary

Constants

```
;flags in trap words
CLEAR             EQU      $200     ;set all bytes in block to 0
SYS               EQU      $400     ;use the system heap

;values for the tag byte of a block header
tyBkFree          EQU      0        ;free block
tyBkNRel          EQU      1        ;nonrelocatable block
tyBkRel           EQU      2        ;relocatable block
```

```
;flags for the high-order byte of a 24-bit master pointer
lock              EQU      7        ;lock bit
purge             EQU      6        ;purge bit
resource          EQU      5        ;resource bit
```

Data Structures

Zone Data Structure

0	`bkLim`	long	pointer to first usable byte after zone
4	`purgePtr`	long	used internally
8	`hFstFree`	long	first free master pointer
12	`zcbFree`	4 bytes	number of free bytes in zone
16	`gzProc`	long	grow-zone function
20	`mAllocCnt`	word	number of master pointers to allocate
22	`flags`	word	used internally
24	`cntRel`	word	reserved
26	`maxRel`	word	reserved
28	`cntNRel`	word	reserved
30	`maxNRel`	word	reserved
32	`cntEmpty`	word	reserved
34	`cntHandles`	word	reserved
36	`minCBFree`	long	reserved
40	`purgeProc`	long	purge-warning procedure
44	`sparePtr`	long	used internally
48	`allocPtr`	long	used internally
52	`heapData`	word	first usable byte in zone

Parameter Block for `InitZone` Procedure

0	`startPtr`	long	first byte of new zone
4	`limitPtr`	long	first byte beyond new zone
8	`cMoreMasters`	word	number of master pointers to be allocated at a time
10	`pGrowZone`	long	pointer to grow-zone function for new zone

Trap Macros

Trap Macro Names

Pascal name	Trap macro name
`BlockMove`	`_BlockMove`
`CompactMem`	`_CompactMem`
`CompactMemSys`	`_CompactMem`
`DisposeHandle`	`_DisposeHandle`
`DisposePtr`	`_DisposePtr`

Pascal name	Trap macro name
EmptyHandle	_EmptyHandle
FreeMem	_FreeMem
FreeMemSys	_FreeMem
GetHandleSize	_GetHandleSize
GetPtrSize	_GetPtrSize
GetZone	_GetZone
HandAndHand	_HandAndHand
HandleZone	_HandleZone
HandToHand	_HandToHand
HClrRBit	_HClrRBit
HGetState	_HGetState
HLock	_HLock
HNoPurge	_HNoPurge
HPurge	_HPurge
HSetRBit	_HSetRBit
HSetState	_HSetState
HUnlock	_HUnlock
InitApplZone	_InitApplZone
InitZone	_InitZone
MaxApplZone	_MaxApplZone
MaxBlock	_MaxBlock
MaxBlockSys	_MaxBlock
MaxMem	_MaxMem
MaxMemSys	_MaxMem
MoreMasters	_MoreMasters
MoveHHi	_MoveHHi
NewEmptyHandle	_NewEmptyHandle
NewEmptyHandleSys	_NewEmptyHandle
NewHandle	_NewHandle
NewHandleClear	_NewHandle
NewHandleSys	_NewHandle
NewHandleSysClear	_NewHandle
NewPtr	_NewPtr
NewPtrClear	_NewPtr
NewPtrSys	_NewPtr
NewPtrSysClear	_NewPtr

Pascal name	Trap macro name
PtrAndHand	_PtrAndHand
PtrToHand	_PtrToHand
PtrToXHand	_PtrToXHand
PtrZone	_PtrZone
PurgeMem	_PurgeMem
PurgeMemSys	_PurgeMem
PurgeSpace	_PurgeSpace
ReallocateHandle	_ReallocHandle
RecoverHandle	_RecoverHandle
ReserveMem	_ResrvMem
ReserveMemSys	_ResrvMem
SetApplBase	_SetApplBase
SetApplLimit	_SetApplLimit
SetGrowZone	_SetGrowZone
SetHandleSize	_SetHandleSize
SetPtrSize	_SetPtrSize
SetZone	_SetZone
StackSpace	_StackSpace

Trap Macro Requiring Routine Selectors

_OSDispatch

Selector	Routine
$0015	TempMaxMem
$0018	TempFreeMem
$001D	TempNewHandle

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.
CurStackBase	long	Address of base of stack; start of application global variables.
GZRootHnd	long	A handle to a block that the grow-zone function must not move.
HeapEnd	long	Address of end of application heap zone.
MemErr	word	The current value that MemError would return.
MemTop	long	After startup time, the address at the end of an application's partition.
SysZone	long	A pointer to the system heap zone.
TheZone	long	A pointer to the current heap zone.

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