This chapter describes the Virtual Memory Manager, the part of the Operating System that allows memory to be extended beyond the limits of the physical address space provided by the available RAM. A user can select (in the Memory control panel) whether to enable this larger or "virtual" address space.

Most applications are completely unaffected by the operation of the Virtual Memory Manager and have no need to know whether any virtual memory is available. You might, however, need to intervene in the otherwise automatic workings of the Virtual Memory Manager if your application has critical timing requirements, executes code at interrupt time, or performs debugging operations.

The Virtual Memory Manager also offers services that might be of use to software components even if virtual memory is not enabled on a particular computer. On some Macintosh computers, the physical address space is discontiguous and is therefore not identical with the logical address space. In normal operations, the Operating System uses the MMU coprocessor to map logical addresses to their corresponding physical addresses. In some cases, however, you might need to perform this address mapping yourself. For example, if you are writing software that runs in the Macintosh Operating System but communicates addresses to NuBus™ expansion cards with bus master or direct memory access (DMA) capabilities, you need to pass physical and not logical addresses. You can use the Virtual Memory Manager to determine those physical addresses.

To use this chapter, you should be familiar with the normal operation of the Memory Manager, as described in the chapter "Introduction to Memory Management" in this book. If your application or other software executes code at interrupt time, you should also be familiar with the process of scheduling interrupt code, as described in the chapter "Introduction to Processes and Tasks" in *Inside Macintosh: Processes*.

This chapter begins with a description of how the Virtual Memory Manager provides virtual memory. It explains how the logical and physical address spaces are mapped to one another and when you might need to use the services provided by the Virtual Memory Manager. Then it explains how you can use the Virtual Memory Manager to

- make portions of the logical address space resident in physical RAM
- make portions of the logical address space immovable in physical RAM
- map logical to physical addresses
- defer execution of application-defined interrupt code until a safe time

This chapter also provides information about a number of routines that are useful only for the implementation of debuggers that operate under virtual memory.

About the Virtual Memory Manager

The Virtual Memory Manager is the part of the Operating System that provides **virtual memory,** addressable memory beyond the limits of the available physical RAM. The principal benefit of using virtual memory is that a user can run more applications at once

and work with larger amounts of data than would be possible if the logical address space were limited to the available RAM. Instead of equipping a computer with amounts of RAM large enough to handle all possible needs, the user can install only enough RAM to meet average needs. Then, during those occasional times when more memory is needed for large tasks or many applications, the user can take advantage of virtual memory. When virtual memory is present, the perceived amount of RAM can be extended to as much as 14 MB on systems with 24-bit addressing and as much as 1 GB on systems with 32-bit addressing.

The Virtual Memory Manager also provides a number of routines that your software can use to modify or get information about its operations. You can use the Virtual Memory Manager to

- hold portions of the logical address space in physical RAM
- lock portions of the logical address space in their physical RAM locations
- determine whether a particular portion of the logical address space is currently in physical RAM
- determine, from a logical address, the physical address of a block of memory

This section describes how the Virtual Memory Manager provides virtual memory. It also explains why you might need to use certain Virtual Memory Manager routines even when virtual memory is not available.

Virtual Memory

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

For the most part, the Virtual Memory Manager operates invisibly to applications and to the user. Most applications do not need to know whether virtual memory is installed unless they have critical timing requirements, execute code at interrupt time, or perform debugging operations. The only time that users need to know about virtual memory is when they configure it in the Memory control panel. One visible cost of this extra memory is the use of an equivalent amount of storage on a storage device, such as a SCSI hard disk. Another cost of using virtual memory is a possible perception of sluggishness as paged-out segments of memory are pulled back into physical memory. Performance degradation due to the use of virtual memory ranges from unnoticeable to severe, depending on the ratio of virtual memory to physical RAM and the behavior of the actual applications running.

There are two main requirements for running virtual memory. First, the computer must be running system software version 7.0 or later. Second, the computer must be equipped with an **MMU** or **PMMU** coprocessor. Apple's 68040- and 68030-based machines have an MMU built into the CPU and are ready to run virtual memory with no additional hardware. A Macintosh II (68020-based) computer can take advantage of virtual memory if it has the 68851 PMMU coprocessor on its main logic board in place of the standard **Address Management Unit (AMU).** (The PMMU is the same coprocessor needed to run A/UX.) Apple's 68000-based machines cannot take advantage of virtual memory.

Users control and configure virtual memory through the Memory control panel. Controls in this panel allow the user to turn virtual memory on or off, set the size of virtual memory, and set the volume on which the invisible backing-store file resides. (The **backing-store file** is the file in which the Operating System stores the contents of nonresident pages of memory.) Other memory-related user controls appear in this control panel. These include settings for the disk cache and for 24-bit or 32-bit Memory Manager addressing. If users change the virtual memory, addressing, or disk cache settings, they must restart the computer for the changes to take effect.

The virtual memory setting in the control panel reflects the total amount of memory available to the system (and not simply the amount of memory to be added to available RAM). Also, the backing-store file is as large as the amount of virtual memory. This backing-store file can be located on any HFS volume that allows block-level access. (This volume is known as the **paging device** or **backing volume.**) Because the paging device must support block-level access, users cannot select as the paging device a volume mounted through AppleShare. Also, users cannot select removable disks, including floppy disks, as paging devices.

The Logical Address Space

When virtual memory is present, the logical address space is larger than the physical address space provided by the available RAM. The actual size of the logical address space, and hence the amount of virtual memory, depends on a number of factors, including

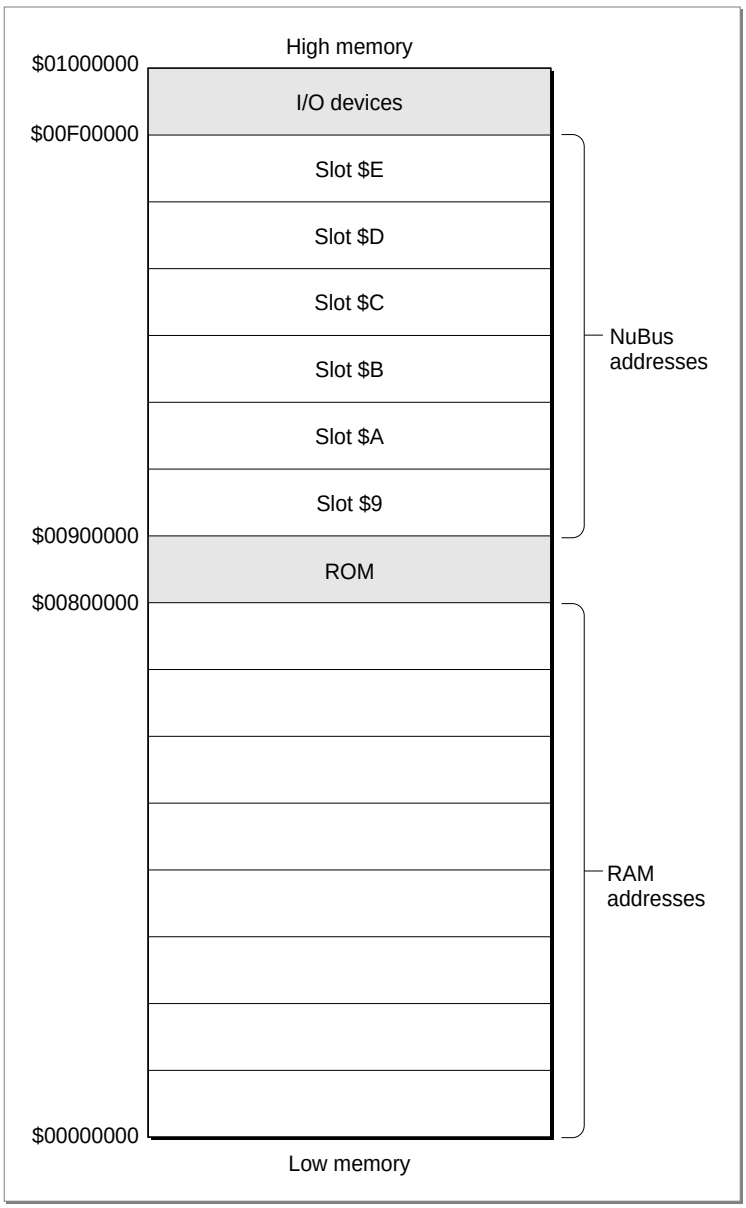
- the addressing mode currently used by the Memory Manager
- the amount of space available on a secondary storage device for use by the backing-store file
- if 24-bit addressing is in operation, the number of NuBus expansion cards, if any, installed in the computer

24-Bit Addressing

When running with **24-bit addressing,** the Memory Manager can address at most 2^{24} bytes, or 16 MB. Of these 16 MB, at most 8 MB can be used to address physical RAM. The remaining 8 MB are devoted to ROM addresses, I/O device addresses, and NuBus slot addresses. Figure 3-1 illustrates the logical address space mapping used by the 24-bit Memory Manager.

Note
In some Macintosh computers, the ROM is mapped to the address range $01000000 to $010FFFFF (indicated as belonging to slot $A in Figure 3-1). In these computers, the maximum amount of physical RAM is 10 MB instead of 8 MB. The remainder of this section describes the original layout of the 24-bit logical address space only. ◆

Figure 3-1 24-bit Memory Manager logical address space



When 24-bit addressing is in operation and virtual memory is available, the Virtual Memory Manager uses, as part of the addressable application memory, any 1 MB segments not assigned to a NuBus card. For example, if a Macintosh computer has three NuBus expansion cards installed, that computer can address at most 11 MB of virtual memory. The maximum amount of virtual memory possible in a 24-bit environment is 14 MB (that is, 8 MB of physical RAM + 6 MB of additional space previously reserved for the NuBus); this maximum is achievable only on a computer with no NuBus expansion cards installed.

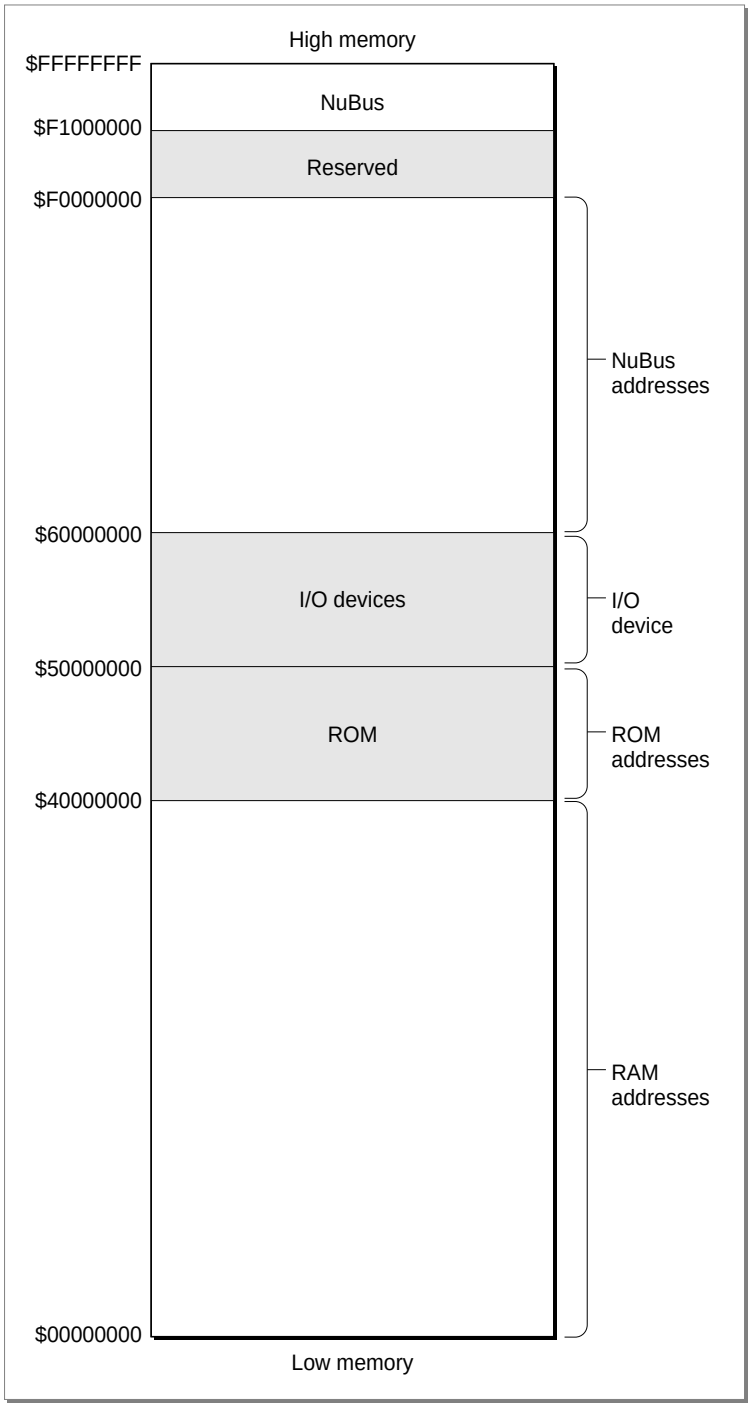
Notice in Figure 3-1 that addresses from $00800000 to $008FFFFF are reserved for ROM. In other words, the largest contiguous block of space that an application can allocate when virtual memory is available is somewhat less than 8 MB, even though the total amount of virtual memory available can be as large as 14 MB. The rest of the virtual memory can be in a contiguous block as large as 4 or 5 MB, unless the user has fragmented the NuBus space by making a poor choice of slots in which to install expansion cards. To maximize the amount of contiguous virtual memory, users should place cards in consecutive slots at either end of the expansion bus. A haphazard placement of NuBus cards may result in a number of 1 MB or 2 MB "islands" in the upper portion of the 24-bit address space; in general, this kind of fragmentation reduces the effectiveness of a large virtual address space.

Note
Some Macintosh computers have fewer than six NuBus slots, and the numbering of the slots is not consistent across different models. In a Macintosh IIcx, the three available slots are numbered $9 through $B, so expansion cards should be grouped toward the lowest-numbered slot (contiguous with the ROM space). In a Macintosh IIci, the slots are numbered $C through $E, so expansion cards should be grouped toward the highest-numbered slot (contiguous with the I/O space). However, the RAM-based video on the Macintosh IIci occupies addresses reserved for slot $B; as a result, it is impossible to avoid some degree of fragmentation of the virtual address space when you use the RAM-based video option on that computer. ◆

32-Bit Addressing

When running with **32-bit addressing,** the Memory Manager can address at most 2^{32} bytes, or 4 GB. Of these 4 GB, at most 1 GB can be used to address physical RAM. The remaining 3 GB are devoted to ROM addresses, I/O device addresses, and NuBus slot addresses. Figure 3-2 illustrates the logical address space mapping used by the 32-bit Memory Manager.

Figure 3-2 32-bit Memory Manager logical address space



Note
The fragmentation of the virtual address space that sometimes occurs when 24-bit addressing is in operation is never a problem when 32-bit addressing is in operation. In the 32-bit address space, virtual memory and the NuBus slots do not share space. ◆

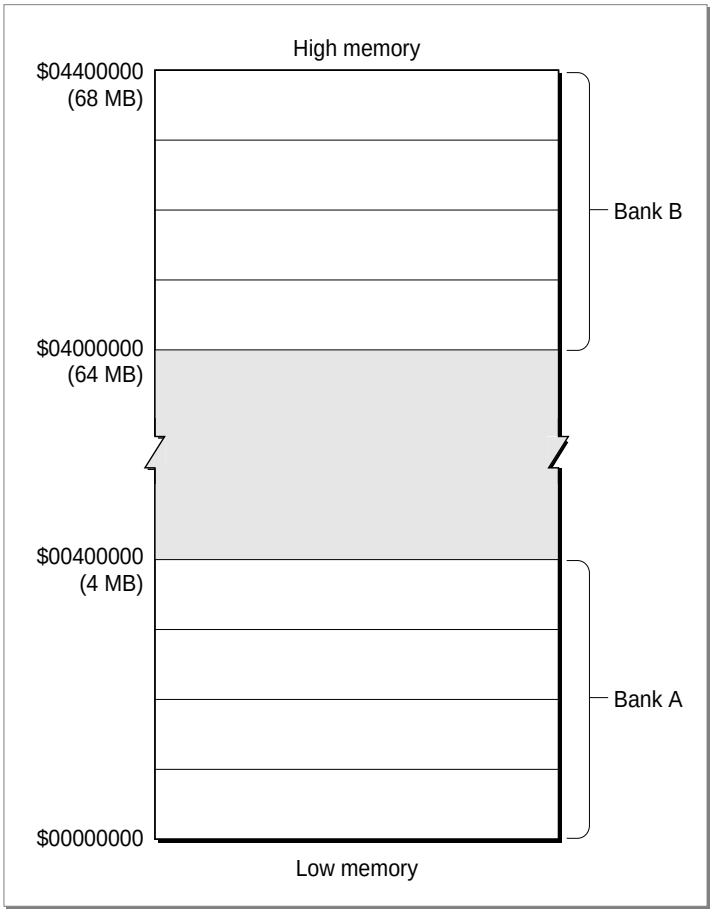
The Physical Address Space

The original versions of the Macintosh Operating System used physical addresses exclusively. A particular location in RAM could be accessed by its physical address, regardless of whether that address was generated by an application, by the system software, or even by a NuBus expansion card. In short, there was no difference between the logical and the physical address spaces.

However, both hardware and software advances have forced the Operating System to abstract the logical address space from the physical address space. As you have seen, the logical address space is larger than the physical address space when virtual memory is available. The Operating System uses the MMU coprocessor to map logical addresses to their corresponding physical addresses.

In addition, some Macintosh computers have a discontiguous physical address space. For example, on a Macintosh IIci with 8 MB of physical RAM, the physical memory appears to the CPU and to the NuBus expansion bus as two separate 4 MB ranges (see Figure 3-3). As you can see, the physical RAM occupies two separate ranges: the RAM installed in bank A, ranging from $00000000 to $003FFFFF, and the RAM installed in bank B, ranging from $04000000 to $043FFFFF.

Figure 3-3 The physical address space on a Macintosh IIci with 8 MB of RAM



In most cases, a discontiguous physical address space causes no problems, because the Operating System uses the MMU coprocessor to map the available physical memory into a single contiguous logical address space. All memory addresses returned to your application by the Memory Manager (for instance, when you allocate a new block by calling `NewHandle`) are logical addresses. When you read or write a logical memory address, the Operating System uses the MMU coprocessor to determine the physical address corresponding to your logical address. This address translation is completely transparent to your application. For example, if you read the system global variable located at address $10C, it doesn't matter that the CPU actually looks at the physical address $0400010C.

In some cases, however, you can run into problems if you don't account for the possibility that the logical address space and the physical address space might differ. Suppose, for instance, that you are developing a driver that passes addresses to NuBus master hardware. In this case, you need to take care to pass it *physical* addresses only, because NuBus hardware does not use the MMU to translate logical addresses into physical addresses. If your driver passes a logical address, the NuBus hardware cannot translate it into a physical address because it does not have access to the MMU's address-mapping tables. If your hardware then attempts to write data to that address, it is likely to overwrite some other portion of physical memory.

To prevent this problem, you need to make certain that you always convert logical addresses to their corresponding physical addresses before you pass those addresses to any alternate bus master. You can do this by calling the `GetPhysical` function, as described later in "Mapping Logical to Physical Addresses," which begins on page 3-16. The `GetPhysical` function is implemented in ROM on all machines that have a discontiguous physical address space—whether or not virtual memory is available. Accordingly, before you pass addresses to an alternate bus master, you should check for the availability of the `GetPhysical` call; if it's available, you should use it to translate logical to physical addresses.

Note
Passive or slave NuBus cards (such as video cards) that do not read or write physical RAM are not likely to be affected by the presence of virtual memory or by a discontiguous physical address space. ◆

Page Faults

When an application or other software component tries to access data in a page of memory that is not currently resident in RAM, the Operating System issues a special kind of bus error known as a **page fault.** The Virtual Memory Manager intercepts page faults and tries to load the affected page or pages into memory. It does so by executing its own internal page-fault handler, which handles page faults and passes other bus errors to the standard bus-error vector in low memory.

To load the required pages into memory, the Virtual Memory Manager's page-fault handler takes over the SCSI bus and makes calls directly to the driver of the backing-store file. While the Virtual Memory Manager is handling a page fault, it is essential that no other page faults occur. If a page fault did occur during page-fault handling—a condition known as a **double page fault**—the Virtual Memory Manager would have to interrupt the driver of the paging device to make a further request to load the needed page. Unless the driver of the paging device is **concurrent** (that is, able to handle several requests at once), the driver cannot handle this second request. Unfortunately, current versions of most SCSI disk drivers are not concurrent. As a result, a double page fault results in a system crash.

The Virtual Memory Manager takes special steps to avoid double page faults caused by user code (that is, code that is not executed as the result of an exception). It defers all

user code while the driver of the paging device is busy. In particular, the Virtual Memory Manager defers until a safe time the following types of code:

- VBL tasks
- Slot-based VBL tasks
- Time Manager tasks
- I/O completion routines

Note
Because these types of tasks may be deferred under virtual memory, any application or device driver that uses them to achieve real-time performance might be adversely affected by the operation of the Virtual Memory Manager. ◆

Other software components must take care not to cause page faults at interrupt time. In particular, device drivers, which commonly run at interrupt time, should make certain that any data structures or buffers that they reference at interrupt time are in physical memory at that time. You can make sure that this happens by holding the required data in physical memory, as described in "Holding and Releasing Memory" on page 3-14.

In an effort to maintain compatibility with existing drivers, the Operating System automatically keeps the entire system heap in physical memory at all times. Therefore, if your device driver and its associated data structures are loaded into the system heap, you do not need to worry about causing page faults at interrupt time.

▲ **WARNING**
Future versions of the system software are not guaranteed to keep the entire system heap in physical memory. To be safe, you should explicitly hold in physical memory any code or data that you know might be accessed at interrupt time. ▲

The Virtual Memory Manager provides this further level of protection against page faults caused by device drivers at interrupt time: it automatically holds in physical memory any buffers used by the Device Manager `_Read` and `_Write` operations. Any driver that uses the `_Read` and `_Write` calls to move data between main memory and the driver's associated hardware device is therefore automatically compatible with virtual memory. If, however, you use `_Status` or `_Control` calls to move data at interrupt time, you must explicitly hold or lock all buffers that are referenced in the `_Status` or `_Control` parameter block. If possible, you should rewrite your driver so that it uses `_Read` and `_Write` calls instead of `_Status` and `_Control` calls to move data.

The Virtual Memory Manager provides one other routine that you can use to help prevent double page faults. If your application or other code installs interrupt routines other than those handled automatically by the Virtual Memory Manager (such as VBL tasks, Time Manager tasks, and Device Manager completion calls), you can explicitly defer the execution of the routine by calling it via the function `DeferUserFn`. See "Deferring User Interrupt Handling" on page 3-20 for details on calling `DeferUserFn`.

Using the Virtual Memory Manager

The routines described in this section allow drivers and applications with critical timing needs to intervene in the otherwise automatic workings of the Virtual Memory Manager's paging mechanism.

Note
The vast majority of applications do not need to use these routines. They are used primarily by drivers, debuggers, and other interrupt-servicing code. ◆

If necessary, your software can request that a range of memory be held in physical memory. **Holding** means that the specified memory range cannot be paged out to disk, although it might be moved within physical RAM. As a result, no page faults can result from reading or writing memory addresses of pages that are held in memory.

Similarly, a page or range of pages can be locked in physical memory. **Locking** means that the specified memory cannot be paged out to disk and that the memory cannot change its real (physical) RAM location. You can also request that a range of pages be locked in a contiguous range of physical memory, although contiguity is not guaranteed. The need to lock pages in a contiguous area of memory arises primarily when external hardware transfers data directly into physical RAM. In this case, locking might be useful for keeping a contiguous range of memory stationary during operations of an external CPU (on a NuBus card, for example) that cannot support a DMA action.

Most applications do not need to hold or lock pages in physical RAM. The Virtual Memory Manager usually works quickly enough that your application is not affected by any delay that might result from paging. Device drivers or sound and animation applications with critical timing requirements usually need only to hold memory, not lock it. Here are some general rules regarding when to hold or lock memory:

- Avoid executing tasks that could cause page faults at interrupt time. The less work done at interrupt time, the better for all applications running.
- You cannot hold or lock memory (or call any Memory Manager routines that move or purge memory) at interrupt time.
- Don't lock or hold everything in RAM. Sometimes you do need to hold or lock pages in RAM, but if you are in doubt, then probably you need to do neither.
- Your application must explicitly release or unlock whatever it held or locked. If for some reason an area of RAM is held and locked, or held twice, then it must be released and unlocked, or released twice.

The last directive is especially important. Your application is responsible for undoing the effects of locking or holding ranges of memory. In particular, the Virtual Memory Manager does not automatically unlock pages that have been locked. If you do not undo these effects in a timely fashion, you are likely to degrade performance. In the worst case, you could cause the system to run out of physical memory.

Obtaining Information About Virtual Memory

You should always determine whether virtual memory is available before attempting to use any Virtual Memory Manager routines. To do this, pass the `Gestalt` function the `gestaltVMAttr` selector. The `Gestalt` function's response indicates the version of virtual memory, if any, installed. If bit 0 of the response is set to 1, then the system software version 7.0 implementation of virtual memory is installed.

Note
Sometimes you don't need to check whether virtual memory is actually available before calling some Virtual Memory Manager routines. For example, you might need to call the `GetPhysical` function even if virtual memory is not enabled. Instead of calling `Gestalt` to see whether virtual memory is available, you should simply test whether the appropriate trap is available. In the case of the `GetPhysical` function, you should check that the `_MemoryDispatchA0Result` trap is available. ◆

You can also use the `Gestalt` function to obtain information about the memory configuration of the system, in particular, information about the amount of physical memory installed in a computer, the amount of logical memory available in a computer, the version of virtual memory installed (if any), and the size of a logical page. By obtaining this information from `Gestalt`, you can help insulate your applications or drivers from possible future changes in the details of the virtual memory implementation.

Holding and Releasing Memory

You can use the `HoldMemory` function to make a portion of the address space resident in physical memory and ineligible for paging. This function is intended primarily for use by drivers that access user data buffers at interrupt level, whether transferring data to or from them. Calling `HoldMemory` on the appropriate memory ranges thus prevents them from causing page faults at interrupt level and effectively prevents them from generating fatal double page faults. The contents of the specified range of virtual addresses can move in physical memory, but they are guaranteed always to be in physical memory when accessed.

Note
If you use the device-level `_Read` and `_Write` functions when doing data transfers, the Virtual Memory Manager automatically ensures that the data buffers and parameter blocks are held before the transfer of data. ◆

The following sample code instructs the Virtual Memory Manager to hold in RAM an 8192-byte range of memory starting at address $32500:

```
myAddress := $32500;
myLength := 8192;
myErr := HoldMemory(myAddress, myLength);
```

Note that whole pages of the virtual address space are held, regardless of the starting address and length parameters you supply. If the starting address parameter supplied to the `HoldMemory` function is not on a page boundary, then it is rounded down to the nearest page boundary. Similarly, if the specified range does not end on a page boundary, the length parameter is rounded up so that one or more whole pages are held. This rounding might result in the holding of several pages of physical memory, even if the specified range is less than a page in length.

To release memory held as a result of a call to `HoldMemory`, you must use the `UnholdMemory` function, which simply reverses the effects of the `HoldMemory` function. For example, the page or pages held in memory in the previous example can be released as follows:

```
myErr := UnholdMemory(myAddress, myLength);
```

Like holding, releasing applies to whole pages of the virtual address space. Similar rounding of the address and length parameters is performed, as required, to make the range begin and end on page boundaries.

Note
In current versions of system software, the system heap is always held in memory and is never paged out. ◆

Locking and Unlocking Memory

You can use the `LockMemory` function to make a portion of the address space immovable in physical memory and ineligible for paging. The Operating System may move the contents of the specified range of logical addresses to a more convenient location in physical memory during the locking operation, but on completion, the contents of the specified range of logical addresses are resident and do not move in physical memory.

Locking a range of memory is a more drastic measure than just holding it. Locking not only forces the range to be held resident in RAM but also prevents its logical address from moving with respect to its physical address. The `LockMemory` function is used by drivers and other code when hardware other than the Macintosh CPU is transferring data to or from user buffers, such as any NuBus master peripheral card or DMA hardware. This function prevents both paging and physical relocation of a specified memory area and allows the physical addresses of a memory area to be exported to the non-CPU hardware. Typically, you would use this service for the duration of a single I/O request. However, you could use this service to lock data structures that are permanently shared between a driver (or other code) and a NuBus master.

Note
Don't confuse locking address ranges in RAM (using `LockMemory`) with locking a handle (using `HLock`). A locked handle can still be paged out. ◆

The main reason to disable movement of pages in physical memory is to allow translation of virtual memory addresses to physical addresses. This translation is needed by bus masters, which must write to memory in the physical address space. To avoid stale data, the memory locked in RAM is marked as noncacheable in the MMU page tables.

You can lock a range of memory in a contiguous range of physical memory by calling the `LockMemoryContiguous` function. This function can be used by driver and NuBus master or driver and DMA hardware combinations when a non-CPU device accessing memory cannot handle physically discontiguous data transfers. You can also use this service when the transfer of physically discontiguous data would degrade performance. However, the call to `LockMemoryContiguous` may be expensive, because sometimes entire pages must be copied to make a range contiguous.

Note
It might not be possible to make a range physically contiguous if any of the pages in the range are already locked. Because a call to `LockMemoryContiguous` is not guaranteed to return the desired results, you must include in your code an alternate method for locking the necessary ranges of memory. In general, you should avoid calling `LockMemoryContiguous` if at all possible. If you must call it, do so as early as possible—preferably at system startup time—to increase the likelihood of finding enough contiguous memory. ◆

To unlock a range of previously locked pages, use the `UnlockMemory` function. This function reverses the effects of `LockMemory` or `LockMemoryContiguous`. Unlocked pages are marked as cacheable.

Locking, contiguous locking, and unlocking operations are applied to ranges of the logical address space. If necessary to force the ranges onto page boundaries, the Virtual Memory Manager performs rounding of addresses and sizes, as described in "Holding and Releasing Memory" on page 3-14.

Mapping Logical to Physical Addresses

To obtain information about page mapping between logical and physical addresses, use the `GetPhysical` function, which translates logical addresses into their corresponding physical addresses. It provides drivers and other software with the actual physical memory addresses of a specified logical address range. Non-CPU devices need this information to access memory mapped by the CPU.

The `GetPhysical` function allows you to obtain the physical addresses that correspond to any logically addressable range of main memory. To specify the logical address range to be translated, you use a **memory-block record,** defined by the `MemoryBlock` data type.

```
TYPE MemoryBlock =
RECORD
   address:    Ptr;              {start of block}
   count:      LongInt;          {size of block}
END;
```

A memory-block record identifies a single contiguous block of memory by specifying the first byte in the block and the length of the block.

Note
Don't confuse the blocks of memory defined by the `MemoryBlock` data type with memory blocks as manipulated by the Memory Manager. The portion of the logical address space to be translated by `GetPhysical` can overlap several Memory Manager memory blocks or be just a part of one. Typically, however, that range coincides with the contents of a single Memory Manager block. ◆

A single logical address range sometimes corresponds to more than one range of physical addresses. As a result, `GetPhysical` needs to pass back to your application an array of memory-block records. You pass a logical address range to `GetPhysical`, and it returns an array of physical address ranges. This operation requires the use of a logical-to-physical **translation table,** defined by the `LogicalToPhysicalTable` data type.

```
TYPE LogicalToPhysicalTable =
RECORD
   logical:    MemoryBlock;         {a logical block}
   physical:   ARRAY[0..defaultPhysicalEntryCount-1] OF
                      MemoryBlock;  {equivalent physical blocks}
END;
```

To call `GetPhysical`, you need to pass a translation table whose `logical` field specifies the logical address range you want to translate. You also need to specify how many contiguous physical address ranges you want returned. In this way, you can adjust the number of elements in the array to suit your own needs. By default, a translation table contains enough space for eight physical memory blocks.

```
CONST    defaultPhysicalEntryCount  = 8;
```

If the variable `myTable` is of type `LogicalToPhysicalTable` and `myCount` is of type `LongInt`, you can call `GetPhysical` as follows:

```
myCount := (SizeOf(myTable) DIV SizeOf(MemoryBlock)) - 1;
myErr := GetPhysical(myTable, myCount);
```

The algorithm used here to calculate the number of physical entries returned (`myCount`) allows you to change the size (and hence the type) of the `myTable` variable to include more or fewer memory blocks. The default size of the translation table is sufficient for most purposes. Before you do the translation, you can determine how many physical blocks you need to accommodate the entire logical address space specified in the table's `logical` parameter. To determine this, you pass a variable whose initial value is 0:

```
myCount := 0;        {get number of blocks needed for given range}
myErr := GetPhysical(myTable, myCount);
```

If the value of its second parameter is 0, `GetPhysical` returns in that parameter the total number of physical blocks that would be required to translate the entire logical address range. In this case, both the `logical` and `physical` fields of the translation table are unchanged.

If the value of its second parameter is not 0, `GetPhysical` returns in the `physical` field of the translation table an array specifying the physical blocks that correspond to the logical address specified in the `logical` field. The `GetPhysical` function returns in its second parameter the number of entries in that array (which may be fewer than were asked for). If the translation table was not large enough to contain all the physical blocks corresponding to the logical block, `GetPhysical` updates the fields of the `logical` memory block to reflect the remaining number of bytes in the logical range left to translate (`count` field) and the next address in the logical address range to translate (`start` field).

Note
You must lock (using `LockMemory`) the address range passed to `GetPhysical` to guarantee that the translation data returned are accurate (that is, that the logical pages do not move around in physical memory and that paging activity has not invalidated the translation data). An error is returned if you call `GetPhysical` on an address range that is not locked. ◆

Recall that you sometimes need to call `GetPhysical` even if virtual memory is not available. (See "The Physical Address Space" on page 3-9 for details.) In general, if `GetPhysical` is available in the operating environment, then you should call it any time your software exports addresses to a NuBus expansion card that can read or write physical RAM directly. Listing 3-1 defines a general algorithm for implementing driver calls to a generic NuBus master card. To maximize compatibility with virtual memory, make sure that your hardware and device drivers support this method of issuing driver calls.

Listing 3-1 Translating logical to physical addresses

```
PROGRAM GetPhysicalUsage;
USES Types, Traps, Memory, Utilities;
CONST
   kTestPtrSize = $100000;
VAR
   myPtr:            Ptr;
   myPtrSize:        LongInt;
   hasGetPhysical:   Boolean;       {does this machine have GetPhysical?}
   lockOK:           Boolean;       {was the block successfully locked?}
   myErr:            OSErr;
   myTable:          LogicalToPhysicalTable;
   myCount:          LongInt;
   index:            Integer;

   PROCEDURE SendDMACmd (addr: Ptr; count: LongInt);
      BEGIN
         {This is where you would probably make a driver call }
         { to initiate DMA from a NuBus master or similar hardware.}
      END;
BEGIN
   myPtrSize := kTestPtrSize;
   myPtr := NewPtr(myPtrSize);
   IF myPtr <> NIL THEN
   BEGIN
      hasGetPhysical := TrapAvailable(_MemoryDispatch);
      IF hasGetPhysical THEN
      BEGIN
         myErr := LockMemory(myPtr, myPtrSize);
         lockOK := (myErr = noErr);
         IF lockOK THEN
         BEGIN
            myTable.logical.address := myPtr;
            myTable.logical.count := myPtrSize;
            myErr := noErr;
            WHILE (myErr = noErr) & (myTable.logical.count <> 0) DO
            BEGIN
               myCount := SizeOf(myTable) DIV SizeOf(MemoryBlock) - 1;
               myErr := GetPhysical(myTable, myCount);
               IF myErr = noErr THEN
                  FOR index := 0 TO (myCount - 1) DO
                     WITH myTable DO
                        SendDMACmd(physical[index].address,
```

```
                                                physical[index].count)
            ELSE
            BEGIN
               {Handle GetPhysical error indicated by myErr.}
               {Loop will terminate unless myErr is reset to noErr.}
            END;
          END; {WHILE}
          {Always unlock a range you locked; ignore any error here.}
          myErr := UnlockMemory(myPtr, myPtrSize);
        END
        ELSE     {not lockOK}
        BEGIN
           {handle LockMemory error indicated by myErr}
        END;
      END
      ELSE                          {GetPhysical not available}
         SendDMACmd(myPtr, myPtrSize);
   END; {IF myPtr}
END.
```

If the `GetPhysical` function is not available, the program defined in Listing 3-1 simply calls your routine to send a DMA command to the NuBus hardware. In that case, no address translation is necessary. If, however, `GetPhysical` is available, you need to lock the logical address range whose physical addresses you want to get. If you successfully lock the range, you can call `GetPhysical` as illustrated earlier. Be sure to unlock the range you previously locked before exiting the program.

▲ **WARNING**
Some Macintosh computers contain the `_MemoryDispatch` trap in ROM, even though they do not contain an MMU coprocessor. In this case, the system software patches the `_MemoryDispatch` trap to make it appear unimplemented. However, software that executes before system patches are installed cannot use this as a test of whether to call `GetPhysical` or not. If your code is executed before the installation of system patches, you should use the `Gestalt` function to test directly for the existence of an MMU coprocessor. ▲

Deferring User Interrupt Handling

During the time that the Macintosh is handling a page fault, it is critical that no other page faults occur. Because the system performs no other work while it is handling a page fault, only code that runs as a result of an interrupt can generate a second page fault. For this reason, you must call the `HoldMemory` function on buffers or code that are to be referenced by any interrupt service routine. You must call this function at noninterrupt level because the `MemoryDispatch` calls may cause movement of logical memory or physical memory and possible I/O.

The use of procedure pointers (variables of type `ProcPtr`) in specifying I/O completion routines, socket listeners, and so forth makes it impossible for drivers to know the exact location and size of all code or buffers that might be referenced when these routines are invoked. However, these routines must still be called only at a safe time, when paging is not currently in progress. Because the locations of all needed pages cannot be known, an alternate strategy is used to prevent a fatal double page fault.

The `DeferUserFn` function is provided to allow interrupt service routines to defer, until a safe time, code that might cause page faults. This function determines whether the call can be made immediately and, if it is safe, makes the call. If a page fault is in progress, the address of the service routine and its parameter are saved, and the routine is deferred until page faults are again permitted.

Virtual Memory and Debuggers

Note
You need the information in this section only if you are writing a debugger that is to operate under virtual memory. ◆

Debuggers running under virtual memory can use any of the virtual memory routines discussed in the previous sections. For example, if a debugger is in a situation where page faulting would be fatal, it can use `DeferUserFn` to defer the debugging until paging is safe. However, debuggers running under virtual memory might require a few routines that differ from those available to other applications. In addition, debuggers might depend on some specific features of virtual memory that other applications should not depend on.

For example, because debugger code might be entered at a time when paging would be unsafe, you should lock (and not just hold) the debugger and all of its data and buffer space in memory. Normally, the locking operation is used to allow NuBus masters or other DMA devices to transfer data directly into physical memory. This requires that data caching be disabled on the locked page. You might, however, want your debugger to benefit from the performance of the data cache on pages belonging solely to the debugger. The `DebuggerLockMemory` function does exactly what `LockMemory` does, except that it leaves data caching enabled on the affected pages. You can call the `DebuggerUnlockMemory` function to reverse the effects of `DebuggerLockMemory`.

Other special debugger support functions

- determine whether paging is safe
- allow the debugger to enter supervisor mode
- enter and exit the debugging state
- obtain keyboard input while in the debugging state
- determine the state of a page of logical memory

All of these functions are implemented as extensions of `_DebugUtil`, a trap intended for use by debuggers to allow greater machine independence. This trap is not present in the Macintosh II, Macintosh IIx, Macintosh IIcx, or Macintosh SE/30 models, but it is present in all later models. The Virtual Memory Manager implements this trap for all machines that it supports, so a debugger can use `_DebugUtil` (and functions defined in terms of `_DebugUtil`) if `Gestalt` reports that virtual memory is present.

When the virtual memory extensions to `_DebugUtil` are not present (that is, when the computer supports virtual memory but is *not* a Macintosh II, Macintosh IIx, Macintosh IIcx, or Macintosh SE/30), `_DebugUtil` provides functions that can determine the highest `_DebugUtil` function supported, enter the debugging state, poll the keyboard for input, and exit the debugging state.

Bus-Error Vectors

The Operating System needs to intercept page faults and do the necessary paging. In addition, various applications and pieces of system software need to handle other kinds of bus errors. Virtual memory takes care of the complications of bus-error handling by providing two bus-error vectors. The vector that applications and other system software see is the one in low memory (at address $8). The vector that virtual memory uses (the one actually used by the processor) is in virtual memory's private storage and is pointed to by the Vector Base Register (VBR). Virtual memory's bus-error handler handles page faults and passes other bus errors to the vector in low memory at address $8.

When a debugger wants the contents of a page to be loaded into memory, it can read a byte from that page. The Operating System detects the page fault and loads the appropriate page (perhaps swapping another page to disk).

Note that a debugger will probably temporarily replace one or both of the bus-error vectors while it is executing. A debugger that wants virtual memory to continue paging while the debugger runs can put a handler only in the low-memory bus-error vector. A debugger that displays memory without allowing virtual memory to continue paging can put a handler in the virtual memory's bus-error vector (at VBR + $8).

Because the current version of virtual memory is not reentrant, there are times when trying to load a page into memory would be fatal. To allow for this, you can use the `PageFaultFatal` function to determine whether a page fault would be fatal at that time. If this function returns `TRUE`, the debugger should not allow the virtual memory's bus-error handler to detect any page faults. Thus, you should always replace the virtual memory's bus-error vector if the `PageFaultFatal` function returns `TRUE`.

Special Nonmaskable Interrupt Needs

Because a debugger can be triggered with a nonmaskable interrupt (level 7, triggered by the interrupt switch), it has special needs that other code in the system does not. For example, because a nonmaskable interrupt might occur while virtual memory is moving pages (to make them contiguous, for example), debugger code must be locked (instead of held, like most other code that must run at a time when page faults would be fatal).

Unfortunately, the `LockMemory` function is intended for use by device drivers and automatically disables data caching for the locked pages. Because this is not desirable for the debugger, the functions `DebuggerLockMemory` and `DebuggerUnlockMemory` lock pages without inhibiting the caching of those pages. Note that both stack, code, and other storage used by the debugger might need to be locked in this way.

Supervisor Mode

Because a debugger is typically activated through one of the processor vectors, it usually executes in supervisor mode, allowing it access to all of memory and all processor registers. When the debugger is entered in another way—for example, through the `_Debugger` or `_DebugStr` trap or when it is first loaded—it is necessary to enter supervisor mode. You can accomplish this with the following assembly-language instructions:

```
MOVEQ #EnterSupervisorMode,D0
_DebugUtil                    ;OS trap to DebugUtils
                              ;on exit, D0 still holds old SR
```

The code switches the caller into supervisor mode, and the previous status register is returned in register D0. Thus, when the debugger returns to the interrupted code, you can restore the previous interrupt level, condition codes, and so forth. When the debugger is ready to return to user mode, it simply loads the status register with the result returned in D0. Entering supervisor mode also switches the stack pointer from the user stack pointer (USP) to the interrupt stack pointer (ISP); reentering user mode changes the stack pointer back to the user stack pointer.

The Debugging State

When activated by an exception, `_Debug` or `_DebugStr` trap, or any other means, the debugger should call the `DebuggerEnter` procedure to notify `_DebugUtil` that the debugger is entering the debugging state. Then `_DebugUtil` can place hardware in a quiescent state and prepare for subsequent `_DebugUtil` calls.

Before returning to the interrupted application code, the debugger must call the `DebuggerExit` procedure to allow `_DebugUtil` to return hardware affected by `DebuggerEnter` to its previous state.

Keyboard Input

A debugger can obtain the user's keyboard input by calling the `DebuggerPoll` procedure. This routine can obtain keyboard input even when interrupts are disabled. After you call this service, you must then obtain keyboard events through the normal event-queue mechanism.

Page States

Debuggers need a way to display the contents of memory without paging or to display the contents of pages currently on disk. The `GetPageState` function returns one of these values to specify the state of a page containing a virtual address:

```
TYPE PageState = Integer;

CONST
   kPageInMemory     = 0;          {page is in RAM}
   kPageOnDisk       = 1;          {page is on disk}
   kNotPaged         = 2;          {address is not paged}
```

A debugger can use this information to determine whether certain memory addresses should be referenced. Note that ROM and I/O space are not pageable and therefore are considered not paged.

Virtual Memory Manager Reference

This section describes the data structures and routines that are provided by the Virtual Memory Manager.

Data Structures

The Virtual Memory Manager defines two data structures for use with the `GetPhysical` function, the memory-block record and the translation table.

Memory-Block Record

The `GetPhysical` function uses a memory-block record to hold information about a block of memory, either logical or physical. The memory-block record is a data structure of type `MemoryBlock`.

```
TYPE MemoryBlock =
RECORD
   address:    Ptr;              {start of block}
   count:      LongInt;          {size of block}
END;
```

Field descriptions

`address` A pointer to the beginning of a block of memory.

`count` The number of bytes in the block of memory.

Translation Table

The `GetPhysical` function uses a translation table to hold information about a logical address range and its corresponding physical addresses. A translation table is defined by the data type `LogicalToPhysicalTable`.

```
TYPE LogicalToPhysicalTable =
RECORD
   logical:    MemoryBlock;         {a logical block}
   physical:   ARRAY[0..defaultPhysicalEntryCount-1] OF
                      MemoryBlock;  {equivalent physical blocks}
END;
```

Field descriptions

`logical` A logical block of memory whose corresponding physical blocks are to be determined.

`physical` A physical translation table that identifies the blocks of physical memory corresponding to the logical block identified in the `logical` field.

Routines

This section describes the routines you can use to control virtual memory. The section "Virtual Memory Management" describes the routines that allow you to control pages in physical memory, and the section "Virtual Memory Debugger Support Routines" describes the routines that only programmers implementing debuggers need to use.

Virtual Memory Management

This section describes the routines you can use to hold logical pages in physical memory and let go of them, lock and unlock pages in physical memory, obtain information about page mapping, and handle interrupts. To hold and release pages, use the `HoldMemory` and `UnholdMemory` functions. To lock and unlock pages, use the `LockMemory`, `LockMemoryContiguous`, and `UnlockMemory` functions. To obtain page-mapping information, use the `GetPhysical` function. To defer user interrupt handling, use the `DeferUserFn` function.

HoldMemory

To make a portion of the address space resident in physical memory and ineligible for paging, use the `HoldMemory` function.

```
FUNCTION HoldMemory (address: UNIV Ptr; count: LongInt): OSErr;
```

`address`	The starting address of the range of memory to be held in RAM.
`count`	The size, in bytes, of the range of memory to be held in RAM.

DESCRIPTION

The `HoldMemory` function makes the portion of the address space beginning at `address` and having a size of `count` bytes resident in physical memory and ineligible for paging.

If the `address` parameter supplied to the `HoldMemory` function is not on a page boundary, then it is rounded down to the nearest page boundary. Similarly, if the specified range does not end on a page boundary, the `count` parameter is rounded up so that the entire range of memory is held.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `HoldMemory` function are

Trap macro	Selector
`_MemoryDispatch`	$0000

The registers on entry and exit for this routine are

Registers on entry

D0	Selector code
A0	Starting address
A1	Number of bytes to hold

Registers on exit

D0	Result code

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Error in parameter list
`notEnoughMemoryErr`	–620	Insufficient physical memory
`interruptsMaskedErr`	–624	Called with interrupts masked

UnholdMemory

To make a currently held range of memory eligible for paging again, use the `UnholdMemory` function.

```
FUNCTION UnholdMemory (address: UNIV Ptr; count: LongInt): OSErr;
```

`address` The starting address of the range of memory to be released.

`count` The size, in bytes, of the range of memory to be released.

DESCRIPTION

The `UnholdMemory` function makes the portion of the address space beginning at `address` and having a size of `count` bytes eligible for paging.

If the `address` parameter supplied to the `UnholdMemory` function is not on a page boundary, then it is rounded down to the nearest page boundary. Similarly, if the specified range does not end on a page boundary, the `count` parameter is rounded up so that the entire range of memory is released.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `UnholdMemory` function are

Trap macro	Selector
`_MemoryDispatch`	$0001

The registers on entry and exit for this routine are

Registers on entry

D0 Selector code

A0 Starting address

A1 Number of bytes to release

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Error in parameter list
`notHeldErr`	–621	Specified range of memory is not held
`interruptsMaskedErr`	–624	Called with interrupts masked

LockMemory

To make a portion of the address space immovable in physical memory and ineligible for paging, use the `LockMemory` function.

```
FUNCTION LockMemory (address: UNIV Ptr; count: LongInt): OSErr;
```

`address` The starting address of the range of memory to be locked in RAM.

`count` The size, in bytes, of the range of memory to be locked in RAM.

DESCRIPTION

The `LockMemory` function makes the portion of the address space beginning at `address` and having a size of `count` bytes immovable in physical memory and ineligible for paging.

If the `address` parameter supplied to the `LockMemory` function is not on a page boundary, it is rounded down to the nearest page boundary. Similarly, if the specified range does not end on a page boundary, the `count` parameter is rounded up so that the entire range of memory is locked.

The CPU marks locked pages as noncacheable. On Macintosh computers containing the Macintosh IIci ROM, all physical RAM is marked noncacheable.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LockMemory` function are

Trap macro	Selector
`_MemoryDispatch`	$0002

The registers on entry and exit for this routine are

Registers on entry

D0 Selector code

A0 Starting address

A1 Number of bytes to lock

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Error in parameter list
`notEnoughMemoryErr`	–620	Insufficient physical memory
`interruptsMaskedErr`	–624	Called with interrupts masked

LockMemoryContiguous

The `LockMemoryContiguous` function is exactly like the `LockMemory` function, except that it attempts to obtain a contiguous block of physical memory associated with the specified logical address range.

```
FUNCTION LockMemoryContiguous (address: UNIV Ptr; count: LongInt):
                               OSErr;
```

`address` The starting address of the range of memory to be locked in RAM.

`count` The size, in bytes, of the range of memory to be locked in RAM.

DESCRIPTION

The `LockMemoryContiguous` function makes the portion of the address space beginning at `address` and having a size of `count` bytes immovable in physical memory and ineligible for paging. The function attempts to obtain a contiguous block of physical memory associated with the specified logical address range. It might not be possible to make a range physically contiguous if any of the pages contained in the range are already locked.

If the `address` parameter supplied to the `LockMemoryContiguous` function is not on a page boundary, it is rounded down to the nearest page boundary. Similarly, if the specified range does not end on a page boundary, the `count` parameter is rounded up so that the entire range of memory is locked.

The CPU marks locked pages as noncacheable. On Macintosh computers containing the Macintosh IIci ROM, all physical RAM is marked noncacheable.

SPECIAL CONSIDERATIONS

Because a call to `LockMemoryContiguous` is not guaranteed to succeed, all code that uses `LockMemoryContiguous` must have an alternate method for locking the necessary ranges of memory. In general, you should avoid using `LockMemoryContiguous` if at all possible. If you must call it, do so as early as possible—preferably at system startup time—to increase the likelihood that enough contiguous memory can be found.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LockMemoryContiguous` function are

Trap macro	Selector
`_MemoryDispatch`	$0004

The registers on entry and exit for this routine are

Registers on entry

D0 Selector code

A0 Starting address

A1 Number of bytes to unlock

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Error in parameter list
`notEnoughMemoryErr`	–620	Insufficient physical memory
`cannotMakeContiguousErr`	–622	Cannot make specified range contiguous
`interruptsMaskedErr`	–624	Called with interrupts masked

UnlockMemory

To undo the effects of either `LockMemory` or `LockMemoryContiguous`, use the `UnlockMemory` function.

```
FUNCTION UnlockMemory (address: UNIV Ptr; count: LongInt): OSErr;
```

`address` The starting address of the range of memory to be unlocked.

`count` The size, in bytes, of the range of memory to be unlocked.

DESCRIPTION

The `UnlockMemory` function makes the portion of the address space beginning at `address` and having a size of `count` bytes movable in real memory and eligible for paging again.

If the `address` parameter supplied to the `UnlockMemory` function is not on a page boundary, then it is rounded down to the nearest page boundary. Similarly, if the specified range does not end on a page boundary, the `count` parameter is rounded up so that the entire range of memory is unlocked.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `UnlockMemory` function are

Trap macro	Selector
`_MemoryDispatch`	$0003

The registers on entry and exit for this routine are

Registers on entry

D0 Selector code

A0 Starting address

A1 Number of bytes to unlock

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Error in parameter list
`notLockedErr`	–623	Specified range of memory is not locked
`interruptsMaskedErr`	–624	Called with interrupts masked

GetPhysical

To translate logical addresses into their corresponding physical addresses, use the `GetPhysical` function.

```
FUNCTION GetPhysical (VAR addresses: LogicalToPhysicalTable;
                      VAR physicalEntryCount: LongInt): OSErr;
```

`addresses` A translation table. On entry, set the `logical` field of this record to the block of memory to translate. On exit, the `physical` field of this record holds the corresponding physical address blocks.

`physicalEntryCount`
The number of physical entries to translate. On entry, set this field to 0 if you want `GetPhysical` to return the number of table entries needed to translate the entire logical address range.

DESCRIPTION

The `GetPhysical` function translates a logical address range into its corresponding physical address ranges. The `logical` field of the `addresses` translation table specifies the logical address range to be translated. `GetPhysical` translates up to the size of the physical table or until it completes the translation, whichever occurs first.

If you call `GetPhysical` with the `physicalEntryCount` parameter set to 0, it returns in `physicalEntryCount` the number of table entries needed to translate the entire address range. In this case, the translation table specified by the `addresses` parameter is unchanged.

If you call `GetPhysical` with the `physicalEntryCount` parameter set to a number greater than 0, it returns in the `physical` field of the `addresses` translation table an array specifying the physical blocks that correspond to the logical address specified in the `logical` field. In the `physicalEntryCount` parameter, `GetPhysical` returns the number of entries in that array (which may be fewer than were asked for). If the `physical` field of the translation table was not large enough to contain all the physical blocks corresponding to the logical block, `GetPhysical` updates the fields of the `logical` memory block to reflect the remaining number of bytes in the logical range left to translate (`count` field) and the next address in the logical address range to translate (`start` field).

Note
The logical address range must be locked to ensure validity of the translation data. ◆

SPECIAL CONSIDERATIONS

The `GetPhysical` function as currently implemented under virtual memory supports only logical RAM. You cannot use `GetPhysical` to translate addresses in the address spaces of the ROM, I/O devices, or NuBus slots. Some Macintosh computers map a portion of the physical RAM into NuBus space, to simulate the presence of a video expansion card. `GetPhysical` returns the result code `paramErr` if you attempt to read that memory.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetPhysical` function are

Trap macro	Selector
`_MemoryDispatchA0Result`	$0005

The registers on entry and exit for this routine are

Registers on entry

D0	Selector code
A0	Pointer to a translation table
A1	`physicalEntryCount` in table

Registers on exit

A0	`physicalEntryCount` translated
D0	Result code

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Error in parameter list
`notLockedErr`	–623	Specified range of memory is not locked
`interruptsMaskedErr`	–624	Called with interrupts masked

SEE ALSO

See "Mapping Logical to Physical Addresses," beginning on page 3-16, for a method of calling `GetPhysical` to translate addresses to be sent to a NuBus master card.

DeferUserFn

To determine whether code that might cause page faults can safely be called immediately, use the `DeferUserFn` function.

```
FUNCTION DeferUserFn (userFunction: ProcPtr;
                      argument: UNIV Ptr): OSErr;
```

`userFunction`
The address of the routine to run.

`argument` A pointer to the argument to pass to the specified routine.

DESCRIPTION

The `DeferUserFn` function determines whether or not code that might call page faults can safely be called immediately. If the code can be called safely, `DeferUserFn` calls the routine designated by `userFunction` with register A0 containing the value designated by `argument`. If a page fault is in progress, however, the routine address and its parameter are saved, and the routine is deferred until page faults are again permitted.

Note that the routine might be called immediately (before returning to the caller of `DeferUserFn`). Deferred functions must follow the register conventions used by interrupt handlers: they can use registers A0–A3 and D0–D3, and they must restore all other registers used.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for the `DeferUserFn` function are

Registers on entry

A0 Address of function

D0 Argument for function

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`cannotDeferErr`	–625	Unable to defer additional user functions

Virtual Memory Debugger Support Routines

This section describes the virtual-memory routines that pertain primarily to debuggers. You need to read this section only if you are implementing a debugger. To determine which debugger functions are present, use the `DebuggerGetMax` function. When entering and exiting the debugging state, use the `DebuggerEnter` and the `DebuggerExit` procedures. To determine whether paging is safe, use the `PageFaultFatal` function. To lock and unlock memory with caching enabled, use the `DebuggerLockMemory` and the `DebuggerUnlockMemory` functions. To poll for keyboard input, use the `DebuggerPoll` procedure. To determine the state of a page of logical memory, use the `GetPageState` function.

DebuggerGetMax

The Memory Manager includes a special routine that debuggers use, instead of the `Gestalt` function, to determine which debugger functions are present.

```
FUNCTION DebuggerGetMax: LongInt;
```

DESCRIPTION

The `DebuggerGetMax` function returns the highest selector number of the debugger routines that are defined in terms of the `_DebugUtil` trap. The numbers correspond to the following routines:

Selector	Routine
$0000	`DebuggerGetMax`
$0001	`DebuggerEnter`
$0002	`DebuggerExit`
$0003	`DebuggerPoll`
$0004	`GetPageState`
$0005	`PageFaultFatal`
$0006	`DebuggerLockMemory`
$0007	`DebuggerUnlockMemory`
$0008	`EnterSupervisorMode`

Of course, you should use the `Gestalt` function to check whether virtual memory is available at all before you call the `DebuggerGetMax` function.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DebuggerGetMax` function are

Trap macro	Selector
`_DebugUtil`	$0000

The registers on entry and exit for this routine are

Registers on entry

D0 Selector code

Registers on exit

D0 Highest available selector

DebuggerEnter

Before entering the debugging state, call the `DebuggerEnter` procedure.

```
PROCEDURE DebuggerEnter;
```

DESCRIPTION

Call the `DebuggerEnter` procedure to enter the debugging state. This allows the `_DebugUtil` trap to make preparations for subsequent debugging calls.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DebuggerEnter` procedure are

Trap macro	Selector
`_DebugUtil`	$0001

The registers on entry for this routine are

Registers on entry

D0 Selector code

DebuggerExit

Before exiting the debugging state, call the `DebuggerExit` procedure.

```
PROCEDURE DebuggerExit;
```

DESCRIPTION

The `DebuggerExit` procedure allows the `_DebugUtil` trap to clean up after all debugging calls are completed.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DebuggerExit` procedure are

Trap macro	Selector
`_DebugUtil`	$0002

The registers on entry for this routine are

Registers on entry

D0 Selector code

PageFaultFatal

A debugger can use the `PageFaultFatal` function to determine whether it should capture all bus errors or whether it is safe to allow them to flow through to virtual memory. When paging is safe, the debugger can allow virtual memory to continue servicing page faults, and the user can view all of memory.

```
FUNCTION PageFaultFatal: Boolean;
```

DESCRIPTION

The `PageFaultFatal` function returns `TRUE` if the debugger should not allow the virtual memory's bus-error handler to detect any page faults.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PageFaultFatal` function are

Trap macro	Selector
`_DebugUtil`	$0005

The registers on entry and exit for this routine are

Registers on entry

D0 Selector code

Registers on exit

D0 Returned value

DebuggerLockMemory

To lock a portion of the address space (as the `LockMemory` function does) while leaving data caching enabled on the affected pages, use the `DebuggerLockMemory` function.

```
FUNCTION DebuggerLockMemory (address: UNIV Ptr; count: LongInt):
                                OSErr;
```

`address` The start address of the range of memory that is to be locked in RAM.

`count` The size in bytes of the range of memory that is to be locked in RAM.

DESCRIPTION

The `DebuggerLockMemory` function makes the portion of the address space beginning at `address` and having a size of `count` bytes immovable in physical memory and ineligible for paging. The function leaves data caching enabled on the affected pages.

If the `address` parameter supplied to the `DebuggerLockMemory` function is not on a page boundary, then it is rounded down to the nearest page boundary. Similarly, if the specified range does not end on a page boundary, the `count` parameter is rounded up so that the entire range of memory is locked.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DebuggerLockMemory` function are

Trap macro	Selector
`_DebuggerLockMemory`	$0006

The registers on entry and exit for this routine are

Registers on entry

D0 Selector code

A0 Starting address

A1 Number of bytes to hold

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Error in parameter list
`notEnoughMemoryErr`	–620	Insufficient physical memory

DebuggerUnlockMemory

To reverse the effects of `DebuggerLockMemory`, use the `DebuggerUnlockMemory` function.

```
FUNCTION DebuggerUnlockMemory (address: UNIV Ptr; count: LongInt):
                                    OSErr;
```

`address` The starting address of the range of memory that is to be unlocked.

`count` The size, in bytes, of the range of memory that is to be unlocked.

DESCRIPTION

The `DebuggerUnlockMemory` function makes the portion of the address space beginning at `address` and having a size of `count` bytes movable in real memory and eligible for paging again.

If the `address` parameter supplied to the `DebuggerUnlockMemory` function is not on a page boundary, then it is rounded down to the nearest page boundary. Similarly, if the specified range does not end on a page boundary, the `count` parameter is rounded up so that the entire range of memory is unlocked.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DebuggerUnlockMemory` function are

Trap macro	Selector
`_DebugUtil`	$0007

The registers on entry and exit for this routine are

Registers on entry

D0 Selector code

A0 Starting address

A1 Number of bytes to hold

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Error in parameter list
`notLockedErr`	–623	Specified range of memory is not locked

DebuggerPoll

To poll for keyboard input, use the `DebuggerPoll` procedure.

```
PROCEDURE DebuggerPoll;
```

DESCRIPTION

Call the `DebuggerPoll` procedure, which you can use even if interrupts are disabled, to poll for keyboard input.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DebuggerPoll` procedure are

Trap macro	Selector
`_DebugUtil`	$0003

The registers on entry and exit for this routine are

Registers on entry

D0 Selector code

Registers on exit

D0 Result code

GetPageState

To obtain the state of a page of logical memory, use the `GetPageState` function.

```
FUNCTION GetPageState (address: UNIV Ptr): PageState;
```

`address` An address in the page whose state you want to determine.

DESCRIPTION

The `GetPageState` function returns the page state of the page containing the address passed in the `address` parameter. The returned value is one of these constants:

```
TYPE PageState = Integer;

CONST
   kPageInMemory  = 0;          {page is in RAM}
   kPageOnDisk    = 1;          {page is on disk}
   kNotPaged      = 2;          {address is not paged}
```

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetPageState` function are

Trap macro	Selector
`_DebugUtil`	$0004

The registers on entry and exit for this routine are

Registers on entry

A0 Address in the page whose state is to be determined

D0 Selector code

Registers on exit

D0 Page state

Summary of the Virtual Memory Manager

Pascal Summary

Constants

```
CONST
   {Gestalt constants}
   gestaltVMAttr              = 'vm  ';   {virtual memory attributes}
   gestaltVMPresent           = 0;        {bit set if virtual memory present}

   {default number of physical blocks in a translation table}
   defaultPhysicalEntryCount  = 8;

   {page states}
   kPageInMemory              = 0;        {page is in RAM}
   kPageOnDisk                = 1;        {page is on disk}
   kNotPaged                  = 2;        {address is not paged}
```

Data Types

```
TYPE
   PageState            = Integer;


   LogicalToPhysicalTable =            {translation table}
   RECORD
      logical:       MemoryBlock;      {logical block}
      physical:      ARRAY[0..defaultPhysicalEntryCount-1] OF MemoryBlock;
                                       {equivalent physical blocks}
   END;

   MemoryBlock =                       {memory-block record}
   RECORD
      address:       Ptr;              {start of block}
      count:         LongInt;          {size of block}
   END;
```

Routines

Virtual Memory Management

```
FUNCTION HoldMemory         (address: UNIV Ptr; count: LongInt): OSErr;
FUNCTION UnholdMemory       (address: UNIV Ptr; count: LongInt): OSErr;
FUNCTION LockMemory         (address: UNIV Ptr; count: LongInt): OSErr;
FUNCTION LockMemoryContiguous
                            (address: UNIV Ptr; count: LongInt): OSErr;
FUNCTION UnlockMemory       (address: UNIV Ptr; count: LongInt): OSErr;
FUNCTION GetPhysical        (VAR addresses: LogicalToPhysicalTable;
                             VAR physicalEntryCount: LongInt): OSErr;
FUNCTION DeferUserFn        (userFunction: ProcPtr; argument: UNIV Ptr):
                             OSErr;
```

Virtual Memory Debugger Support Routines

```
FUNCTION DebuggerGetMax     : LongInt;
PROCEDURE DebuggerEnter;
PROCEDURE DebuggerExit;
FUNCTION PageFaultFatal     : Boolean;
FUNCTION DebuggerLockMemory (address: UNIV Ptr; count: LongInt): OSErr;
FUNCTION DebuggerUnlockMemory
                            (address: UNIV Ptr; count: LongInt): OSErr;
PROCEDURE DebuggerPoll;
FUNCTION GetPageState       (address: UNIV Ptr): PageState;
```

C Summary

Constants

```
/*Gestalt constants*/
#define gestaltVMAttr         'vm  ';  /*virtual memory attributes*/
#define gestaltVMPresent       0;      /*bit set if virtual memory present*/

/*default number of physical blocks in table*/
enum {
   defaultPhysicalEntryCount     = 8
};
```

```
/*page states*/
enum {
   kPageInMemory                = 0,     /*page is in RAM*/
   kPageOnDisk                  = 1,     /*page is on disk*/
   kNotPaged                    = 2      /*address is not paged*/
};
```

Data Types

```
typedef short PageState;

struct LogicalToPhysicalTable {           /*translation table*/
      MemoryBlock       logical;          /*logical block*/
      MemoryBlock       physical[defaultPhysicalEntryCount];
                                          /*equivalent physical blocks*/
   };
typedef struct LogicalToPhysicalTable LogicalToPhysicalTable;

struct MemoryBlock {                      /*memory-block record*/
      void              *address;         /*start of block*/
      unsigned long     count;            /*size of block*/
};
typedef struct MemoryBlock MemoryBlock;
```

Routines

Virtual Memory Management

```
pascal OSErr HoldMemory     (void *address, unsigned long count);
pascal OSErr UnholdMemory   (void *address, unsigned long count);
pascal OSErr LockMemory     (void *address, unsigned long count);
pascal OSErr LockMemoryContiguous
                            (void *address, unsigned long count);
pascal OSErr UnlockMemory   (void *address, unsigned long count);
pascal OSErr GetPhysical    (LogicalToPhysicalTable *addresses,
                             unsigned long *physicalEntryCount);
pascal OSErr DeferUserFn    (ProcPtr userFunction, void *argument);
```

Virtual Memory Debugger Support Routines

```
pascal long DebuggerGetMax  (void);
pascal void DebuggerEnter   (void);
pascal void DebuggerExit    (void);
pascal Boolean PageFaultFatal
                            (void);
pascal OSErr DebuggerLockMemory
                            (void *address, unsigned long count);
pascal OSErr DebuggerUnlockMemory
                            (void *address, unsigned long count);
pascal void DebuggerPoll    (void);
pascal PageState GetPageState
                            (const void *address);
```

Assembly-Language Summary

Data Types

Memory-Block Data Structure

0	`address`	long	start of block
4	`count`	4 bytes	size of block

Translation Table Data Structure

0	`logical`	8 bytes	logical block
8	`physical`	64 bytes	equivalent physical blocks

Trap Macros

Trap Macros Requiring Routine Selectors

`_MemoryDispatch`

Selector	Routine
$0000	`HoldMemory`
$0001	`UnholdMemory`
$0002	`LockMemory`
$0003	`UnlockMemory`
$0004	`LockMemoryContiguous`

_MemoryDispatchA0Result

Selector	Routine
$0005	GetPhysical

_DebugUtil

Selector	Routine
$0000	DebuggerGetMax
$0001	DebuggerEnter
$0002	DebuggerExit
$0003	DebuggerPoll
$0004	GetPageState
$0005	PageFaultFatal
$0006	DebuggerLockMemory
$0007	DebuggerUnlockMemory
$0008	EnterSupervisorMode

Result Codes

noErr	0	No error
paramErr	–50	Error in parameter list
notEnoughMemoryErr	–620	Insufficient physical memory
notHeldErr	–621	Specified range of memory is not held
cannotMakeContiguousErr	–622	Cannot make specified range contiguous
notLockedErr	–623	Specified range of memory is not locked
interruptsMaskedErr	–624	Called with interrupts masked
cannotDeferErr	–625	Unable to defer additional user functions