

# New Technical Notes

Macintosh



®

---

Developer Support

## Coping With VM and Memory Mappings

### Memory M.ME.VM Memory Mapping

Revised by: Craig Prouse

April 1991

Written by: Craig Prouse

February 1991

The purpose of this Note is twofold. First, it describes in detail how to use the `GetPhysical` routine. This routine is critical to the support of alternate bus masters on certain machines without Virtual Memory (VM) and all machines with VM. Included is an ancillary discussion of several closely-related VM routines. Second, it reiterates a number of issues important to VM compatibility and elucidates some of the deeper VM issues of which specialized developers should be aware. Compatibility issues are especially important for developers of SCSI drivers, NuBus™ master hardware, and code which runs at interrupt time.

**Changes since February 1991:** This update incorporates new issues which have come up during System 7.0 beta testing, and it also attempts to clarify some issues which have proven to be particularly troublesome or widely misunderstood.

---

### Everybody Must Get Physical

If you are developing NuBus expansion cards with bus mastership or direct memory access (DMA) capabilities, and if you have ever done development or compatibility testing with Apple's recent machines, like the Macintosh IIci and Macintosh IIsx, you have undoubtedly noticed some strange behavior. You might tell the card to dump data into a buffer at \$00300000 and the data instead appears at \$006B0000. "What's happening here?" you must ask yourself.

Well, there's a new game in town—it's called a **discontiguous physical address space**. What that means in simple terms is that there is potentially a big hole in memory. If you have eight megabytes installed in a Macintosh IIci, for instance, that memory appears to the CPU and to NuBus in two separate 4 MB ranges: [\$00000000 – \$003FFFFFFF] and [\$04000000 – \$043FFFFFFF]. Everything from the end of Bank A to the beginning of Bank B is essentially empty. Bank B memory does not start until at least \$04000000.

To compensate for this, the operating system uses the memory management unit (MMU) to map all the **physical** memory (what the hardware sees) into a single contiguous **logical** address space (what all Macintosh code sees). The logical address space looks exactly like the memory map you've known for years. The translation is completely transparent to software. If you're an applications developer and you read the low-memory global at \$10C, you don't care that the address that the processor actually looks at is \$0400010C. When the processor originally put a value in that spot, it went through the same translation. Everything is relative and you always get just what you'd expect.



The sole exception is for software which runs on the Macintosh but communicates addresses to NuBus master hardware. Say, for instance, that you have developed a video frame grabber which dumps an image into a handle you've allocated for that purpose. When you call `_NewHandle` with an argument of `frameSize`, you get back a logical address. If you use a 68030, or a 68020 with a 68851 PMMU, to store data into that handle, the MMU performs an address translation and places data into a corresponding physical address. NuBus hardware, however, does not use the MMU's address mapping tables. If your driver passes along a logical address from the Memory Manager, the frame grabber does not know to translate it (indeed it cannot), and the logical address is interpreted as a physical address. External hardware may dump a beautiful captured image well outside your carefully allocated handle and perhaps right across the top of MacsBug and other similarly important things. Bugs like this are extremely difficult to isolate unless you understand their behavior and anticipate them.

The point is, now you must be sure to always convert logical addresses to corresponding physical addresses before passing them to any alternate bus master. A new function to support this is `GetPhysical`, which is documented in *Inside Macintosh*, Volume VI. "Great," you say, once you've read the documentation. "But `GetPhysical` is a System Software 7.0 feature and a Virtual Memory feature to boot. What do I do for System 6.0.x or if I'm not running VM?"

I'm glad you asked, because VM and the memory architecture of the Macintosh IIci are related topics. `GetPhysical`, a routine required by the IIci, is one of a suite of functions dispatched by a trap called `_MemoryDispatch` (\$A05C), which is the same trap used by the major VM calls. Because some machines require `GetPhysical` even without VM, those machines have a limited form of `_MemoryDispatch` implemented in ROM.

You **can** call `GetPhysical` under System 6.0.x or under System 7.0 even when VM is not running—all you must do first is check to see that the `_MemoryDispatch` trap is implemented. If this trap is implemented, it is there for a reason, and you should use it. Although `GetPhysical` is present only for certain machines without VM, it is present and required for all machines running VM. If you update your code to be compatible with the IIci and IIsi in the 6.0.x world, you are already doing part of what is required to be compatible with Virtual Memory and System Software 7.0.

## **Holding and Locking Memory Versus Locking Handles**

Virtual Memory introduces two new concepts—holding and locking a range of virtual memory. These are not to be confused with locking a handle. Locking a handle prevents the handle from changing its logical address during Memory Manager operations. Holding and locking virtual memory affects how VM deals with arbitrary ranges of memory during paging operations.

Holding and locking memory (as opposed to a handle) are VM functions exclusively and are

accomplished with four new `_MemoryDispatch` routines: `HoldMemory` and `LockMemory`, and the corresponding routines to undo these operations, `UnholdMemory` and `UnlockMemory`. Pay special attention any time you hold or lock a range of memory that you subsequently unhold or unlock the same range. Every single call to `HoldMemory` or `LockMemory` must be balanced by a corresponding `UnholdMemory` or `UnlockMemory` because the operating system supports multiple levels of locking and holding, much like it supports multiple levels of cursor obscuration with `_ShowCursor` and `_HideCursor`.

Holding a range of memory guarantees that the data in that range is actually somewhere in physical Macintosh RAM and that no paging activity is necessary to load it. This is critical for tasks which run at interrupt time, since paging activity should not be initiated at interrupt time. VM is not guaranteed to be reentrant, and because interrupts may occur in the middle of paging, any data accessed by an interrupt handler should reside in a held block of memory. Only hold memory which legitimately needs to be held though, because any memory which is held becomes ineligible for paging. This reduces the space VM has to work with and may significantly impact system performance. Some interrupt-time tasks are deferred by VM until paging is safe, so memory they touch does not always have to be held. These tasks are called out below, in the section “Compatibility With Other Device Drivers and Interrupt-Level Code.”

Locking a range of memory is more severe than holding it. This not only forces the range to be held resident in physical RAM, but also prevents its logical address from moving with respect to its physical address. This is important for drivers which initiate DMA transactions, because there must be a known, static relationship between logical and physical addresses for the duration of such an operation. Part of the behavior of `LockMemory` is to make the associated memory non-cachable which is important for DMA transfers.

**Warning:** Apple cannot make the point too strongly that memory should only be held or locked when absolutely necessary, and only as long as necessary. It is worth restating that the impact on performance can be significant or even fatal in severe cases. It is a crime against the machine to hold or lock memory unnecessarily. Failure to unhold or unlock memory previously held or locked is most heinous.

In non-VM environments, there is no page swapping activity. This is similar to all of memory being locked, except that caching is still enabled. Truly locked memory is neither cached nor paged. If you are running System Software 7.0 with VM, you **must** explicitly lock a range of memory with `LockMemory` before calling `GetPhysical`. You may only call `GetPhysical` on a locked block of virtual memory, or you get an error, since, among other reasons, any paging activity could invalidate the results of a `GetPhysical` call. Although it is not necessary to call `LockMemory` before `GetPhysical` if VM is not running, `LockMemory` may still be used for its favorable effect of disabling caching. This Note includes a code template (located at the end) which illustrates a “way rad” method to implement driver calls to a generic NuBus master card. It doesn’t even have to know if VM is running. Hardware and drivers should be designed to support this method for maximum VM friendliness.

There is one more VM routine of interest, `LockMemoryContiguous`, which is provided to assist developers whose DMA hardware is not capable of transferring blocks of arbitrary size or for some other reason cannot use a generalized algorithm such as the one provided. Apple can only warn developers that `LockMemoryContiguous` is potentially an expensive operation in terms of performance and is one very likely to fail since contiguous physical memory may be difficult, if not impossible, to find. `LockMemoryContiguous`

is not particularly useful, unless VM is running, should a range of memory happen to cross a physical discontinuity like that found on a Macintosh IIci. No hardware or software product should require VM in order to run. `LockMemoryContiguous` might be useful for determining whether a range of logical memory is actually physically contiguous, although `GetPhysical` can do the same thing without actually locking the memory.

Apple's primary recommendation regarding `LockMemoryContiguous` is to avoid its use if at all possible. If you must use `LockMemoryContiguous`, Apple recommends that you allocate your buffer as early as possible (preferably at startup) and lock it down contiguously at that time. VM is an entropic system, meaning its pages tend to become shuffled over time, so it's easiest to find contiguous memory early in a session.

## When to Call?

### `HoldMemory`

- Before taking control of the SCSI bus.
- Before accessing memory at interrupt time.
- To keep **critical** ranges of memory resident for performance reasons.

### `LockMemory`

- Rarely. (Always `UnlockMemory` as soon as possible.)
- Before calling `GetPhysical`.
- Before initiating a DMA transfer.

### `LockMemoryContiguous`

- Never, if you can help it. (If necessary, do so as early as possible—see text above).

## When Not to Call?

### `HoldMemory`

- To keep **large** ranges of memory resident for performance reasons.

### `LockMemory`

- Before dereferencing a handle. (`LockMemory` should not be confused with `_HLock`.)
- When you really mean `HoldMemory`.

## What Form Of Address To Pass?

All `_MemoryDispatch` routines described above work as expected in either 24-bit mode or 32-bit mode. In 24-bit mode, for instance, master pointer flags or other garbage bits in the high-order eight bits are ignored and taken to be zero. When switching between 24-bit and 32-bit modes, remember to use `_StripAddress` as outlined in Technical Note #213, `_StripAddress: The Untold Story`.

## **Special Considerations**

The `GetPhysical` call in ROM and system software currently supports only logical RAM. This excludes the ROM, I/O, and NuBus spaces from the set of addresses `GetPhysical` knows how to translate. Unfortunately, machines like the Macintosh IIci and Macintosh IIsi use the MMU to map a small amount of physical memory into NuBus space so that it looks like



a regular video card. Ideally one might like to use `GetPhysical` to get the actual RAM address of the video buffer (to provide DMA support for certain multimedia products and graphics accelerators), but the current ROM implementation of `GetPhysical` returns a `paramErr` (-50) in response to logical NuBus addresses.

Because its ROM is derived from that of the Macintosh IIci, the Macintosh LC may appear to have `_MemoryDispatch` implemented. This doesn't make sense, however, because the LC has no MMU. Although System Software 7.0 patches `_MemoryDispatch` in this case to make it unimplemented, `PrimaryInit` code and SCSI drivers which run before system patches are installed could be affected. Code running at this time should qualify the existence of `_MemoryDispatch` with the existence of an MMU, using `_Gestalt`.

In order to solve both of these problems as cleanly as possible, the MPW libraries contain an enhanced version of `GetPhysical` with greater flexibility than the ROM version.\* Although the enhanced version is the same as the ROM version in most cases, it provides extra validation checks to guarantee stability before system patches are installed, and it applies alternate mechanisms to determine the physical address of a RAM-based video buffer. You should therefore call `GetPhysical` where it is indicated, even for address spaces where the ROM version is known to return an error. The glue code may pick up the slack or a future ROM might not return an error. In any case, your code should always be prepared to cope with any of the `GetPhysical` error results documented in *Inside Macintosh*. Remember always to call `LockMemory` before calling `GetPhysical`, and `UnlockMemory` as soon as possible afterwards.

(\*At this writing, the enhanced `GetPhysical` code has not yet been incorporated into beta versions of the System 7.0 interface libraries. This code will be made available at the earliest opportunity and this Note will be revised to indicate its availability. If you need `GetPhysical` to operate on RAM-based video buffers or you need to call `GetPhysical` as part of a `PrimaryInit` or SCSI driver initialization, you should be certain to take defensive measures against the special cases described above.)

## **VM Compatibility**

### **Compatibility With Accelerator Upgrades**

The burden of compatibility has long been on the shoulders of accelerator manufacturers. VM may present some additional compatibility challenges for these manufacturers.

Virtual Memory requires services which are not present in the ROMs of 68000-based machines, so VM is not supported by the Macintosh SE, even one with a 68030 accelerator. The same is true of the Macintosh Plus, the Macintosh Classic, and the Macintosh Portable. There is no guarantee that these older machines will ever be able to support VM. For practical reasons, Apple has chosen not to implement VM in a wholly ROM-independent

manner. In the foreseeable future, only machines in which Apple intended to include memory management units can support Virtual Memory. Machines never intended to include an MMU do not have all the ROM code required by VM.

Virtual Memory depends on low-memory globals to indicate the presence of a memory management unit at a very early stage of the boot process. In some cases, the low-memory globals are not properly set by the boot code in ROM if the hardware features of an accelerator

are significantly different from those of the stock Macintosh. The most likely problems are exhibited by 68000 Macintoshes, 68020 Macintoshes with 68030 accelerators, and Macintoshes with 68040 accelerators. There is third-party virtual memory software which provides much of the VM functionality of System Software 7.0, and which is also compatible with accelerator products. In some cases this software may be bundled with the accelerator.

Apple is not saying that VM does not work with any accelerator, but rather that the System 7.0 implementation of Virtual Memory in general does not support accelerators. Some accelerator products may work or may be modified to work. Apple simply does not guarantee that any particular accelerator product works with VM.

### **Compatibility With Removable Media**

Obviously it would be a disaster if a user ejected the cartridge containing his backing store (paged out memory) and handed it to a coworker to take home. This would be much worse than giving away a floppy, to be faced with the “Please insert the disk...” alert. Someone would actually have part of the computer’s memory in his briefcase—try to type Command-period and get out of that one. To guard against this possibility, ejectable media are not permitted to host the VM backing store. Users of removable cartridge drives are not wholly excluded, however. The driver software for such a drive may impose software interlocks to prevent ejection and indicate in the drive queue that the cartridge is nonejectable. VM accepts any sufficiently large, block oriented device as long as it is not ejectable.

### **Compatibility With SCSI Code**

Virtual Memory introduces new requirements for some SCSI hard disk drivers. Users of Apple hard disks may need to update their drivers with a System Software 7.0-compatible Apple HD SC Setup application. Third-party hard disk drivers may also need to be updated. It is up to these third parties to determine what enhancements, if any, are required for their drivers and to provide updates to their customers if necessary.

For SCSI disk driver developers, one requirement for VM compatibility may be summarized as follows (special thanks to Andy Gong for the detailed analysis):

On System 6.0.x and earlier, all calls to the SCSI disk driver came from the file system. This being true, and the file system being single-threaded, only one SCSI disk driver would be called at any one time. Virtual Memory changes this scenario because it makes calls to the driver directly, avoiding the file system. This implies the possibility of SCSI drivers being reentered.

For a SCSI driver to function correctly in the VM environment, the driver must have complete driver data separation at least on a drive-by-drive basis. Such separation makes the driver reentrant on a drive-by-drive basis. If the driver supports multiple HFS partitions on the same physical drive, the driver must be

completely reentrant if any of the HFS partitions are to be used for the VM backing file.

All this means is that a driver which controls multiple drives or partitions must maintain separate driver variables to reference each drive or partition. Otherwise, the state of a transaction to one drive may be lost when the driver is reentered to service another drive. There is no problem with reentrancy for drivers which control only a single drive or partition.

In many cases of SCSI code incompatibility, reentrancy is not the problem. This affects only the small number of SCSI disk drivers which are designed to control multiple drives or partitions from a single driver. A more common problem is caused by a page fault while the SCSI bus is busy. Since VM depends on the SCSI bus to handle a page fault, a page fault is forbidden to happen while the SCSI bus is busy. Code which uses the SCSI Manager needs in general to ensure that all its code, buffers, and data structures (including TIBs) are held in real memory before taking control of the bus.

In the normal course of events, the system heap is held in real memory. Other critical structures are held for you automatically, like any range of memory passed to a Device Manager `_Read` or `_Write` call in `ioBuffer` and `ioReqCount`. So if your SCSI code is written as a device driver, and the buffer's address and length are passed in the normal driver fashion, and if your driver code and data structures are located in the system heap, you should be fully VM-compatible already (as long as you only operate on one drive per driver).

If your SCSI code is not a standard Device Manager driver or if you reference buffers as `csParams` to `_Control` or `_Status` calls, you'll need to do some extra work. Also, Apple does not guarantee that the system heap will always be held for ever and ever, so if you come to revise your driver you should seriously consider holding explicitly everything you touch while you own the SCSI bus and everything you might knowingly touch at interrupt time; and of course you should correspondingly unhold all these structures upon releasing the bus. Be a good citizen.

In addition to the requirement for reentrancy across drives served by a single driver, the driver for a disk used as a backing store must load at the earliest possible opportunity. Drivers which defer installation until INIT time are too late to be used by VM.

## **Compatibility With Other Device Drivers and Interrupt-Level Code**

The primary concern for device drivers is that they commonly run at interrupt time and it is absolutely essential that interrupt-level code does not cause a page fault. To avoid this, drivers should make certain that any data structures they keep or reference at interrupt time are held in physical memory as described earlier. Locking the structures is typically not necessary except in cases where alternate bus master hardware accesses those structures as well.

To improve performance and compatibility with existing software and drivers, the first release of System Software 7.0 always holds the entire system heap in physical memory. No special measures need be taken if your driver and its associated data structures are all installed in the system heap. If your driver uses memory statically allocated above `BufPtr`, it may need to explicitly hold the appropriate ranges of memory to avoid paging at interrupt time. Please be aware that future versions of the Macintosh System Software may **not** hold all of the system heap automatically and it is a good habit to hold explicitly memory you know you access at interrupt time.

The Device Manager deals with `_Read` and `_Write` calls for you, and ensures that the buffers specified for such calls are safe. However, if a buffer is passed as a `csParam` to `_Status` or `_Control` calls, the Device Manager cannot do anything about it. Buffers referenced this way must be held explicitly if they are to be accessed by interrupt-level code.

Certain code types are always deferred until times when paging is safe, and as such don't have to be concerned about whether memory they touch is guaranteed to be held. Those code types include Device Manager I/O completion routines, Time Manager tasks, VBL tasks, and slot

VBL tasks. The trade-off is in real-time performance. Clearly, since these tasks may be deferred, there is an increased possibility of latency which may be unacceptable for some pseudo-real-time applications. (The Macintosh has never supported true real-time processing.) An arbitrary function which might cause a page fault at interrupt time can be deferred explicitly by calling it via the trap `_DeferUserFn`.

The `_DeferUserFn` trap is asynchronous in nature, so subsequent code may be executed before the deferred function completes. If the results of a deferred function are vital to the code which follows, the deferred function needs to signal the calling code when it completes.

Apple Desktop Bus I/O requests are deferred until a time when paging is safe unless VM is certain that all code and associated data structures are located in the system heap. This is required because the ADB Manager normally processes incoming data at interrupt time and there is a potential for page faults if the service routine code or other data structures are not held in real memory. The only problem with this strategy is reduced performance for specialized ADB drivers which require most of the ADB bandwidth and don't live in the system heap. Nonetheless, it's worth mentioning.

One final note of interest pertains to a longstanding anomaly in the Device Manager. As it turns out, when you make an asynchronous `_Open` or `_Close` call to a device driver, any completion routine you supply is never called. Since Virtual Memory patches `_Open` and `_Close`, and generates an entry for the completion routine in the user function queue, the implication is that the user functions are never executed and the queue may simply fill up. There is little reason to call `_Open` or `_Close` asynchronously with a completion routine (it never would have amounted to anything anyway), so the workaround is simple: don't do it.

## **Compatibility With the BufPtr Method of Static Allocation**

*Inside Macintosh*, Volume IV describes, on page 257, a method of static allocation for resident drivers or other data structures. This method has been very popular with a number of developers. The main thing for developers to remember about this method in conjunction with VM is that memory allocated in this way is not held in physical memory by default. It must be explicitly held, unlike memory in the system heap which the operating system automatically holds, at least in the first release of System Software 7.0.

When allocating memory above `BufPtr`, always use the equation defined in *Inside Macintosh*. The actual configuration of memory at boot time is much more complicated than the illustration indicates, especially with System Software 7.0 and VM. The System 7.0 boot code passes a specially-conditioned version of `MemTop` to system extensions, which guarantees that the equation has valid results. For this reason, do not use `MemTop` to determine the actual memory size of the machine; use `_Gestalt` instead. You may use `MemTop` to determine RAM size only if `_Gestalt` is not implemented, and then only at INIT time. (Apple continues to point out that good application software should not need to know this information except under extremely rare circumstances.)

Due to the way memory is organized with VM in 24-bit addressing, you may not be able to achieve nearly as much memory above `BufPtr` as you would think possible for a given virtual memory size. This is due to the possibility of VM fragmentation, which is discussed later. Without VM, the available space above `BufPtr` is generally somewhat less than half the amount of memory installed in the machine. With 24-bit VM, the available space may be



significantly less, and is probably far less than one half of the virtual memory size. The “conditioning” of the MemTop variable takes this into account.

## **Compatibility With 32-Bit Addressing**

To make the most valuable use of Virtual Memory, 32-bit addressing is extremely important. Needless to say, it is critical that all developers test their applications, drivers, and all other types of code extensively under System 7.0 while running 32-bit addressing—both with and without VM. Four megabit SIMMs are becoming less and less expensive, and the day is not far off when machines with at least 16 MB will be common. Correct behavior with 32-bit addressing is critical to the acceptance of both System 7.0 and developer applications. It is not acceptable to ask users to reboot with 24-bit addressing in order to use your hardware or software. For a few classes of applications it may be necessary to turn VM off in order to run **efficiently**, but VM should not prevent an application from running at all. Be sure to include a 'SIZE' resource in your application. It should proclaim your 32-bit compatibility to the world, not to mention the Finder.

## **User Tips and Helpful Hints for Living With VM**

Apple suggests that Virtual Memory runs more efficiently with at least four megabytes of physical RAM. Although System Software 7.0 runs on two-megabyte systems, using VM on such a system may result in unacceptable paging performance and hard disk thrashing. After holding the system heap and other RAM which must remain resident, there is simply not enough room left for efficient paging. Fortunately, with the recommended four or five megabytes, most users should be able to run arbitrarily large virtual memory environments, with little or no annoyance from paging delays and limited primarily by the sacrifice in disk space.

Virtual Memory trades virtual RAM size for some degree of performance. VM users should be aware that VM is not always a viable alternative to physical RAM. For example, an application which makes heavy use of an entire 8 MB partition for image processing may execute very sluggishly on a machine with only 4 MB of real RAM. (The benefit of VM in this case is that such an application runs at all on a machine with limited RAM.) On the other hand, the same machine may concurrently run six or seven different megabyte-plus applications with little or no appreciable performance degradation except when switching among them. (This is where VM really shines.) Performance is determined by virtual RAM size versus physical RAM size with the memory access dynamics of each application thrown in as a wild card. Each VM user will find a combination of settings which he or she finds most comfortable.

## **A Special Note Regarding 24-Bit VM**

Some machines in the installed base are capable of running VM, but do not have 32-bit clean ROMs and must run with 24-bit addressing. What this means to users who want to run VM

is that they can only take advantage of 14 MB of virtual memory. That's all there is room for in a 24-bit address map. More likely the limit is 12 or 13 MB because every installed NuBus card eliminates 1 MB of virtual RAM address space. (The way VM increases RAM size with 24-bit addressing is—more or less—by making each unused NuBus slot look like a 1 MB RAM card and making ROM and each installed NuBus card look like a nonrelocatable 1 MB application partition.)

You can be a real friend to the Process Manager (formerly known as MultiFinder) by taking care in which slots you install NuBus expansion cards: ROM always occupies one megabyte at \$800000, limiting the largest contiguous block of virtual memory to somewhat less than eight megabytes. The balance may be in a contiguous block as large as four or five megabytes unless it is fragmented by a poor selection of slots for expansion cards. Best results are achieved by placing all expansion cards in consecutive slots at either end of the bus—this has the effect of collecting all the immovable one megabyte rocks into a single pile where one is less likely to trip over them. Haphazard placement of NuBus cards may generate a number of one or two megabyte islands interspersed throughout the upper portion of the virtual memory space, and that does **not** help to run more applications or to manipulate larger objects.

In machines with fewer than six NuBus slots, recall that one “end” of the bus is actually in the middle of the slot address space. In a Macintosh IIcx, slots are numbered \$9 through \$B. Expansion cards should be installed from the lowest-numbered slot up (contiguous with the ROM) to avoid fragmentation. In a Macintosh IIfx, slots are numbered \$C through \$E. This poses a greater problem. Due to the RAM-based video in virtual slot \$B, it is nearly impossible to avoid some degree of fragmentation when using the built-in video option. When not using this option, installing NuBus cards from the highest-numbered slot down (at the end of memory) is the best course. Fortunately, the IIfx ROM supports 32-bit addressing. In 32-bit addressing VM, none of this discussion applies. Virtual Memory and NuBus do not share space in the 32-bit address map.

## A Template for GetPhysical Usage

A great deal of the justification for this code may be inferred from the code itself and the comments within. The basic rules are all covered in the previous text, but the simmered-down algorithm *sans* error handling is this:

```
See if there is _MemoryDispatch;
If there is _MemoryDispatch:
    LockMemory the interesting range of memory;
    If the memory is locked:
        Loop:
            Call GetPhysical on memory;
        Loop:
            Process a physical block;
        Until all physical blocks have been processed;
    Until all memory is translated;
    UnlockMemory the interesting range of memory;
Otherwise:
    Process the block of memory the way you used to;
End.
```

```
PROGRAM GetPhysicalUsage;

USES Types,Traps,Memory,
    Utilities;    { see DTS sample code for TrapAvailable }
    {In beta versions of the 7.0 interfaces, also use VMCalls, now in Memory.}

CONST
    kTestHandleSize = $100000;
```

```
VAR
    aHandle      : Handle;
    aPtr         : Ptr;
    aHandleSize  : LongInt;
    hasGetPhysical: Boolean;
    lockOK       : Boolean;
    vmErr        : OSErr;
    table        : LogicalToPhysicalTable;
    physicalEntryCount: 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
    aHandle := NewHandle(kTestHandleSize);
    IF aHandle <> NIL THEN BEGIN
        MoveHHI(aHandle);
        HLock(aHandle);
        aPtr := aHandle^;
        aHandleSize := GetHandleSize(aHandle);

        hasGetPhysical := TrapAvailable(_MemoryDispatch);
        { if GetPhysical is available it should always be used }
        { without it, DMA fails on IIci and many later machines }
        IF hasGetPhysical THEN BEGIN

            { must lock range before calling GetPhysical }
            { Call LockMemoryContiguous instead of LockMemory if a single
              physical block is required, but beware! This is inefficient and
              failure-prone! }
            vmErr := LockMemory {Contiguous} (aPtr,aHandleSize);
            lockOK := (vmErr = noErr);
            IF NOT lockOK THEN BEGIN
                { handle LockMemory error indicated by vmErr }
            END;

            IF lockOK THEN BEGIN
                table.logical.address := aPtr;
                table.logical.count := aHandleSize;
                vmErr := noErr;
                WHILE (vmErr = noErr) & (table.logical.count <> 0) DO BEGIN
                    physicalEntryCount := SizeOf(table) DIV SizeOf(MemoryBlock) - 1;
                    { this makes it easier to change "table" to include more }
                    { MemoryBlocks -- defaultPhysicalEntryCount is a suggestion }

                    vmErr := GetPhysical(table,physicalEntryCount);
                    { GetPhysical returns in physicalEntryCount the number }
                    { of physical entries actually used in the address table }

                    IF vmErr = noErr THEN BEGIN
                        FOR index := 0 TO (physicalEntryCount - 1) DO
                            WITH table DO
                                SendDMACmd(physical[index].address,physical[index].count);
                        END
                    ELSE BEGIN
                        { handle GetPhysical error indicated by vmErr }
                        { loop will terminate unless vmErr is negated }
                    END;
                END;
            END;
        END;
    END;
```

```
        END;

        { always unlock any range you lock! }
        IF Boolean (UnlockMemory(aPtr,aHandleSize)) THEN;  { ignore
UnlockMemory err }
        END;

    END
    ELSE
        { no GetPhysical, life is bliss }
        { remember how easy this used to be before GetPhysical? }
        SendDMACmd(aPtr,aHandleSize);
    END;

END.
```

---

**Further Reference:**

- *Inside Macintosh*, Volume II, Memory Manager
- *Inside Macintosh*, Volume IV, Initialization Resources
- *Inside Macintosh*, Volume VI, Compatibility Guidelines
- *Inside Macintosh*, Volume VI, Memory Management
- Technical Note M.ME.StripAddress —  
    \_StripAddress: The Untold Story
- Technical Note M.HW.Cache —  
    Cache As Cache Can

NuBus is a trademark of Texas Instruments

THINK is a trademark of Symantec Corporation