About the Memory Manager

On suitably equipped Macintosh computers, system 7.0 supports **32-bit addressing,** that is, the ability to use all 32 bits of a pointer or handle in determining memory addresses. Earlier versions of system software use 24-bit addressing, where the upper 8 bits of memory addresses are ignored or used as flag bits. In a 24-bit addressing scheme, the logical address space has a size of 16 MB. Because 8 MB of this total are reserved for I/O space, ROM, and slot space, the largest contiguous program address space is 8 MB. When 32-bit addressing is in operation in system 7.0, the maximum program address space is 1 GB.

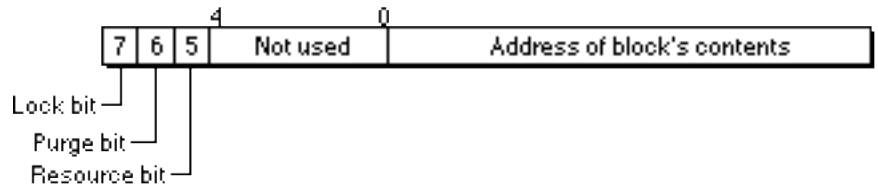
The ability to operate with 32-bit addressing is available only on certain models of the Macintosh, namely those with ROMs that contain a 32-bit **Memory Manager.** (For compatibility reasons, these ROMs also contain a 24-bit **Memory Manager**.) In order for your application to work when the machine is using 32-bit addressing, it must be **32-bit clean,** that is, able to run in an environment where all 32 bits of a memory address are significant. Fortunately, writing applications that are 32-bit clean is relatively easy if you follow Apple's guidelines. The major reason that some applications are not 32-bit clean is that their developers have ignored warnings not to manipulate data structures directly. In particular, the single most common reason that some applications are not 32-bit clean is that they manipulate bits in master pointers directly (for instance, to mark the associated memory blocks as locked or purgeable) instead of using **Memory Manager** routines that achieve the desired result.

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

The following four sections provide more explanation of how to make sure that your application is 32-bit clean, with detailed discussion of several additional issues, including customized window and control definition functions, and the **StripAddress** trap.

Using Master Pointers

The **Memory Manager** on the original Macintosh computers used a 24-bit addressing system. To the underlying hardware, only the lower 24 bits of a 32-bit address were significant. The upper 8 bits were always ignored in an address reference, a circumstance that led both system software developers and third-party software developers to put those 8 bits to some other use. For example, the **Memory Manager** itself took advantage of the upper 8 bits of an address in a master pointer to maintain information about heap blocks. (Master pointers are pointers to blocks of memory in the heap.) In the original Macintosh computers, the master pointers had a structure illustrated in the figure below.



THINK Reference © 1991-1992 Symantec Corporation

A master pointer structure in the 24-bit **Memory Manager**

Both A/UX and system 7.0 support 32-bit addressing, where all 32 bits of a memory address are significant. In this case, the flag bits in a master pointer must be stored elsewhere. Applications do not need to know where or how those flags are now stored if they use routines provided by the **Memory Manager** for setting and clearing those flags. (For example, to set or clear the Lock flag, you should use the procedures **HLock** and **HUnlock**.) If your application bypasses these routines and takes advantage of knowledge about the structure of master pointers to set and clear the flag bits directly, then it will not execute correctly in an environment where all 32 bits of the master pointer are significant. On such systems, setting or clearing the upper 3 bits of a master pointer directly does not change the flags but changes the address itself.

> **Note:** The issue of being 32-bit clean is not limited to direct manipulation of a master pointer's flag bits. Rather, *every* memory address must contain a full 32 bits. If, under systems using 24-bit addressing, you have used any of the upper 8 bits of pointers or handles for anything other than part of an address, then you must find an alternate representation for that information.

Using Window and Control Definition Functions

Two other times that you need to avoid 32-bit address violations are when using customized window definition functions (stored in resources of type 'WDEF') and when using customized control definition functions (stored in resources of type 'CDEF'). In earlier versions of system software, the **Window Manager** stored the window variant code number (which defines how the window looks) in the upper 8 bits of the handle to the window definition procedure. Under system 7.0, this is no longer true. As a result, if you need to retrieve the window variant code, use the **GetWVariant** function.

Similarly, the **Control Manager** used to store a variant code in the high bits of the control definition procedure handle. This also has changed under system 7.0, so you should use the **GetCVariant** function to retrieve the variant control value for a control.

A further problem arises if you define your own controls. To define a customized control, you need to provide a control definition function that interprets messages indicating what action your function is to perform. A customized control definition function is a function having the following declaration:

```
pascal long MyControl (short varCode, ControlHandle theControl, short
                       message, long param);
```

Previously, when passed the calcCRgns message, your control definition needed to test the high-order bit of the param parameter to determine whether to return a handle to the region of the control indicator or to the region of the entire control. In addition, your function had to clear the high bit of that parameter before passing back a handle to the calculated region. As a result, it was impossible to write a 32-bit clean control definition.
The **Control Manager** in system 7.0 has a new mechanism for instructing your control definition to calculate control regions. In particular, two new messages have been defined:

THINK Reference © 1991-1992 Symantec Corporation

calcCntlRgn //calculate control's region

calcThumbRgn //calculate indicator's region

When the 32-bit **Memory Manager** is in operation, the **Control Manager** uses one of these two new messages in cases where it would previously have used calcCRgns. When the 24-bit **Memory Manager** is in operation, the **Control Manager** still uses calcCRgns. If your application uses customized control definitions, you should update it to support the new messages as soon as possible. Because many users will still be running your application on machines with 24-bit addressing, you should also continue to support calcCRgns.

Manipulating 24-Bit and 32-Bit Memory Addresses

In environments where the machine might be executing with a 24-bit **Memory Manager**, it is sometimes necessary to strip off the flag bits of a memory address before you use that address. The **Memory Manager** provides the **StripAddress** function to allow you to do this.

The **StripAddress** function takes an address as a parameter and returns the value of the low-order 3 bytes if the 24-bit **Memory Manager** is operating. Otherwise, **StripAddress** returns the full 32-bit address unchanged.

> **Note:** This discussion replaces the information about the **StripAddress** function in the Operating System Utilities .

To appreciate the need for **StripAddress**, you need to distinguish between the addressing state of the **Memory Manager** and the addressing state of the underlying hardware. Macintosh II hardware, for example, is fully capable of handling all 32 bits of a memory address. For consistency with earlier Macintoshes, the Macintosh II CPU spends most of its time in 24-bit mode and ignores the high-order byte of all memory addresses. (The information encoded there is of significance only to the **Memory Manager**, not to the CPU.) However, a driver might temporarily switch the CPU to 32-bit mode to access a hardware address on a NuBus card. Suppose that this driver has an address in the heap (which is under the control of the **Memory Manager**) to which it wants to transfer data. If a 24-bit **Memory Manager** is in operation, the driver must call **StripAddress** on the heap address before using that address; otherwise, the CPU would interpret the high byte of that heap address as part of the address and (probably) transfer the data to the wrong address.

Even if you are not writing Macintosh drivers, you might still find it useful to call **StripAddress**. Occasionally, you need to compare two memory addresses (for example, two master pointers). If you're using the 24-bit **Memory Manager** and you compare those addresses without first stripping off the flag bits, you might end up with invalid results. You should call StripAddress to convert those addresses to the correct format.

As you can see, the operation of **StripAddress** is not dependent on the 24-bit or 32-bit state of the hardware, but on the 24-bit or 32-bit state of the **Memory Manager**. Calling **StripAddress** is necessary only when a 24-bit **Memory Manager** is operating. When a 32-bit **Memory Manager** is operating, **StripAddress** does nothing to addresses passed to it because those addresses are already valid 32-bit addresses.

THINK Reference © 1991-1992 Symantec Corporation

Sometimes drivers or other code must run in 32-bit mode to perform special hardware manipulations. You can use the **Translate24To32** function to translate 24-bit addresses into 32-bit addresses.

THINK Reference © 1991-1992 Symantec Corporation