Memory Management Utilities

Contents

The Memory Control Panel 4-3
About the Memory Management Utilities 4-5
The A5 Register 4-5
Addressing Modes 4-7
Address Translation 4-8
Processor Caches 4-8
Stale Instructions 4-9
Stale Data 4-10
Using the Memory Management Utilities 4-13
Accessing the A5 World in Completion Routines 4-14
Accessing the A5 World in Interrupt Tasks 4-16
Using QuickDraw Global Variables in Stand-Alone Code 4-18
Switching Addressing Modes 4-20
Stripping Flag Bits From Memory Addresses 4-21
Translating Memory Addresses 4-23
Memory Management Utilities Reference 4-24
Routines 4-24
Setting and Restoring the A5 Register 4-24
Changing the Addressing Mode 4-26
Manipulating Memory Addresses 4-27
Manipulating the Processor Caches 4-29
Summary of the Memory Management Utilities 4-34
Pascal Summary 4-34
Constants 4-34
Routines 4-34

C Summary 4-35
- Constants 4-35
- Routines 4-35

Assembly-Language Summary 4-36
- Trap Macros 4-36
- Global Variables 4-36

Result Codes 4-36

This chapter describes a number of utility routines you can use to control certain aspects of the memory environment in Macintosh computers. Some features of the memory environment are controlled by the user through the Memory control panel; others are controlled by the Process Manager or other parts of the Macintosh Operating System and Toolbox. The utility routines described in this chapter allow you to modify some of the normal operations of the Operating System or the Toolbox.

You need to read this chapter if your application or driver

- installs completion routines or interrupt tasks that are executed by the Operating System or Toolbox, not directly by your application
- modifies the addressing mode or converts addresses from one form to another
- moves executable code in memory, or performs DMA operations

To use this chapter, you should be familiar with the information in the chapter “Introduction to Memory Management” earlier in this book. Also, you can read the chapter “Introduction to Processes and Tasks” in *Inside Macintosh: Processes* for a related discussion of the A5 register.

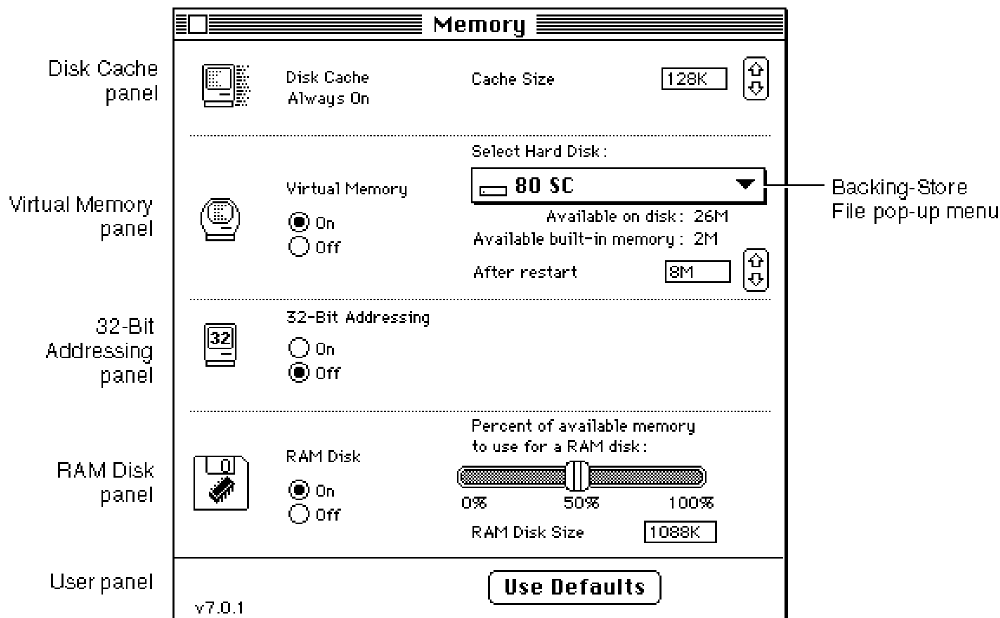
This chapter begins with a brief description of the Memory control panel, which allows users to alter several aspects of the Operating System’s memory configuration. Then it shows how you can use the Memory Management Utilities to

- set up the A5 register so that your application-defined completion routines and interrupt tasks can access your application’s global variables
- get the value of the A5 register so that you can read your application’s QuickDraw global variables from within stand-alone code
- get or set a computer’s address-translation mode
- strip the flag bits from a master pointer or other memory address
- convert 24-bit addresses to 32-bit addresses
- flush the microprocessor’s instruction and data caches

The Memory Control Panel

A user can alter several aspects of the system memory configuration by setting certain controls in the Memory control panel. This panel contains controls governing the operation of the disk cache, virtual memory, and the addressing mode used by the Memory Manager. Figure 4-1 shows the Memory control panel.

Figure 4-1 The Memory control panel



The Disk Cache panel replaces the HFS RAM Cache panel (part of the General control panel) used in earlier versions of system software. A **disk cache** is a part of RAM that acts as an intermediate buffer when data is read from and written to file systems on secondary storage devices. Data is saved there in case it is needed again in the very near future. If it is, the Operating System reads the data from the disk cache rather than the secondary storage device (which would take considerably longer). By increasing the cache size, the user increases the likelihood that data recently read from or written to the file system will be in the cache. The controls in the Disk Cache panel allow the user to configure the size of the disk cache used by the Operating System during file-access operations. In system software version 7.0, unlike earlier versions, the user cannot turn off disk caching.

In system software version 7.0, the minimum cache size is 16 KB. The default size is 32 KB per megabyte of installed RAM (thus, the default disk cache size for a computer with 4 MB of RAM is 128 KB). The maximum disk cache size is 320 KB per megabyte of installed RAM (thus, the maximum disk cache size for a computer with 4 MB of RAM is 1280 KB). The operation of the disk cache is completely transparent to your application.

Note

These cache size values are provided for informational purposes only and may differ in later system software versions or on different Macintosh computers. In addition, the use of RAM for a RAM-based video interface or a RAM disk affects the amount of RAM available for the disk cache. ◆

The Virtual Memory panel allows the user to set various features of virtual memory, including whether virtual memory is turned on and, if so, how much is available. The user can also specify the volume of the **backing-store file,** in which the Virtual Memory Manager stores unused portions of code and data. Changes to the virtual memory configuration do not take effect until the user restarts the computer. Note that the Virtual Memory panel appears only on computers that support virtual memory. For information on how your application can interact with virtual memory, see the chapter "Virtual Memory Manager" in this book.

Using the 32-Bit Addressing controls, the user can select the maximum size of the address space used in the computer. The maximum size of the address space is determined by the number of bits used to store memory addresses, as explained in the chapter "Virtual Memory Manager" in this book. The 32-Bit Addressing panel appears only on computers that support 32-bit addressing mode. By clicking the panel's controls, the user can turn 32-bit addressing off and on. Changes made in this panel do not go into effect until the user restarts the computer.

Using the RAM Disk controls, the user can determine the amount of the available RAM that is to be treated as a **RAM disk,** a portion of RAM reserved for use as a temporary storage device. It is most useful to create a RAM disk on battery-powered computers (such as the Macintosh PowerBook computers) because the computer uses less energy to access RAM than to access a hard disk or a floppy disk.

About the Memory Management Utilities

You can use the Memory Management Utilities to ensure that

- your application's callback routines, interrupt tasks, and stand-alone code can access application global variables or QuickDraw global variables
- your application or driver functions properly in both 24- and 32-bit modes
- data or instructions in the microprocessor's internal caches remain consistent with data or instructions in RAM

This section explains when and why you might need to use these utilities; for actual implementation details, see the section "Using the Memory Management Utilities," which begins on page 4-13.

The A5 Register

If you write code that accesses your application's A5 world (usually to read or write the application global variables) at a time that your application is not the current application, you must ensure that the A5 register points to the boundary between your application's parameters and global variables. Because the Operating System accesses your A5 world relative to the address stored in the A5 register, you can obtain unpredictable results if you attempt to read or write data in your A5 world when the contents of A5 are not valid.

There are two general cases in which code might execute when the contents of the A5 register are invalid:

- when you install a completion routine that is executed when some other operation (for instance, writing data to disk or playing a sound) is completed
- when you install a routine (for instance, a VBL task) that is called in response to an interrupt

If you install code that is to be executed at either of these times, you must make sure to set up the A5 register upon entry and to restore it before exit. The sections "Accessing the A5 World in Completion Routines" on page 4-14 and "Accessing the A5 World in Interrupt Tasks" on page 4-16 describe how to do this in each case.

You might also need to determine the location of your application's A5 world if you want to read information in it from within a stand-alone code segment. You might want to do this in application-defined definition procedures called on behalf of your application. These include

- control definition functions
- window definition functions
- menu definition functions

The problem with these kinds of stand-alone code segments is *not* that the value in the A5 register is incorrect at the time they are executed; rather, it is that they have no A5 world at all. During execution, these stand-alone code segments can effectively "borrow" the A5 world of the current application. However, they must be compiled and linked separately from your application. (A custom window definition procedure, for example, is separately compiled and linked, and then included as a resource of type `'WDEF'` in your application's resource fork.) The linker cannot resolve any offsets from the value in the A5 register, because the code segment doesn't have an A5 world.

A stand-alone code segment can solve this problem quite simply at run time, by determining the location of your application's A5 world and then copying the data it needs to access into blocks of memory that it allocates itself. In the code segment, all references to data in the A5 world are indirect: the code segment manipulates local copies of the relevant data. Using this technique, you can avoid explicit symbolic references to the A5 world, which the linker cannot resolve.

In theory, you could use this technique of copying global data into a stand-alone code segment's private storage to access any data contained in your application's A5 world. In practice, however, the A5 world can contain so much data that you wouldn't want to make local copies of it all. In addition, the precise organization of the entire A5 world is not generally determinate. Usually, a custom definition procedure or other stand-alone code segment needs to read only the QuickDraw global variables, which are of fixed size and have a well-documented organization. See the section "Using QuickDraw Global Variables in Stand-Alone Code" on page 4-18 for a complete description of how to read your application's QuickDraw global variables from within a stand-alone code segment.

Addressing Modes

The Memory Manager on the original Macintosh computers uses a 24-bit addressing mode. To the underlying hardware, only the lower 24 bits of any 32-bit address are significant. The CPU effectively ignores the upper 8 bits in a memory address by using a 24-bit address-translation mode. In this mode, the CPU (or the MMU coprocessor, if present) maps all addresses to their lower-order 24 bits whenever it reads or writes a memory location. This led both system software developers and third-party software developers to put those upper 8 bits to other uses. For example, the Memory Manager itself uses the upper 8 bits of the address in a master pointer to maintain information about the associated relocatable block. These upper 8 bits are known as **master pointer flag bits.**

When the Operating System is running in 24-bit mode, you can address at most 1 MB of the address space assigned to a NuBus expansion card. Some cards, however, can work with far more than 1 MB of memory. As a result, a device driver might need to switch the Operating System into 32-bit mode temporarily, so that it can access the entire address range of the associated device (perhaps to copy data from the device's RAM into the heap). When 32-bit address translation is enabled, the CPU or the MMU does not ignore the upper 8 bits of a memory address.

Note

Don't confuse the current address-translation mode of the Macintosh hardware with the current addressing state of the Memory Manager. The addressing state of the Memory Manager is selectable on a per-boot basis and cannot be changed by an application or driver. The address-translation mode of the underlying hardware is controlled by the CPU and MMU (if one is available) and can be changed, if necessary, at any time. ◆

The Operating System provides two utilities, `GetMMUMode` and `SwapMMUMode`, that allow you to get and set the current address-translation mode. See "Switching Addressing Modes" on page 4-20 for details.

If your device driver does in fact temporarily set the Macintosh hardware into 32-bit address-translation mode, you need to be careful when you pass addresses to the associated device. Suppose, for example, that your driver wants to transfer data to an address in the heap (which is under the control of the Memory Manager). If the 24-bit Memory Manager is in operation, you need to strip the high byte from the memory address; otherwise, the CPU would interpret the high byte of flags as part of the address and transfer the data to the wrong location.

Note

You might also need to make the block of memory in the heap immovable in physical memory, so that it is not paged out under virtual memory. See the discussion of locking memory in the chapter "Virtual Memory Manager" in this book. ◆

The Operating System provides the `StripAddress` function, which you can use to **strip** the high-order byte from a memory address. Even if you are not writing Macintosh drivers, you might still find it useful to call `StripAddress`. For example, suppose you need to compare two memory addresses (two master pointers, perhaps). If the system is running the 24-bit Memory Manager and you compare those addresses without first clearing the flag bits, you might get invalid results. You should first call `StripAddress` to convert those addresses to their correct format before comparing them.

As you can see, the operation of `StripAddress` is not dependent on the 24-bit or 32-bit address translation state of the hardware, but on the 24-bit or 32-bit addressing state of the Memory Manager. You need to call `StripAddress` only when the 24-bit Memory Manager is operating. When the 32-bit Memory Manager is operating, `StripAddress` returns unchanged any addresses passed to it, because they are already valid 32-bit addresses. See “Stripping Flag Bits From Memory Addresses” on page 4-21 for complete details on calling `StripAddress`.

Address Translation

When a driver or other software component switches the system to the 32-bit address-translation mode (perhaps to manipulate special hardware on a slot device), certain addresses normally accessible in 24-bit mode are not mapped to the same location by the Macintosh hardware. In particular, the Virtual Memory Manager uses some of the slot address space as part of the addressable RAM. In that case, the standard 24-to-32 bit translation is not valid for slot spaces that the MMU has remapped into the application address space.

You can use the `Translate24To32` function to translate 24-bit addresses that might have been remapped by the Macintosh hardware. If you intend to use 24-bit addresses when your software is executing in 32-bit mode, your code should check for the presence of that function. If it is available, you should use it to map 24-bit addresses into the 32-bit address space. For details, see “Translating Memory Addresses” on page 4-23.

Processor Caches

Some members of the Motorola MC680x0 family of microprocessors contain internal caches that can significantly improve the overall performance of software executing on those microprocessors. For example, the MC68020 microprocessor contains a 256-byte on-board **instruction cache,** an area of memory within the microprocessor that stores the most recently executed instructions. Whenever the processor needs to fetch an instruction, it first checks the instruction cache to determine whether the word required is in the cache. The operation is much faster when the information is in the cache than when it is only in RAM (which is external to the microprocessor).

Some other members of the MC680x0 family of microprocessors also contain an internal **data cache,** an area of memory that holds recently accessed data. The data cache operates much as the instruction cache does, but it caches data instead of instructions. Before reading data from RAM, the microprocessor checks the data cache to determine whether

the operand required for an instruction is in the cache. Again, the overall performance of the software is greatly increased by the operation of the data cache.

Table 4-1 lists the available caches and their sizes for the various microprocessors currently used in Macintosh computers.

Table 4-1 Caches available in MC680x0 microprocessors

Microprocessor	Instruction cache?	Data cache?
MC68000	No	No
MC68020	Yes (256 bytes)	No
MC68030	Yes (256 bytes)	Yes (256 bytes)
MC68040	Yes (4 KB)	Yes (4 KB)

The operation of any available instruction and data caches is generally transparent to your application. In certain cases, however, you need to make sure that the information in the caches and the corresponding information in main memory remain consistent. When some information in RAM changes but the corresponding information in the cache does not, the cached information is said to be **stale.** The following two sections describe in detail how cached instructions and data can become stale. You can avoid using stale instructions or data by **flushing** the affected cache whenever you do something that can cause instructions or data to become stale. See "Manipulating the Processor Caches," beginning on page 4-29, for routines that you can use to maintain consistency between a cache and main memory.

Stale Instructions

Any time that you modify part of the executable code of your application or other software, you risk creating **stale instructions** in the instruction cache. Recall that the microprocessor stores the most recently executed instructions in its internal instruction cache, separately from main memory. Whenever your code modifies itself or any data in memory that contains executable code, there is a possibility that a copy of the modified instructions will be in the instruction cache (because they were executed recently). If so, attempting to execute the modified instructions actually results in the execution of the cached instructions, which are stale.

You can avoid using stale instructions by flushing the instruction cache every time you modify executable instructions in memory. Flushing the cache invalidates all entries in it and forces the processor to refill the cache from main memory.

IMPORTANT

Flushing the instruction cache has an adverse effect on the CPU's performance. You should flush the instruction cache only when absolutely necessary. ▲

Any code that modifies itself directly is likely to create stale instructions in the instruction cache. In addition, you can create stale instructions by modifying other parts of memory that contain executable instructions. For example, if you modify jump table entries, you'll need to flush the instruction cache to avoid using stale instructions. Similarly, if you install patches by copying code from one part of memory to another and modifying `JMP` instructions in order to execute the original routine, you'll need to flush the instruction cache. See the description of the `FlushInstructionCache` procedure on page 4-30 for details.

The system software automatically flushes the instruction cache when you call certain traps that are often used to move code from one location to another in memory. The system flushes the instruction cache whenever you call `_BlockMove`, `_Read`, `_LoadSeg`, and `_UnloadSeg`.

▲ **WARNING**

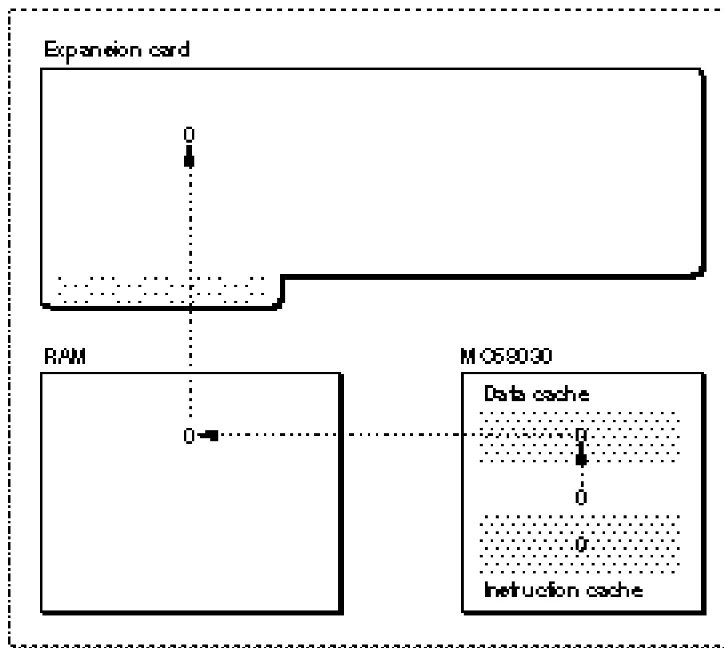
The `_BlockMove` trap is not guaranteed to flush the instruction cache for blocks that are 12 bytes or smaller. If you use `_BlockMove` to move very small blocks of code, you should flush the instruction cache yourself. ▲

Other traps may flush the instruction cache. In general, you need to worry about stale instructions only when your application moves code and not when the system software moves it.

Stale Data

A cache may contain **stale data** whenever information in RAM is changed and that information is already cached in the microprocessor's data cache. Suppose, for example, that a computer contains an expansion card capable of DMA data transfers from the card to main memory. The card typically reads commands from a buffer in RAM, executes the commands, and writes status information back to the buffer when the command completes. Before the card reads a command, the CPU sets up the command buffer and initializes the status code to 0. Figure 4-2 shows this situation on a computer with an MC68030 microprocessor.

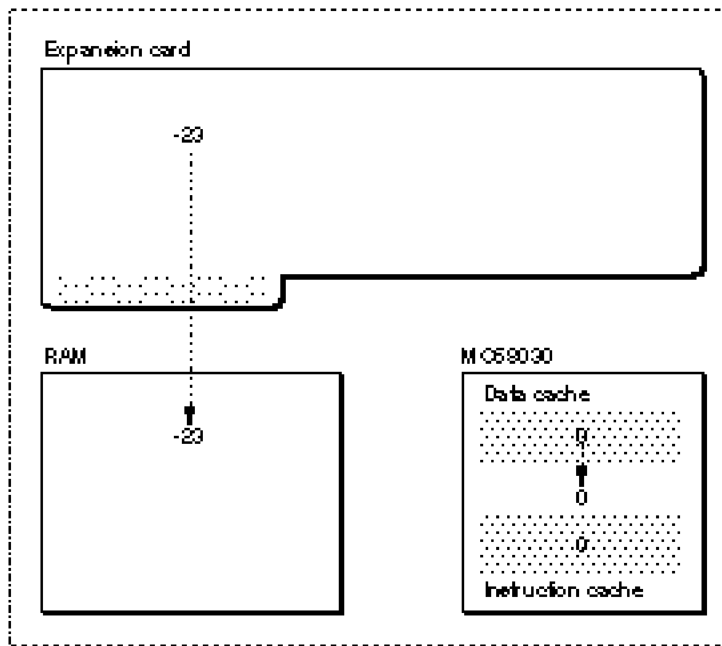
Figure 4-2 Initializing a status code



The MC68030 has a **write-through cache:** any data written to the cache is immediately written out to RAM (to avoid stale data in RAM). As a result, the cache and RAM both contain the same value (0) for the status code. Suppose next that the expansion card executes the first command and writes a nonzero status code to RAM. The card then sends an interrupt to the CPU, indicating that the operation has completed.

At this point, the microprocessor might attempt to read the status code returned by the external hardware. However, because the status code is in the microprocessor's data cache, the CPU reads the value in the cache, which is stale, instead of the value in main memory (see Figure 4-3).

Figure 4-3 Reading stale data



To avoid using this stale data, have your driver flush the data cache whenever you transfer data directly into main memory.

IMPORTANT

Flushing the data cache has an adverse effect on the CPU's performance. You should flush the data cache only when absolutely necessary. ▲

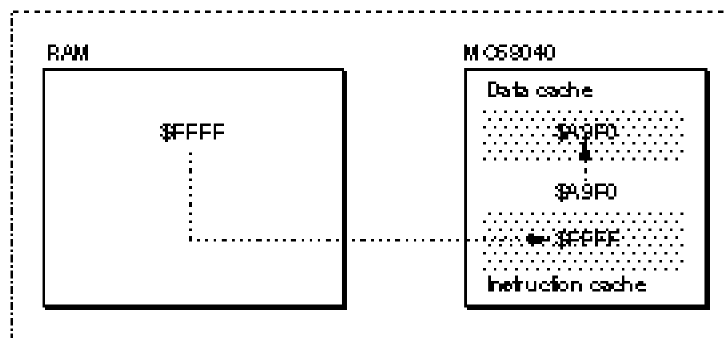
The MC68040 has a **copy-back cache:** any data written to the cache is written to RAM only when necessary to make room in the cache for data accessed more recently or when the cache is explicitly flushed. As you can see, a copy-back cache allows for even greater performance improvements than a write-through cache, because the data in the cache has to be written to main memory less often. This is extremely valuable for relatively small amounts of data that are needed for only a short while, such as local stack frames for C or Pascal function calls.

Because the data in a copy-back cache is written to main memory only in certain circumstances, it's possible to get stale data in RAM. If you write data that is to be read by non-CPU devices (such as an expansion card that performs DMA operations), you need to flush the data cache before instructing the alternate bus master to read that data. If you don't update the RAM, the DMA transfer from RAM will read stale data.

A copy-back data cache can also lead to the use of invalid instructions if the stale data in RAM contains executable code. When fetching instructions, the CPU looks only in the instruction cache and (if necessary) in main memory, not in the data cache. Because the instruction and data caches are separate, it's possible that the CPU will fetch invalid instructions from memory, in the following way. Suppose that you alter some

jump table entries and, in doing so, write the value $A9F0 (that is, the trap number of the _LoadSeg trap) to memory. If the data cache is a copy-back cache, the data in main memory is not updated immediately, but only when necessary to make room in the cache (or when you explicitly flush the cache). As a result, the CPU might read invalid instructions from memory when attempting to execute a routine whose jump table entry you changed. Figure 4-4 illustrates this problem.

Figure 4-4 Reading invalid instructions



To avoid reading invalid instructions in this way, you need to flush the data cache before calling any routines whose jump table entries you've altered. More generally, whenever you need to flush the instruction cache, you also first need to flush the data cache—but only if you've changed any executable code and those changes might not have been written to main memory.

Another way to avoid using stale data is to prevent the data from being cached (and hence from becoming stale). The Virtual Memory Manager function LockMemory locks a specified range of pages in physical RAM and either disables the data cache or marks the specified pages as noncacheable (depending on what's possible and what makes the most sense). Accordingly, you need not explicitly flush the processor's data cache for data buffers located in pages that are locked in memory. See the chapter "Virtual Memory Manager" in this book for more information about locking page ranges.

Using the Memory Management Utilities

This section describes how you can

- save and restore the value of the A5 register so that you can access your application's A5 world in completion routines or other interrupt tasks
- access your application's QuickDraw global variables from within stand-alone code
- change the address-translation mode so that you can temporarily use 32-bit addresses

- strip the flag bits from a master pointer or other memory address
- convert 24-bit addresses to 32-bit addresses

Accessing the A5 World in Completion Routines

Some Toolbox and Operating System routines require you to pass the address of an application-defined **callback routine,** usually in a variable of type `ProcPtr`. After a certain condition has been met, the Toolbox executes the specified routine. The exact time at which the Toolbox executes the routine varies. The timing of execution is determined by the Toolbox routine to which you passed the routine's address and the action that must be completed before the routine is called.

Callback routines are quite common in the Macintosh system software. A grow-zone function, for instance, is an application-defined callback routine that is called every time the Memory Manager cannot find enough space in your heap to honor a memory-allocation request. Similarly, if your application plays a sound asynchronously, you can have the Sound Manager execute a **completion routine** after the sound is played. The completion routine might release the sound channel used to play the sound or perform other cleanup operations.

In general, you cannot predict what your application will be doing when an asynchronous completion or callback routine is actually executed. The routine could be called while your application is executing code of its own or executing another Toolbox or Operating System routine.

Note

The completion or callback routine might even be called when your application is in the background. Before executing a completion or callback routine belonging to your application, the Process Manager checks whether your application is in the foreground. If not, the Process Manager performs a minor switch to give your application temporary control of the CPU. ◆

Many Toolbox and Operating System routines do not need to access the calling application's global variables, QuickDraw global variables, or jump table. As a result, they sometimes use the A5 register for their own purposes. They save the current value of the register upon entry, modify the register as necessary, and then restore the original value on exit. As you can see, if one of these routines is executing when your callback routine is executed, your callback routine cannot depend on the value in the A5 register. This effectively prevents your callback routine from using any part of its A5 world.

To solve this problem, simply use the strategy that the Toolbox employs when it takes over the A5 register: save the current value in the A5 register at the start of your callback procedure, install your application's A5 value, and then restore the original value when you exit. Listing 4-1 illustrates a very simple grow-zone function that uses this technique. It uses the `SetCurrentA5` and `SetA5` utilities to manipulate the A5 register.

Listing 4-1 A sample grow-zone function

```
FUNCTION MyGrowZone (cbNeeded: Size): LongInt;
VAR
   theA5:   LongInt;              {value of A5 when function is called}
BEGIN
   theA5 := SetCurrentA5;         {remember current value of A5; install ours}
   IF (gEmergencyMemory^ <> NIL) & (gEmergencyMemory <> GZSaveHnd) THEN
      BEGIN
         EmptyHandle(gEmergencyMemory);
         MyGrowZone := kEmergencyMemorySize;
      END
   ELSE
      MyGrowZone := 0;            {no more memory to release}
   theA5 := SetA5(theA5);         {restore previous value of A5}
END;
```

The function `SetCurrentA5` does two things: it returns the current value in the A5 register, and it sets the A5 register to the value of the `CurrentA5` low-memory global variable. This global variable always contains a value that points to the boundary between the current application's parameters and its global variables. The `MyGrowZone` function defined in Listing 4-1 calls `SetCurrentA5` on entry to make sure that it can read the value of the `gEmergencyMemory` global variable.

The function `SetA5` also does two things: it returns the current value in the A5 register, and it sets the A5 register to whatever value you pass to the function. The `MyGrowZone` function calls `SetA5` with the original value of the A5 register as the parameter. In this case, the value returned by `SetA5` is ignored.

There is no way to test whether, at the time your callback routine is called, your application is executing a Toolbox routine that could change the A5 register. Therefore, to be safe, you should save and restore the A5 register in any callback routine that accesses any part of your A5 world. Such routines include

- grow-zone functions
- Sound Manager completion routines
- File Manager I/O completion routines
- control-action procedures
- TextEdit word-break and click-loop routines
- trap patches
- custom menu definition, window definition, and control definition procedures

See the section of *Inside Macintosh* describing any particular completion or callback routine for details on whether you need to save and restore the A5 register in this way.

Accessing the A5 World in Interrupt Tasks

Sometimes, an application-defined routine executes at a time when you can't reliably call `SetCurrentA5`. For example, if your application is not the current application and you call `SetCurrentA5` as illustrated in Listing 4-1, the function will not return your application's value of `CurrentA5`. The `SetCurrentA5` function always returns the value of the low-memory global variable `CurrentA5`, which always belongs to the *current* application. You'll end up reading some other application's A5 world.

In general, you cannot reliably call `SetCurrentA5` in any code that is executed in response to an interrupt, including the following:

- Time Manager tasks
- VBL tasks
- tasks installed using the Deferred Task Manager
- Notification Manager response procedures

Instead of calling `SetCurrentA5` at interrupt time, you can call it at noninterrupt time when yours is the current application. Then store the returned value where you can read it at interrupt time. For example, the Notification Manager allows you to store information in the notification record passed to `NMInstall`. When you set up a notification record, you can use the `nmRefCon` field to hold the value in the A5 register. Listing 4-2 illustrates how to save the current value in the A5 register and pass that value to a response procedure.

Listing 4-2 Passing A5 to a notification response procedure

```
VAR
   gMyNotification:  NMRec;          {a notification record}

BEGIN
   WITH gMyNotification DO
   BEGIN
      qType := ORD(nmType);          {set queue type}
      nmMark := 1;                   {put mark in Application menu}
      nmIcon := NIL;                 {no alternating icon}
      nmSound := Handle(-1);         {play system alert sound}
      nmStr := NIL;                  {no alert box}
      nmResp := @SampleResponse;     {set response procedure}
      nmRefCon := SetCurrentA5;      {pass A5 to notification task}
   END;
END;
```

The key step is to save the value of `CurrentA5` where the response procedure can find it—in this case, in the `nmRefCon` field. You must call `SetCurrentA5` at noninterrupt time; otherwise, you cannot be certain that it will return the correct value.

When the notification response procedure is executed, its first task should be to call the `SetA5` function, which sets register A5 to the value stored in the `nmRefCon` field. At the end of the routine, the notification response procedure should call the `SetA5` function again to restore the previous value of register A5. Listing 4-3 shows a simple response procedure that sets up the A5 register, modifies a global variable, and then restores the A5 register.

Listing 4-3 Setting up and restoring the A5 register at interrupt time

```
PROCEDURE SampleResponse (nmReqPtr: NMRecPtr);
VAR
   oldA5:      LongInt;       {A5 when procedure is called}
BEGIN
   oldA5 := SetA5(nmReqPtr^.nmRefCon);
                              {set A5 to the application's A5}
   gNotifReceived := TRUE;    {set an application global }
                              { to show alert was received}
   oldA5 := SetA5(oldA5);     {restore A5 to original value}
END;
```

Note

Many optimizing compilers (including MPW) might put the address of a global variable used by the interrupt routine into a register before the call to `SetA5`, thereby possibly generating incorrect references to global data. To avoid this problem, you can divide your completion routine into two separate routines, one to set up and restore A5 and one to do the actual completion work. Check the documentation for your development system to see if this division is necessary, or contact Macintosh Developer Technical Support. ◆

Several of the other managers that you can use to install interrupt code—including the Deferred Task Manager, the Time Manager, and the Vertical Retrace Manager—do not include a reference constant field in their task records. Therefore, if you wish to access global variables from within one of these tasks, you must use another mechanism to attach the value of the A5 register to the task record.

To do this, you can define a new record that contains the task record and your own reference constant field. You can initialize the task record as you normally would and then copy the value of your application's A5 register into the reference constant field you created. Then, when you obtain a pointer to the task record at interrupt time, you can use your knowledge of the size of the task record to compute the location of your reference constant field. See the chapters "Time Manager" and "Vertical Retrace Manager" in *Inside Macintosh: Processes* for detailed illustrations of these techniques.

Using QuickDraw Global Variables in Stand-Alone Code

If you are writing a stand-alone code segment such as a definition procedure for a window, menu, or control, you might want routines in that segment to examine the QuickDraw global variables of the current application. For example, you might want a control definition function to reference some of the QuickDraw global variables, such as `thePort`, `screenBits`, or the predefined patterns. Stand-alone segments, however, have no A5 world; if you try to link a stand-alone code segment that references your application's global variables, the linker may be unable to resolve those references.

To solve this problem, you can have the definition function find the value of the application's A5 register (by calling the `SetCurrentA5` function) and then use that information to copy all of the application's QuickDraw global variables into a record in the function's own private storage. Listing 4-4 defines a record type with the same structure as the QuickDraw global variables. Note that `randSeed` is stored lowest in memory and `thePort` is stored highest in memory.

Listing 4-4 Structure of the QuickDraw global variables

```
TYPE
   QDVarRecPtr = ^QDVarRec;
   QDVarRec =
   RECORD
      randSeed:   LongInt;          {for random-number generator}
      screenBits: BitMap;           {rectangle enclosing screen}
      arrow:      Cursor;           {standard arrow cursor}
      dkGray:     Pattern;          {75% gray pattern}
      ltGray:     Pattern;          {25% gray pattern}
      gray:       Pattern;          {50% gray pattern}
      black:      Pattern;          {all-black pattern}
      white:      Pattern;          {all-white pattern}
      thePort:    GrafPtr;          {pointer to current GrafPort}
   END;
```

The location of these variables is linker-dependent. However, the A5 register always points to the last of these global variables, `thePort`. The Operating System references all other QuickDraw global variables as negative offsets from `thePort`. Therefore, you must dereference the value in A5 (to obtain the address of `thePort`), and then subtract the combined size of the other QuickDraw global variables from that address. The difference is a pointer to the first of the QuickDraw global variables, `randSeed`. You can copy the entire record into a local variable simply by dereferencing that pointer, as illustrated in Listing 4-5.

Listing 4-5 Copying the QuickDraw global variables into a record

```
PROCEDURE GetQDVars (VAR qdVars: QDVarRec);
TYPE
   LongPtr = ^LongInt;
BEGIN
   qdVars := QDVarRecPtr(LongPtr(SetCurrentA5)^ -
                          (SizeOf(QDVarRec) - SizeOf(thePort)))^;
END;
```

Thereafter, your stand-alone code segment can read QuickDraw global variables through the structure returned by `GetQDVars`. Listing 4-6 defines a very simple draw routine for a control definition function. After reading the calling application's QuickDraw global variables, the draw routine paints a rectangle with a pattern.

Listing 4-6 A control's draw routine using the calling application's QuickDraw patterns

```
PROCEDURE DoDraw (varCode: Integer; myControl: ControlHandle;
                  flag: Integer);
VAR
   cRect: Rect;
   qdVars: QDVarRec;
   origPenState: PenState;
CONST
   kDraw = 1;                        {constant to specify drawing}
BEGIN
   GetPenState(origPenState);        {get original pen state}
   cRect := myControl^^.contrlRect; {get control's rectangle}
   IF flag = kDraw THEN
      BEGIN
         GetQDVars(qdVars);          {patterns are QD globals}
         PenPat(qdVars.gray);        {install desired pattern}
         PaintRect(cRect);           {paint the control}
      END;
   SetPenState(origPenState);        {restore original pen state}
END;
```

The `DoDraw` drawing routine defined in Listing 4-6 retrieves the calling application's QuickDraw global variables and paints the control rectangle with a light gray pattern. It also saves and restores the pen state, because the `PenPat` procedure changes that state.

Switching Addressing Modes

If you are writing a driver for a slot-card device, you can use the `SwapMMUMode` procedure to change to 32-bit address-translation mode temporarily, as follows:

```
myMode := true32b;                {specify switch to 32-bit mode}
SwapMMUMode(myMode);              {perform switch}
```

The parameter passed to `SwapMMUMode` must be a variable that is equal to the constant `false32b` or the constant `true32b`.

```
CONST
   false32b    = 0;               {24-bit addressing mode}
   true32b     = 1;               {32-bit addressing mode}
```

The `SwapMMUMode` procedure switches to the specified mode and then changes the parameter to indicate the mode previously in use. Thereafter, you can restore the previous address-translation mode by again calling

```
SwapMMUMode(myMode);
```

Note

You should switch to 32-bit mode only if the computer supports 32-bit addressing. To find out whether a system supports 32-bit mode and whether a system started up in 32-bit mode, use the `Gestalt` function, described in the chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities*. To determine the current address-translation mode, call the `GetMMUMode` function. ◆

If you do call `SwapMMUMode`, be careful to avoid situations that can cause the system to read an invalid address from the program counter. When the system is in 24-bit mode and you load a code resource into a block of memory (for example, by calling `GetResource`), the high byte of that block's master pointer contains Memory Manager flag bits. If you try to execute that code by performing an assembly-language `JSR` instruction (typically `JSR (A0)`, with the master pointer in register A0), the entire master pointer is translated directly into the program counter. This, however, is not a valid 32-bit address. As soon as you switch to 32-bit mode, the program counter contains an invalid value. This is virtually certain to cause the system to crash.

Note

This problem can arise when you change to 32-bit mode in code loaded from a resource or placed into a block of memory that was allocated by calls to Memory Manager routines. It does not arise with standard `'CODE'` resources because the Segment Manager fixes the program counter. ◆

To avoid this problem, simply call StripAddress on the address in the program counter before you call SwapMMUMode. Listing 4-7 shows one way to do this.

Listing 4-7 Stripping the program counter

```
PROCEDURE FixPC;
   INLINE   $41FA, $000A,   {LEA *+$000C,A0}
            $2008,          {MOVE.L A0,D0}
            $A055,          {_StripAddress}
            $2040,          {MOVEA.L D0,A0}
            $4ED0;          {JMP (A0); jump to next instruction}
```

For these same reasons, you also need to call StripAddress on any address you pass to the _SetTrapAddress trap, if the address references a block in your application heap.

Stripping Flag Bits From Memory Addresses

If your code runs on a system that might have started up with the 24-bit Memory Manager, you sometimes need to **strip** the flag bits from a memory address before you use it. The Operating System provides the StripAddress function for this purpose.

The StripAddress function takes an address as a parameter and returns the value of the address's low-order 3 bytes if the computer started up in 24-bit mode. If the system started up in 32-bit mode, StripAddress returns the address unchanged (because it must already be a valid 32-bit address). Note that if a system starts up in 32-bit mode, you cannot switch it to 24-bit mode.

▲ **WARNING**
If you pass a valid 32-bit address to StripAddress and the computer started in 24-bit mode, the function still strips off the high byte of the address, thus probably rendering the address invalid. You can pass 32-bit addresses to StripAddress if the system started up in 32-bit mode, but then the function does nothing to the address. Therefore, you should ordinarily pass only 24-bit addresses to the StripAddress function. ▲

You need to use StripAddress primarily in device drivers or other software that communicates heap addresses to external hardware (such as a NuBus card). Because the external hardware might interpret the flag bits of a master pointer as part of the address, you need to call StripAddress to clear those flag bits.

There is nothing inherently dangerous about 24-bit addresses. They cause problems only when you try to use them in 32-bit mode. So, unless you are switching addressing modes (by calling SwapMMUMode), you generally don't need to call StripAddress.

You might, however, need to call `StripAddress` in these special cases, even if you are not designing a driver:

- Making ordered address comparisons. If you want to sort an array by address or do any other kind of ordered address comparison (that is, using <, >, ≥, or ≤), you need to call `StripAddress` on each address before the comparison. Even though the CPU uses only the lower 3 bytes when it determines memory addresses in 24-bit mode, it uses all 32 bits when it performs arithmetic operations.
- Comparing master pointers. If you want to perform any type of comparison on master pointers (that is, on dereferenced handles), you must first call `StripAddress` on each address. The master pointer flag bits can change at any time, so you need to clear them before making the comparison. In general, you should call `StripAddress` when comparing any two pointers, if either of them might be a dereferenced handle.
- Accessing addresses in 32-bit mode. If you switch the computer to 32-bit mode manually, you need to call `StripAddress` on all 24-bit pointers and handles that you access while in 32-bit mode. Be careful, however, not to call `StripAddress` on a valid 32-bit address.
- Fixing the program counter. You might need to use `StripAddress` to fix the value of the program counter before you switch manually to 32-bit mode. See "Switching Addressing Modes" on page 4-20 for details.
- Overcoming Resource Manager limitations. To avoid a limitation in the Resource Manager's `OpenResFile` and `OpenRFPerm` routines, you should call `StripAddress` on pointers to the filenames that you pass to those functions, but only if the strings that represent the files are hard-coded into your application's code instead of in a separate resource. When the string is embedded in a code resource, the Resource Manager calls the `RecoverHandle` function with an invalid master pointer. Here is an example of the correct way to call `OpenResFile`:

```
fileName := 'This file';
myRef := OpenResFile(StringPtr(StripAddress(@fileName))^);
```

In virtually all other cases, you don't need to call `StripAddress` before using a valid 24-bit address. In particular, you don't need to call `StripAddress` before dereferencing a pointer or handle in 24-bit mode, unless you subsequently switch to 32-bit mode by calling `SwapMMUMode`. Also, you don't need to call `StripAddress` when checking pointers and handles for equality or when performing address arithmetic.

Because you need to call `StripAddress` rarely (if ever), the additional processing time required to call `StripAddress` shouldn't adversely affect the execution of your software. In some cases, however, you might want to avoid the overhead of calling the trap dispatcher every time you need to call `StripAddress`. (A good example might be a time-critical loop in an interrupt task.) You can use the `QuickStrip` function defined in Listing 4-8 in place of `StripAddress` when speed is a real concern.

Listing 4-8 Stripping addresses in time-critical code

```
FUNCTION QuickStrip (thePtr: Ptr): Ptr;
BEGIN
   QuickStrip := Ptr(BAND(LongInt(thePtr), gStripAddressMask));
END;
```

The `QuickStrip` function defined in Listing 4-8 simply masks the address it is passed with the same mask `StripAddress` uses. You can calculate that mask by executing the lines of code in Listing 4-9 early in the execution of your software:

Listing 4-9 Calculating the `StripAddress` mask

```
VAR
   gStripAddressMask:   LongInt;    {global mask variable}

   gStripAddressMask := $FFFFFFFF;
   gStripAddressMask :=
                  LongInt(StripAddress(Ptr(gStripAddressMask)));
```

Unless you are calling `StripAddress` repeatedly at interrupt time, you probably don't need to use this technique.

Translating Memory Addresses

As explained earlier in "Address Translation" on page 4-8, you sometimes need to override the Operating System's standard translation of 24-bit addresses into their 32-bit equivalents. This is necessary because the Virtual Memory Manager might have programmed the MMU to map unused NuBus slot addresses into the address space reserved for RAM. If you try to use a 24-bit address when the system switches to 32-bit mode, the standard translation might result in a 32-bit address that points to the space reserved for expansion cards. In that case, you are virtually guaranteed to obtain invalid results.

To prevent this problem, you can use the `Translate24To32` function to get the 32-bit equivalent of a 24-bit address. In general, you should test for the presence of the `_Translate24To32` trap before you use any 24-bit addresses in 32-bit mode. If it is available, you should use it in place of the static translation process performed automatically by the Operating System while running in 32-bit mode.

Note
You need to use the `Translate24To32` function only when the computer is running in 32-bit mode, it was booted in 24-bit mode, and you are communicating with external hardware. Most applications do not need to use it. ◆

Listing 4-10 illustrates how to use `Translate24To32`. The `DoRoutine` procedure defined there calls the application-defined routine `MyRoutine` to process a block of data while in 32-bit mode. It checks whether the `_Translate24To32` trap is available, and if so, makes sure that the address to be read is a valid 32-bit address.

Listing 4-10 Translating 24-bit to 32-bit addresses

```
PROCEDURE DoRoutine (oldAddr: Ptr; length: LongInt);
BEGIN
   IF TrapAvailable(_Translate24To32) THEN
      MyRoutine(Translate24To32(oldAddr), length);
   ELSE
      MyRoutine(oldAddr, length);
END;
```

Note that you don't need to call `StripAddress` before calling `Translate24To32`, because the `Translate24To32` function automatically ignores the high-order byte of the 24-bit address you pass it. (For a definition of the `TrapAvailable` function, see the chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities*.)

Memory Management Utilities Reference

This section describes the memory management utilities provided by the Operating System.

Routines

This section describes the routines you use to set and restore the A5 register, change the addressing mode, manipulate memory addresses, and manipulate the processor caches.

Setting and Restoring the A5 Register

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

SetCurrentA5

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

```
FUNCTION SetCurrentA5: LongInt;
```

DESCRIPTION

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

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

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

SetA5

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

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

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

DESCRIPTION

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

SEE ALSO

See "The A5 Register" on page 4-5 for a discussion of when you need to call `SetA5`.

Changing the Addressing Mode

If you wish to change address-translation modes manually, you can use the `GetMMUMode` function to find out which mode is currently in use and the `SwapMMUMode` procedure to swap modes.

Note
In general, you need to alter the CPU's addressing mode manually only if you are designing device drivers or other software that communicates with NuBus expansion cards. ◆

GetMMUMode

To find out which address-translation mode (24-bit or 32-bit) is currently in use, use the `GetMMUMode` function.

```
FUNCTION GetMMUMode: SignedByte;
```

DESCRIPTION

The `GetMMUMode` function returns the address-translation mode currently in use. On exit, `GetMMUMode` returns one of the following constants:

```
CONST
   false32b    = 0;              {24-bit addressing mode}
   true32b     = 1;              {32-bit addressing mode}
```

SPECIAL CONSIDERATIONS

To find out which addressing mode was in effect at system startup, use the `Gestalt` function.

ASSEMBLY-LANGUAGE INFORMATION

To determine the current address-translation mode, you can test the contents of the global variable `MMU32Bit`. The value `TRUE` indicates that 32-bit mode is in effect.

SwapMMUMode

To change the address-translation mode from 24-bit to 32- bit or vice versa, use the `SwapMMUMode` procedure.

```
PROCEDURE SwapMMUMode (VAR mode: SignedByte);
```

`mode` On entry, the desired address-translation mode. On exit, the address translation mode previously in use.

DESCRIPTION

The `SwapMMUMode` procedure sets the address-translation mode to the value specified by the `mode` parameter. The mode in use prior to the call is returned in `mode`, and you can restore the previous mode by calling `SwapMMUMode` again. The value of `mode` should be one of the following constants on entry and will be one of the following constants on exit:

```
CONST
   false32b    = 0;            {24-bit addressing mode}
   true32b     = 1;            {32-bit addressing mode}
```

SPECIAL CONSIDERATIONS

You might cause a system crash if you switch to 32-bit addressing mode when your application is executing a code resource you loaded into memory while 24-bit mode was in effect. See "Switching Addressing Modes" on page 4-20 for a description of how this problem arises and how you can avoid it.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `SwapMMUMode` are

Registers on entry

D0 New mode

Registers on exit

D0 Previous mode

Manipulating Memory Addresses

Sometimes you need to modify a memory address before using it. You can strip off a master pointer's flag bits, if any, by calling the `StripAddress` function. You can map 24-bit addresses into the 32-bit address space by calling the `Translate24To32` function.

StripAddress

Use the `StripAddress` function to strip the flag bits from a 24-bit memory address.

```
FUNCTION StripAddress (address: UNIV Ptr): Ptr;
```

`address` The address to strip.

DESCRIPTION

The `StripAddress` function returns a pointer that references the same address passed in the `address` parameter, but in a form that is comprehensible to the 32-bit Memory Manager.

The effect of the `StripAddress` function depends on the startup mode of the Memory Manager, not on the current mode. Thus, if the Memory Manager started up in 32-bit mode, the address passed to `StripAddress` is unchanged (because it already must be a 32-bit address). If the Memory Manager started up in 24-bit mode, the function returns the low-order 3 bytes of the address. You should not pass valid 32-bit addresses to `StripAddress` if the Memory Manager started up in 24-bit mode.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `StripAddress` are

Registers on entry

D0 The address to strip

Registers on exit

D0 The function result

Translate24To32

You can use the `Translate24To32` function to map 24-bit addresses into the 32-bit address space.

```
FUNCTION Translate24To32 (addr24: UNIV Ptr): Ptr;
```

`addr24` An address that is meaningful to the 24-bit Memory Manager.

DESCRIPTION

The `Translate24To32` function translates the address specified by the `addr24` parameter from 24-bit into 32-bit addressing mode and returns that address. If `addr24` is already a 32-bit address, the function returns it unchanged.

Unlike the `StripAddress` function, `Translate24To32` does not necessarily return an address that can be used in 24-bit mode. Also, you cannot meaningfully call `Translate24To32` on the result of a previous translation.

SPECIAL CONSIDERATIONS

You need to call `Translate24To32` only if you use 24-bit addresses while communicating with external hardware in 32-bit mode and virtual memory is enabled. See “Translating Memory Addresses” on page 4-23 for details.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for `Translate24To32` are

Registers on entry

D0 A 24-bit addressing mode address

Registers on exit

D0 The translated address

Manipulating the Processor Caches

The system software provides routines that allow you to enable, disable, and flush the processor caches. Before you call any of the routines described in this section, be sure to check that the trap `_HWPriv` is implemented. The only exception is the `FlushCodeCache` procedure, which is available whenever the processor has a cache that can be flushed.

▲ **WARNING**
If you call these routines and `_HWPriv` isn't implemented, your application will crash. ▲

SwapInstructionCache

You can use the `SwapInstructionCache` function to enable or disable the instruction cache.

```
FUNCTION SwapInstructionCache (cacheEnable: Boolean): Boolean;
```

`cacheEnable`
The desired state of the instruction cache.

DESCRIPTION

The `SwapInstructionCache` function enables or disables the instruction cache, depending on whether the `cacheEnable` parameter is set to `TRUE` or `FALSE`. On exit, `SwapInstructionCache` returns the previous state of the instruction cache.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `SwapInstructionCache` are

Trap macro	Selector
`_HWPriv`	$0000

FlushInstructionCache

You can use the `FlushInstructionCache` procedure to flush the instruction cache.

```
PROCEDURE FlushInstructionCache;
```

DESCRIPTION

The `FlushInstructionCache` procedure flushes the current contents of the instruction cache. Because flushing this cache degrades performance of the CPU, you should call this routine only when absolutely necessary. See "Stale Instructions" on page 4-9 for details on when to call this procedure.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `FlushInstructionCache` are

Trap macro	Selector
`_HWPriv`	$0001

SPECIAL CONSIDERATIONS

On processors with a copy-back data cache, `FlushInstructionCache` also flushes the data cache before it flushes the instruction cache, to ensure that any instructions subsequently copied to the instruction cache are not copied from stale RAM.

SwapDataCache

You can use the `SwapDataCache` function to enable or disable the data cache.

```
FUNCTION SwapDataCache (cacheEnable: Boolean): Boolean;
```

`cacheEnable`
: The desired state of the data cache.

DESCRIPTION

The `SwapDataCache` function enables or disables the data cache, depending on whether the `cacheEnable` parameter is set to `TRUE` or `FALSE`. On exit, `SwapDataCache` returns the previous state of the data cache.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for SwapDataCache are

Trap macro	Selector
_HWPriv	$0002

FlushDataCache

You can use the FlushDataCache procedure to flush the data cache.

```
PROCEDURE FlushDataCache;
```

DESCRIPTION

The FlushDataCache procedure flushes the current contents of the data cache. Because flushing this cache degrades performance of the CPU, you should call this routine only when absolutely necessary. See "Processor Caches" beginning on page 4-8 for details on when to call this procedure.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for FlushDataCache are

Trap macro	Selector
_HWPriv	$0003

FlushCodeCache

You can use the FlushCodeCache procedure to flush the instruction cache.

```
PROCEDURE FlushCodeCache;
```

DESCRIPTION

The FlushCodeCache procedure flushes the current contents of the instruction cache. Because flushing this cache degrades performance of the CPU, you should call this routine only when absolutely necessary. See "Processor Caches" beginning on page 4-8 for details on when to call this procedure.

SPECIAL CONSIDERATIONS

On processors with a copy-back data cache, FlushCodeCache also flushes the data cache before it flushes the instruction cache, to ensure that any instructions subsequently copied to the instruction cache are not copied from stale RAM.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for FlushCodeCache is _CacheFlush.

FlushCodeCacheRange

You can use the FlushCodeCacheRange function to flush a portion of the instruction cache.

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

address	The starting address of the range to flush.
count	The size, in bytes, of the range to flush.

DESCRIPTION

The FlushCodeCacheRange function flushes the current contents of the instruction cache. FlushCodeCacheRange is an optimized version of FlushCodeCache and is intended for use on processors such as the MC68040 that support flushing only a portion of the instruction cache. On processors that do not have this capability, FlushCodeCacheRange simply flushes the entire instruction cache.

The FlushCodeCacheRange function might flush a larger portion of the instruction cache than requested if it would be inefficient to satisfy the request exactly.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for FlushCodeCacheRange are

Trap macro	Selector
_HWPriv	$0009

The registers on entry and exit for `FlushCodeCacheRange` are

Registers on entry

A0	Starting address of the range to flush
A1	Number of bytes to flush
D0	Routine selector

Registers on exit

D0	Result code

RESULT CODES

`noErr`	0	No error
`hwParamErr`	–502	Processor does not support flushing a range

Summary of the Memory Management Utilities

Pascal Summary

Constants

```
CONST
   {Gestalt constants}
   gestaltAddressingModeAttr  = 'addr';   {addressing mode attributes}
   gestalt32BitAddressing     = 0;        {started in 32-bit mode}
   gestalt32BitSysZone        = 1;        {32-bit compatible sys. zone}
   gestalt32BitCapable        = 2;        {machine is 32-bit capable}

   {addressing mode constants}
   false32b                   = 0;        {24-bit addressing mode}
   true32b                    = 1;        {32-bit addressing mode}
```

Routines

Setting and Restoring the A5 Register

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

Changing the Addressing Mode

```
FUNCTION GetMMUMode:        SignedByte;
PROCEDURE SwapMMUMode       (VAR mode: SignedByte);
```

Manipulating Memory Addresses

```
FUNCTION StripAddress       (address: UNIV Ptr): Ptr;
FUNCTION Translate24To32    (addr24: UNIV Ptr): Ptr;
```

Manipulating the Processor Caches

```
FUNCTION SwapInstructionCache(cacheEnable: Boolean): Boolean;
PROCEDURE FlushInstructionCache;
FUNCTION SwapDataCache      (cacheEnable: Boolean): Boolean;
```

```
PROCEDURE FlushDataCache;
PROCEDURE FlushCodeCache;
FUNCTION FlushCodeCacheRange(address: UNIV Ptr; count: LongInt): OSErr;
```

C Summary

Constants

```
/*Gestalt constants*/
#define gestaltAddressingModeAttr   'addr';  /*addressing mode attributes*/
#define gestalt32BitAddressing      0;       /*started in 32-bit mode*/
#define gestalt32BitSysZone         1;       /*32-bit compatible sys. zone*/
#define gestalt32BitCapable         2;       /*machine is 32-bit capable*/

/*addressing mode constants*/
enum  {false32b      =  0};                 /*24-bit addressing mode*/
enum  {true32b       =  1};                 /*32-bit addressing mode*/
```

Routines

Setting and Restoring the A5 Register

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

Changing the Addressing Mode

```
pascal char GetMMUMode       (void);
pascal void SwapMMUMode      (char *mode);
```

Manipulating Memory Addresses

```
pascal Ptr StripAddress      (Ptr address);
pascal Ptr Translate24To32   (Ptr addr24);
```

Manipulating the Processor Caches

```
pascal Boolean SwapInstructionCache
                              (Boolean cacheEnable);
pascal void FlushInstructionCache
                              (void);
```

```
pascal Boolean SwapDataCache(Boolean cacheEnable);
pascal void FlushDataCache   (void);
void FlushCodeCache          (void);
OSErr FlushCodeCacheRange    (void *address, unsigned long count);
```

Assembly-Language Summary

Trap Macros

Trap Macros Requiring Routine Selectors

`_HWPriv`

Selector	Routine
\$0000	`SwapInstructionCache`
\$0001	`FlushInstructionCache`
\$0002	`SwapDataCache`
\$0003	`FlushDataCache`
\$0009	`FlushCodeCacheRange`

Global Variables

`CurrentA5`	long	Address of the boundary between the application global variables and the application parameters of the current application.
`MMU32Bit`	byte	`TRUE` if 32-bit addressing mode is in effect.

Result Codes

`noErr`	0	No error
`hwParamErr`	–502	Processor does not support flushing a range