find it

main

Technote 1104

Interrupt-Safe Routines

By Brian Bechtel and Quinn "The Eskimo"
Apple Computer, Inc.
devsupport@apple.com

CONTENTS

System 7 has a badly defined set of extremely heterogeneous programming environments. In some of these environments, your code can access some system services but not others. The names given to these environments are often overloaded and confusing. This results in a lot of programmer confusion.

This technote attempts to clear up this confusion by assigning each of the execution levels a unique name, describing how and why your code might find itself running at a particular execution level, and outlining the restrictions your code might face when running at that level.

Introduction

There has been much confusion about which toolbox calls move memory and which do not, and which calls can be used at interrupt time and which cannot. This Technote describes toolbox calls which may be used at interrupt time. This is a list of guaranteed-safe calls that can be used any time. Developers will have some assurance that these calls are safe, since the system itself calls them under the assumption that they are safe.

This Technote contains calls which are safe at interrupt time, rather than those not safe at interrupt time. This is for several reasons.

1. Since calls tend toward moving memory as the systemx is revised, lists of calls which move memory should grow. Since the system sometimes needs revision regardless of what is printed in *Inside Macintosh* , the absence of a call from such a list does not guarantee that call does not move memory.

2. A call which had been interrupt-safe can become non-interrupt-safe due to revisions in system software and patches. Incompetent patching, unfortunately all too common, can lead to calls which previously did not move memory changing behavior.

3. Calls described in *Inside Macintosh* which could easily be implemented with in line code (HiWord, BAND, etc.) sometimes appear as glue in a segment other than the calling segment. These calls are documented in *Inside Macintosh* , so they might appear to be traps, and their semantics are so simple that programmers might assume they could not possibly move memory.

I recommend that programmers assume any call absent from this list moves memory. This is to be considered a form of defensive programming, not a definitive pronouncement.

The old *Inside Macintosh* , volume 6, appendix B had a list of routines which may be called at interrupt time. This technote is an updated list of those routines, along with comments as appropriate. Do not rely on the list of interrupt-safe routines in *Inside Macintosh* , volume 6, appendix B.

I include comments regarding patching some routines. Patching is as discouraged as it has always been, but sometimes, patching is the best solution for some problems under System 7.x. When you patch a routine which is interrupt-safe, you should assume the same behavior as the interrupt-safe routines: do not allocate, move, or purge memory (either directly or indirectly) and do not depend on the validity of handles to unlock blocks.

This technote is important for anyone programming the Macintosh, and vital for anyone doing system level programming under System 7.

Execution Levels

Traditional System 7 supports the following execution levels:

- Hardware Interrupt
- Deferred Task
- SystemTask

The native device driver model defines the following levels in addition to those listed above:

- Native Hardware Interrupt
- Secondary Interrupt
- Task

These execution levels are modeled after the Mac OS 8 execution levels and correspond roughly with their traditional counterparts. However, the distinction is important in certain circumstances.

Note:
You can read more about native device driver execution levels in (ftp://ftp.apple.com/devworld/Development_Kits/PCI_Driver_SDK) Designing PCI Cards and Drivers for Power Macintosh Computers, page 67

Interrupts: 68K and PowerPC

Note that this discussion does not discuss the PowerPC native interrupt mechanism. On Power Macintosh computers running System 7, the PowerPC native interrupts are handled by a nanokernel, which routes the interrupt through the 68K emulator. Where this note references 68K-specific concepts, you can safely assume that this behavior is emulated by the low-level PowerPC system software on machines with PowerPC processors.

Note that the execution level is to some extent independent of the processor interrupt mask, i.e. the value stored in the 680x0 SR register. In some cases interrupts can be enabled during an interrupt (i.e. while running a deferred task); in some cases interrupts can be disabled at SystemTask time (i.e. Enqueue disables interrupts to guarantee mutual exclusion).

Level Description

This section describes each of the execution levels in detail.

Hardware Interrupt

What it is

Hardware interrupt-level execution happens as a direct result of a hardware interrupt request. The software executed at hardware interrupt level includes installable interrupt handlers for NuBus and other devices, as well as interrupt handlers supplied by Apple.

How to get there

You get to hardware interrupt level as the direct result of a installing a hardware interrupt handler (i.e. a NuBus handler installed with SIntInstall or by changing the interrupt vector tables in low memory) or by being called by something that is directly invoked by a hardware interrupt handler (i.e. a SCSI Manager 4.3 completion routine). Note that Time Manager tasks and VBLs are also executed at hardware interrupt level.

Environment restrictions

Hardware interrupts are considered "interrupt level" as defined by the toolbox, virtual memory and Open Transport. See the later sections for a discussion of these restrictions.

In addition, you should make every attempt to minimize the amount of time you spend at hardware interrupt level. Hardware interrupt level requires that all interrupts with lower interrupt priority be disabled for the duration of the hardware interrupt handler. The longer you spend in your hardware interrupt handler, the longer the interrupt latency of the computer. Increased interrupt latency may result in a poor user experience - such as sound break up or mouse tracking problems - and worse. If you need to do extended processing at interrupt time, you can call `DTInstall` to schedule a deferred task.

Is paging safe?

Paging is not safe at hardware interrupt level unless the interrupt has been deferred using `DeferUserFn`. Some system interrupt handlers (Device Manager completion routines, VBLs, slot VBLs, Time Manager tasks) automatically defer their operation until VM safe time, but other hardware interrupt handlers must be sure not to cause page faults. If you need to access memory that might page fault, you should defer that operation using `DeferUserFn`.

Deferred Task

What it is

A Deferred Task is a mechanism whereby hardware interrupt level code can schedule a routine to be executed when interrupts have been re-enabled but before the return from the interrupt. Hardware interrupt handlers do this in order to minimize the amount of time spent in the hardware interrupt handler, and thereby minimize system interrupt latency.

How to get there

The most common way to get to deferred task level is to execute `DTInstall` from a hardware interrupt handler. The interrupt handling system executes deferred tasks just before returning from interrupts, but after re-enabling interrupts.

You can also get to Deferred Task level by being called by something that is executing at Deferred Task level. A good example of this are Open Transport notifier functions, which are often called at Deferred Task level.

Environment restrictions

Deferred tasks are considered "interrupt level" as defined by the toolbox and virtual memory. See the later sections for a discussion of these restrictions.

Is paging safe?

By default paging is not safe at deferred task level. If you need to access memory that might page fault, you should defer that operating using `DeferUserFn`. Note that many deferred tasks are scheduled by code that has already done this, so this is not always necessary. For example, Open Transport notifier functions are allowed to take page faults.

Special considerations

Another useful feature of deferred tasks is that they are serialized. The system will not interrupt a deferred task in order to run another deferred task. This makes a really neat mutual exclusion mechanism.

SystemTask

What it is

SystemTask level is the level at which most general application code runs.

The name is derived from an obsolete Macintosh system call, SystemTask. Prior to the introduction of MultiFinder (now known as the Process Manager), applications were required to call SystemTask at regular intervals to allow device drivers time to do things that could not be done at interrupt time.

Note that SystemTask is now obsolete because WaitNextEvent automatically calls it for you.

How to get there

An application's main entry point is called at System Task level. Cooperatively scheduled Thread Manager threads also run at SystemTask time. For other types of code, technote 1033 "Interrupts in Need of (a Good) Time" describes how to get to System Task level from interrupt level.

Environment restrictions

Code running at System Task level is not considered "interrupt level" by anything. You can do virtually anything at System Task level.

Is paging safe?

By default paging is safe at System Task level. The exceptions are when your code is accessing some resource that the system needs to support paging. For example, if you obtain exclusive access to the SCSI bus using `SCSIGet`, you must not cause a page fault even at SystemTask level.

Native Hardware Interrupt

What it is

Native Hardware Interrupt level is virtually identical to normal hardware interrupt level except that it only comes into play on machines the native driver architecture.

Note that native does not imply native interrupt processing. Under System 7, the nanokernel vectors all interrupts through the 68K emulator in order to ensure 68K interrupt priorities and instruction atomicity. Therefore, even native hardware interrupts involve Mixed Mode switches.

How to get there

You get to native hardware interrupt level by installing a hardware interrupt handling using the native Interrupt Manager, or by being called by something that is directly invoked by such a handler.

Environment restrictions

Native hardware interrupts are considered "interrupt level" as defined by the toolbox, virtual memory and Open Transport. See the later sections for a discussion of these restrictions.

As with traditional hardware interrupts, you should make every attempt to minimize the amount of time you spend at native hardware interrupt level. If you need to do extended processing in response to a native hardware interrupt, you should schedule a secondary interrupt using `QueueSecondaryInterruptHandler`.

Is paging safe?

Paging is never safe at native hardware interrupt level.

Secondary Interrupt

What it is

The native driver model provides Secondary Interrupts, which are much like the traditional Deferred Tasks, for native drivers to defer complex processing in order to minimize interrupt latency.

How to get there

You get to Secondary Interrupts level by scheduling a secondary interrupt handler using `QueueSecondaryInterruptHandler`, or by being called by such a handler.

Environment restrictions

Secondary interrupts are considered "interrupt level" as defined by the toolbox, virtual memory and Open Transport. See the later sections for a discussion of these restrictions.

Is paging safe?

Paging is never safe at secondary interrupt level.

Task

What it is

Under System 7, the native driver model defines task level to be any code that's not at native hardware interrupt level and not at secondary interrupt level.

How to get there

The most common source of task level execution is standard SystemTask level execution, i.e. normal application code. However, other execution levels that are traditionally considered to be interrupts levels, such as non-native hardware interrupts and Deferred Tasks, are also considered to be task level. Remember that under System 7, task level is defined by being not native interrupt level or secondary interrupt level.

Environment restrictions

The environment restrictions of task level are defined by the traditional level that's really being executed.

Is paging safe?

The native driver model defines that paging is always safe at task level.

Execution Levels in Other Documentation

Virtual Memory

The virtual memory documentation chapter 3 of *Inside Macintosh: Memory* and Technote ME 09, "Coping with VM and Memory Mappings" says that page faults are not allowed at "interrupt time". This has caused a lot of confusion amongst programmer's who have heard that, for example, Device Manager completion routines are "interrupt time", and hence assume that paging is unsafe in MacTCP completion routines. In the light of the above description, it's easy to clear up that confusion.

As far as virtual memory is concerned, "interrupt time" means any hardware interrupt that hasn't been deferred by VM itself or using `DeferUserFn`. So it is safe to take page faults from Device Manager completion routines, even though other documentation might refer to that execution level as "interrupt time".

It's also important to stress the deferred between Deferred Task Time, and a hardware interrupt that's been deferred using `DeferUserFn`. Deferred Task Time is about re-enabling interrupts to minimize interrupt latency. `DeferUserFn` is about deferring hardware interrupts until VM is safe. One does not imply the other.

Open Transport

The Open Transport documentation caused much confusion by saying that Open Transport could not be called at "interrupt time". What this means is that you can only call Open Transport from System Task time, or Deferred Task level. So you can call Open Transport at execution levels that would normally be considered "interrupt time" (i.e. from a Deferred Task), as long as you don't call it from hardware interrupt level. [Or native hardware interrupt level, or secondary interrupt level.]

Toolbox

Most toolbox routines cannot be called at "interrupt time". Unlike the two cases described above, in this case "interrupt time" refers to any non-SystemTask execution level.

Also note that there are many different reasons why toolbox routines can not be called at interrupt time. Some routines, like the Memory Manager, rely on global data structures that are not interrupt-safe. Other routines might move or purge unlocked handles. Still others, like synchronous calls to the Device Manager, are architecturally inaccessible. Still others, like ReadDateTime, rely on interrupts in order to complete, and hence cannot be called when interrupts are disabled.

What Interrupt Routines Can Do

- Interrupt routines cannot do everything that ordinary routines can do. The following list summarizes the operations that interrupt routines should not perform. An interrupt routine which violates one of these rules may cause a system crash.
- An interrupt routine must not allocate, move, or purge memory. An interrupt routine cannot rely on the state of any unlocked handle. An interrupt routine must not call any toolbox routines that may do so. An interrupt routine must not call any Memory Manager call which clears the low memory global MemErr.
- An interrupt routine cannot call a routine from another code segment unless it sets up the application's A5 world properly. In addition, that segment must already be loaded in memory.
- An interrupt routine cannot access your application global variables unless it sets up the application's A5 world properly. This technique is explained in "Accessing Application Globals in a VBL Task" beginning on page 4-13 of *Inside Macintosh: Memory* .
- An interrupt routine's code and any data accessed during the execution of the routine must be locked into physical memory if virtual memory is in operation.
- You should do as little as possible inside an interrupt routine. Delay doing any costly processing until you are no longer within your interrupt routine, but are in normal application routines instead.

Specific Toolbox Calls

- Interrupt routines cannot do everything that ordinary routines can do. The following list summarizes the operations that interrupt routines should not perform. An interrupt routine which violates one of these rules may cause a system crash.
- An interrupt routine must not allocate, move, or purge memory. An interrupt routine cannot rely on the state of any unlocked handle. An interrupt routine must not call any toolbox routines that may do so. An interrupt routine must not call any Memory Manager call which clears the low memory global MemErr.
- An interrupt routine cannot call a routine from another code segment unless it sets up the application's A5 world properly. In addition, that segment must already be loaded in memory.
- An interrupt routine cannot access your application global variables unless it sets up the application's A5 world properly. This technique is explained in "Accessing Application Globals in a VBL Task" beginning on page 4-13 of *Inside Macintosh: Memory* .
- An interrupt routine's code and any data accessed during the execution of the routine must be locked into physical memory if virtual memory is in operation.
- You should do as little as possible inside an interrupt routine. Delay doing any costly processing until you are no longer within your interrupt routine, but are in normal application routines instead.

Memory Manager

In general, there are very few widely used Memory Manager calls which you can safely call at interrupt time. The most common exceptions are `BlockMove` and `StripAddress`; these two calls may be safely made at interrupt time. At interrupt time, you cannot allocate, move, or purge memory (either directly or indirectly). You should never rely on the validity of handles to unlock blocks.

There are some calls documented in *Inside Macintosh: Memory* which are safe. The entire suite of debugger calls are interrupt-safe. This includes `DebuggerEnter`, `DebuggerExit`, `DebuggerGetMax`, `DebuggerLockMemory`, `DebuggerPoll`, `PageFaultFatal`, and `DebuggerUnlockMemory`. You can call `SwapMMUMode` and Translate24to32 at interrupt time.

`LockMemory`, `UnlockMemory`, `LockMemoryContiguous`, and `UnholdMemory` may be called at interrupt time. `GetPageState` and `GetPhysical` are interrupt-safe. `DeferUserFN` is interrupt safe.

`HoldMemory` may *not* be called at interrupt time. `HoldMemory` may move memory. You can't call `HoldMemory` at interrupt time if you're in the middle of a page fault. That's because `HoldMemory` brings any "on disk" pages in the range being held into memory by touching it and causing a page fault. If you were in the middle of a page fault, called `HoldMemory`, and any of that memory were not already held, you'd cause a fatal double-page fault.

VM disables or defers most "user code" (VBL tasks, Time Manager tasks, I/O completion routines, etc.) that can run at interrupt time during page faults. That protects VM from most code that can cause a double-page fault.

However, device drivers (and socket listeners, etc.) that handle hardware interrupts directly, and thus can execute their code during a page fault, cannot call `HoldMemory` on memory that might be paged out during the handling of those interrupts.

So, if your device driver, socket listener, etc. holds its code, parameter blocks, buffers, etc. at non-interrupt time, it can safely use that memory (the code, parameter blocks, buffers, etc.) to make calls that VM patches (like Read/Write/Control/Status) without VM patch's calls to `HoldMemory` causing a double-page fault.

`LockMemory` is safe to call at interrupt time only if the memory is already held. For Reads and Writes through a device driver, the VM patches have already held the memory, so it is safe to call `LockMemory` on those buffers at interrupt time.

Do not call `StackSpace` at interrupt time. `StackSpace` operates by comparing two low memory globals in the current process low memory globals. At interrupt time you are not guaranteed that you are even in a valid process. `StackSpace` also has the unfortunate property of clearing the low memory global MemErr, which is returned by the memory manager call `MemError()`. There are other Memory Manager calls which clear MemErr and should never be called at interrupt time, such as `SetHandleSize`.

Note:
Unfortunately, there is already a lot of software out there that calls `StackSpace` at interrupt time. As an example, some versions of the .Enet ethernet driver from Apple call `StackSpace` at interrupt time. This means that your application cannot rely on the values returned by `MemError()`, since it can be cleared at arbitrary times behind your back.

Operating System Utilities

`Enqueue` and `Dequeue` are interrupt-safe, and may be used at any time. `FormatRecToString` (formerly `Format2Str`), `StringToExtended` (formerly `FormatX2Str`), and `ExtendedToString` (formerly `FormatStr2X`) are interrupt-safe as well.

Note:
Do not call `ReadLocation` at interrupt time. `ReadLocation` needs to get information from the parameter RAM, using the poorly documented call `ReadXPRAM`. Some models of Macintosh computers communication with parameter RAM via interrupts. If you call `ReadXPRAM`, or any routine which calls `ReadXPRAM`, at interrupt time, the call will hang your system.

Device Manager

Asynchronous `Prime`, `Control` or `Status` driver calls are interrupt-safe and must remain so. It should always be possible to call a driver with a `Prime, Control` or `Status` call asynchronously at interrupt time. It should never be possible to call a driver synchronously at interrupt time.

One confusing point is the use of the `Open` and `Close` calls. These calls are shared with the file system. Any `Open` call to the device manager (even asynchronous calls) can move or allocate memory, and therefore cannot be made at interrupt time. Similarly a `Close` call to the device manager may deallocate memory, and cannot be made at interrupt time. `Open` and `Close` can be called asynchronously if the calls are made to the file manager. The current Open/Close code decides that a call is for the device manager if

1. the file name starts with a period (e.g. '.Sony')
2. the trapword bit is set indicating it is a desk accessory (e.g. OpenDeskAccessory)

From the viewpoint of writing a driver, this means that you cannot move or allocate memory in your driver if your driver is called asynchronously. If your driver is called synchronously, you may move or allocate memory.

From the viewpoint of using a driver, you should always assume that any synchronous call to a driver may move or allocate memory.

Networking

Classic AppleTalk is implemented as a set of device drivers, and hence may be called at interrupt time as long as the calls are made asynchronously. MacTCP is split into two parts. The core TCP, UDP, and ICMP support is implemented as a device driver, and hence may be called at interrupt time as long as the calls are made asynchronously. The Domain Name Resolver (DNR) is implemented as glue that you should avoid calling at interrupt time. The `StrToAddr`, `AddrToName`, `HInfo` and `MXInfo` calls are safe at interrupt time under MacTCP. However, these calls will fail under Open Transport TCP/IP, if the first time they are called is at interrupt time.

Open Transport and Interrupt Routines

Open Transport also defines many support routines that clients can use to deal with the communications environments. These are described in the Utility routine section toward the end of this document.

The Open Transport API is intended to provide high-performance communications services to client applications. In keeping with this goal, most Open Transport functions may not be called at interrupt time. This includes any interrupt routine from an external device, VBL tasks, or Time Manager. Open Transport functions may only be called at primary task time (also called System Task time, or at Deferred Task time (also called Secondary Interrupt level) scheduled by using either the Open Transport functions `OTScheduleDeferredTask` or `OTScheduleInterruptTask` or by using the system _DTInstall trap.

In order to support calling at primary interrupt time, Open Transport would have to be able to turn interrupts on and off to protect critical resources. On PowerPC machines, this requires a costly mixed-mode switch. Open Transport provides the functions `OTCreateDeferredTask`, `OTScheduleDeferredTask`, `OTScheduleInterruptTask` and `OTDestroyDeferredTask` to make it very easy for clients to defer their operations to deferred task time without using confusing parameter blocks. Please use them.

After having said this, many of the Open Transport utilities routines are usable at primary interrupt time. Refer to Appendix F of "Open Transport Client Note" for a list of those functions.

Power Manager

Installing and removing a sleep queue entry (using `SleepQInstall` and `SleepQRemove`) is safe. `BatteryStatus` and `SetWUTime` are interrupt safe.

Note:
On some computers, your sleep queue entry may be called at a time when you are not in a current process. This means that it is unsafe to try and implement any user interaction from a sleep queue entry. For example, the sleep switch on the lid of some Duos and some PowerBooks gets noticed by a patch to the Process Manager when it is in the middle of switching processes. If you call a routine such as `ModalDialog` at this time, the Process Manager thinks that there is no current front process, so it fails to post any events for the dialog. You will hang because your modal dialog filter will never receive any events.

Notification Manager

You may call NMInstall and NMRemove at interrupt time.

Note:
A notification response procedure is called at SystemTask time and hence most stuff is safe, although putting up user interface is tricky, because you are running in the context of the front most process.

Desktop Manager

All the asynchronous calls are safe. `PBDTAddAPPLAsync`, etc. call be called at interrupt time.

File System

Any asynchronous file system call is interrupt-safe. This includes:

- PBCatSearchAsync
- PBCreateFileIDRefAsync
- PBDeleteFileIDRefAsync
- PBExchangeFilesAsync
- PBGetForeignPrivs
- PBGetVolMountInfo
- PBGetVolMountInfoSize
- PBHGetVolParmsAsync
- PBMakeFSSpecAsync
- PBReadAsync
- PBResolveFileIDRefAsync
- PBSetForeignPrivAsync
- PBWriteAsync

Note:
All File System Manager (FSM) calls are interrupt safe. A FSM agent should assume that it is running at interrupt time, and not violate the provisions of this Technote except where noted in the FSM documentation.

Gestalt

You should not call Gestalt at interrupt time unless you know that the Gestalt selector is interrupt-safe. This generally applies only for those Gestalt selectors which you yourself have installed. In *Inside Macintosh: Operating System Utilities* on page 1-31 there is a long description of when it might or might not be safe to call Gestalt. This description may be summarized as follows:

When passed one of the Apple-defined selector codes, the Gestalt function does not move or purge memory and therefore may be called at any time, even at interrupt time. However, selector functions associated with non-Apple selector codes might move or purge memory, and third-party software can alter the Apple-defined selector functions.

In practice, Apple has not consistently conformed to this restriction when creating new Gestalt selectors. Apple-supplied Gestalt selectors may move or purge memory. Therefore, it is safest always to assume that Gestalt could move or purge memory. To repeat: you should not call Gestalt at interrupt time unless you know that the Gestalt selector is interrupt-safe. This generally applies only for those Gestalt selectors which you yourself have installed

Sound Manager

`MACEVersion`, `SndGetSysBeepState`, `SndManagerStatus`, `SndPauseFilePlay`, `SndSetSysBeepState`, and `SndSoundManagerVersion` are all interrupt-safe. `SndDoCommand` is usable during sound channel callbacks to queue new sound buffers.

Note:
`SysBeep` is *not* on the list. `SysBeep` can move or allocate memory. Do not call `SysBeep` at interrupt time.

Process Manager

`GetFrontProcess`, `GetCurrentProcess`, `GetNextProcess`, `SameProcess`, and `WakeUpProcess` are interrupt safe.

Time Manager

`InsTime`, `InsXTime`, `PrimeTime` and `RmvTime` are interrupt-safe.

Process to Process Communications Toolbox

All asynchronous PPCToolbox calls are interrupt-safe.

Deferred Task Manager

Deferred task initialization via DTInstall is interrupt safe. Because the deferred task is executed during a hardware interrupt cycle, it should not allocate, move, or purge memory (either directly or indirectly) and should not depend on the validity of handles to unlock blocks.

Vertical Retrace Manager

`SlotVInstall`, `VRemove`, `SlotVRemove`, `AttachVBL`, `DoVBLTask`, and `GetVBLQHdr` are all interrupt safe.

Libraries

`SetupA5`, `SetupA4`, `SetCurrentA5`, `SetCurrentA4`, etc. are interrupt-safe as long as the implementations do not reside in an unloaded segment. You should check the code generated by your development environment before using such calls at interrupt time.

Anything in `<PLStringFuncs.h>` is safe, as long as the implementations do not reside in an unloaded segment.

Packages

Do not call any routine in a Package at interrupt time. Any routine found in a Package (e.g. StandardFile, International Utilities) is not interrupt-safe, since the package may not be in memory at that time.

Routines Which May Be Called At Interrupt Time

This is a summary list of routines which may be called at interrupt time. Those routines with an asterisk (*) have restrictions on their use; see the main body of this Technote for details.

```
AddrToName*
AttachVBL
BatteryStatus
BlockMove
Close*
PBControlAsync
DebuggerEnter
DebuggerExit
DebuggerGetMax
DebuggerLockMemory
DebuggerPoll
DebuggerUnlockMemory
DeferUserFN
Dequeue
```

```
DoVBLTask
Enqueue
ExtendedToString
Format2Str
FormatRecToString
FormatStr2X
FormatX2Str
GetCurrentProcess
GetFrontProcess
GetNextProcess
GetPageState
GetPhysical
GetVBLQHdr
HInfo*
InsTime
InsXTime
LockMemory*
LockMemoryContiguous
MACEVersion
MXInfo*
NMInstall
NMRemove
OTCreateDeferredTask
OTDestroyDeferredTask
OTScheduleDeferredTask
OTScheduleInterruptTask
Open*
PBCatSearchAsync
PBCloseAsync*
PBCreateFileIDRefAsync
PBDTAddAPPLAsync
PBDeleteFileIDRefAsync
PBExchangeFilesAsync
PBGetForeignPrivs
PBGetVolMountInfo
PBGetVolMountInfoSize
PBHGetVolParmsAsync
PBHOpenAsync*
PBHOpenDFAsync*
PBHOpenDenyAsync*
PBHOpenRFAsync*
PBMakeFSSpecAsync
PBOpenAsync*
PBOpenDFAsync*
PBOpenRFAsync*
PBReadAsync
PBResolveFileIDRefAsync
PBSetForeignPrivAsync
PBWriteAsync
PageFaultFatal
Prime
PrimeTime
RmvTime
SameProcess
SetWUTime
SleepQInstall
SleepQRemove
SlotVInstall
SlotVRemove
SndDoCommand*
SndGetSysBeepState
```

```
SndManagerStatus
SndPauseFilePlay
SndSetSysBeepState
SndSoundManagerVersion
PBStatusAsync
StrToAddr*
StringToExtended
StripAddress
SwapMMUMode
Translate24to32
UnholdMemory
UnlockMemory
VRemove
WakeUpProcess
```

Further References

- *Inside Macintosh: Memory*
- File System Manager SDK (ftp://ftp.apple.com/devworld/Development_Kits/File_System_Manager.sit.hqx)

Acknowledgments

Thanks to Jim Luther, Cameron Esfahani, Matt Mora, Pete Gontier, Jim Murphy, Dave Lyons, and Peter N. Lewis.