Technical Note TN1104
Interrupt-Safe Routines

CONTENTS

The traditional Mac OS has a badly defined set of heterogeneous programming environments. In some of these environments, your code can access some system services but not others. Furthermore, 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.

This Technote is important for anyone programming any Mac OS code that might run at "interrupt time," and vital for anyone doing system-level programming under the traditional Mac OS.

Updated: [Nov 08 1998]

Introduction

There has been much confusion about which Mac OS routines can be used at interrupt time and which cannot. This Technote lists the Mac OS routines which can be used at interrupt time.

This Technote list routines which are safe at interrupt time, rather than those that are unsafe. As the system evolves, more routines are added, and it may become necessary to do more work in existing routines. So routines that just happen to be interrupt safe may become otherwise. Thus, any list of interrupt-unsafe routines will grow over time, and consequently is hard to maintain. A list of routines that are safe is more likely to remain accurate.

DTS recommends that you assume all routines absent from this list are unsafe to call at interrupt time. This is a general defensive programming guideline, not a definitive pronouncement. If you know of a routine that you always considered to be interrupt safe that is not listed here, please let us know. As an example of how your feedback is valuable to us, the first version of this technote failed to mention that `SetCursor` was interrupt safe. This was an obvious omission which has now been corrected, and it's likely that there are others.

A interrupt-safe routine can become unsafe if it is patched inappropriately. When you patch a routine which is interrupt safe, you should assume that your patch is running at interrupt time and avoid doing things that are illegal at interrupt time.

> **Note:**
> DTS still recommends against patching, as it has always has. The above comments reflect the pragmatic attitude that, if you're going to patch, you should do it correctly.

The old *Inside Macintosh* , volume 6, appendix B had a list of routines which can 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.

Back to top

Execution Levels

The traditional Mac OS supports the following execution levels:

- Hardware Interrupt
- Deferred Task
- System Task

In addition, the native device driver model defines the following execution levels:

- Native-Hardware Interrupt
- Secondary Interrupt
- Task
- Software Interrupt

Since these execution levels are modeled after the execution levels supported by Copland, their implementation on the traditional Mac OS is somewhat imprecise. In broad terms, the following analogies apply:

- native-hardware interrupt *is like* hardware interrupt
- secondary interrupt *is like* deferred task
- task *is like* system task
- software interrupt *is* not supported

However, the distinction between these analogous pairs is important in certain circumstances, as explained later in this note.

> **Note:**
> You can read more about native device driver execution levels in Designing PCI Cards and Drivers for Power Macintosh Computers, page 67.

> **IMPORTANT:**
> This Technote does not discuss the PowerPC hardware interrupt mechanism. On PowerPC computers running the traditional Mac OS, PowerPC hardware 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.

> **IMPORTANT:**
> The execution level is largely independent of the processor interrupt mask, i.e., the value stored in the 680x0 SR register. The interrupt mask is not a reliable way to detect whether you are at "interrupt time." See Determining the Execution Level for details.

This remainder of this section describes each of the execution levels in detail.

Hardware Interrupt

What is it?

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

How do you get there?

You get to hardware interrupt level as the direct result of a installing a hardware interrupt handler (e.g., 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 (e.g., a SCSI Manager 4.3 completion routine). Note that Time Manager tasks and VBLs are also executed at hardware interrupt level.

What can you do there?

Hardware interrupts are considered "interrupt time" as defined by the toolbox, Virtual Memory Manager, and Open Transport. The associated restrictions are described later in this document.

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 will be. Increased interrupt latency may result in a poor user experience -- such as sound breakup or mouse tracking problems -- or worse. If you need to do extended processing at interrupt time, you should schedule a deferred task (using `DTInstall`) to perform the operation.

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`.

> **Note:**
> Do not confuse the semantics of `DeferUserFn`, which defers a hardware interrupt until paging is safe, with those of `DTInstall`, which schedules a deferred task to be executed when interrupts are re-enabled.

Deferred Task

What is it?

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 do you get there?

The most common way to get to deferred task level is to have your hardware interrupt handler call `DTInstall` to schedule a routine, which the system calls back at deferred task time. The interrupt 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.

What can you do there?

Deferred tasks are considered "interrupt time" as defined by the toolbox. The associated restrictions are described later in this document.

Is paging safe?

Paging is safe at deferred task level.

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.

System Task

What is it?

System task level is the level at which most application code runs.

The name is derived from an obsolete Mac OS 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:**
> The `SystemTask` routine itself is now obsolete because `WaitNextEvent` automatically calls it for you. However, the name lives on as a testament to those hardy Mac OS pioneers who actually had to call it.

How do you get there?

An application's main entry point is called at system task level. Cooperatively scheduled Thread Manager threads also run at system task level. 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.

What can you do there?

Code running at system task level is not considered "interrupt time" 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 occur 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 system task level.

Native Hardware Interrupt

What is it?

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

> **Note:**
> The native in the name of this level does not imply fully native-interrupt processing. Under the traditional Mac OS, 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 Manager switches.

How do you get there?

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

What can you do there?

Native hardware interrupts are considered "interrupt time" as defined by the toolbox, Virtual Memory Manager and Open Transport. The associated restrictions are described later in this document.

As with code running at hardware interrupt level, 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`) to continue the interrupt processing.

Is paging safe?

Paging is not safe at native hardware interrupt level.

Secondary Interrupt

What is it?

The native driver model provides secondary interrupts -- which are much like deferred tasks -- allowing native drivers to defer complex processing in order to minimize interrupt latency.

How do you get there?

You can get to secondary interrupt level by having your native hardware interrupt handler call `QueueSecondaryInterruptHandler` to schedule a routine which the system calls back at secondary interrupt level. The interrupt system executes secondary interrupts after re-enabling interrupts but before running deferred tasks and returning from the interrupt handler.

You can also execute a secondary interrupt handler directly from task level using `CallSecondaryInterruptHandler2`.

What can you do there?

Secondary interrupts are considered "interrupt time" as defined by the toolbox, Virtual Memory Manager and Open Transport. The associated restrictions are described later in this document.

Is paging safe?

Paging is not safe at secondary interrupt level.

Task

What is it?

Under the traditional Mac OS, 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 do you get there?

The most common source of task level execution is standard system task level execution, i.e., normal application code. However, other execution levels that are traditionally considered to be interrupt levels, such as non-native hardware interrupts and deferred tasks, are also considered to be task level. Remember that under the traditional Mac OS, task level is defined as either non- native interrupt level or secondary interrupt level.

What can you do there?

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

Is paging safe?

The native driver model defines that paging is always safe at task level. However, on the traditional Mac OS, paging is only safe at task level if the underlying execution level defines it to be safe.

Software Interrupt

What is it?

The native driver model defines the concept of a software interrupt, the ability to force a task to immediately execute a routine in the context of that task. This is distinct from, but commonly confused with, secondary interrupt level.

How do you get there?

Software interrupts are not supported under Mac OS. This is clearly stated in Designing PCI Cards and Drivers for Power Macintosh Computers, page 262:

> Currently, `SendSoftwareInterrupt` calls the user back at the same execution level. In future versions of Mac OS it can be used to force execution of code that can't be called at interrupt time.

This means is that if you call `SendSoftwareInterrupt` at execution level X, the software interrupt will run at execution level X. This makes software interrupts effectively useless on the traditional Mac OS.

What can you do there?

Software interrupts are defined to run at task level, in the context of the task to which the software interrupt was sent.

Is paging safe?

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

Note:
When the native driver model was designed, it was designed with Copland in mind. The goal was that a native driver (binary, not source) would run without modification on both the traditional Mac OS and Copland. A lot of effort was put into both operating systems to support this goal.

In general, the support for the native driver model on the traditional Mac OS is acceptable. However, in some cases, it is just not possible to support features of Copland under the traditional Mac OS. The most obvious of these is software interrupts. These require significant microkernel support and were not implemented on the traditional Mac OS.

Given that Copland is dead, software interrupts linger on in name only, the vestigial appendix on the intestine that is the native driver model.

Back to top

Execution Levels in Other Documentation

In general, the following execution levels are considered to be "interrupt time."

- Hardware Interrupt
- Deferred Task
- Native Hardware Interrupt
- Secondary Interrupt

However, the use of the term "interrupt time" can vary from manager to manager. This section documents some of the more confusing cases.

Toolbox

Most toolbox routines cannot be called at "interrupt time," as it is defined above.

There are many different reasons why toolbox routines cannot be called at interrupt time. Some routines, like all of the Memory Manager, rely on global data structures that are not interrupt safe. Other routines might move or purge unlocked handles, which is equivalent to calling the Memory Manager. Still others, like synchronous calls to the File Manager, are architecturally inaccessible. Finally, some routines, like `ReadDateTime`, rely on interrupts in order to complete, and hence cannot be called when interrupts are disabled.

The fact that a routine doesn't move or purge memory does not mean it is interrupt safe.

Virtual Memory Manager

The Virtual Memory Manager 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 among programmers 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 the 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."

For the full story about virtual memory on the traditional Mac OS, check out Technote 1094: "Virtual Memory Application Compatibility".

Open Transport

The original Open Transport documentation caused much confusion by saying that Open Transport could not be called at "interrupt time." This means that you can only call Open Transport from system task level or deferred task level. So you can call Open Transport at execution levels that would normally be considered "interrupt time" (specifically, from a deferred task) as long as you don't call it from hardware interrupt level (or native hardware or secondary interrupt levels).

This confusion has been cleared up in the latest release of Inside Macintosh: Networking with Open Transport, which has an extensive table of which Open Transport routines can be called from which execution levels.

Back to top

Determining the Execution Level

There is no good general purpose way to determine the current execution level. In general, your code must know in advance the level at which it is executing. However, there are a number of mechanisms that work for specific environments. The following sections describe those mechanisms and their flaws.

Tracking Interrupts Yourself

One solution for determining the current execution level is to track interrupts yourself. The code in Listing 1 outlines how you can do this.

Listing 1. Tracking the interrupt level yourself.

```
static SInt32 gMyInterruptDepth = 0;

static void MyIOCompletion(ParmBlkPtr pb)
{
    (void) AddAtomic(1, &gMyInterruptDepth);

    ... your code goes here ...

    (void) AddAtomic(-1, &gMyInterruptDepth);
}

static OSStatus MyCommonCode(void)
    // This code may be called at interrupt or non-interrupt time.
{
    if (gMyInterruptDepth == 0) {
        ... run interrupt-safe code ...
    } else {
        ... run system task code ...
    }
}
```

IMPORTANT:
The code in Listing 1 code adjusts `gMyInterruptCountDepth` using an atomic operation. This ensures that the operation yields the correct results even if it's interrupted by another nested interrupt.

This technique works well if you know the execution level of all entry points to your code. However, it fails when your code can be called at unknown execution levels. For example, a disk driver's Prime entry point can be called at a variety of execution levels.

Testing the Interrupt Mask

A common mistake made by developers is to assume that a non-zero 68K interrupt mask (bits 8 through 10 of the 680x0 SR register) indicates that the processor is running at 'interrupt time'. This is an incorrect assumption. It provides both false negatives and false positives.

- In some cases, such as when running a deferred task or a secondary interrupt, the interrupt mask is 0 while inside an interrupt context.
- In other cases the interrupt mask is non-zero at system task time. Numerous older parts of the system, for example the `Enqueue` routine, disable interrupts to guarantee mutual exclusion.

Open Transport

Open Transport provides three routines, `OTIsAtInterruptLevel`, `OTCanLoadLibraries`, and `OTCanMakeSyncCall`, that you might think useful in determining the current execution level. However, these routines only work correctly within contexts in which it is legal to call Open Transport. Specifically, you can call these routines and get meaningful results in the following situations.

- from system task level
- from a deferred task
- from an OT notifier (these run at deferred task time, so this is simply a specialization of the previous point)
- from arbitrary 'interrupt time', if you have follow the OT rule that all interrupt routines (other than deferred tasks) must call `OTEnterInterrupt` on entry and `OTLeaveInterrupt` on exit

As an example of where these routines go wrong, if you call `OTCanMakeSyncCall` from secondary interrupt level on older systems you will find that it returns true!

The upshot of this is that these Open Transport routines are helpful for OT programmers, but do not solve the problem in general.

CurrentExecutionLevel

DriverServicesLib, introduced with the first PCI Power Macintosh computers, exports a routine called `CurrentExecutionLevel` that purports to return the execution level of the currently running code. `CurrentExecutionLevel` is accurate within the contexts supported by the native driver model, but it is not useful outside of those contexts.

As originally implemented, `CurrentExecutionLevel` used the algorithm shown in Listing 2.

Listing 2. The original `CurrentExecutionLevel` algorithm.

```
if we're in the context of a native hardware interrupt then
  return kHardwareInterruptLevel
else if we're in the context of a secondary interrupt then
  return kSecondaryInterruptLevel
else
  return kTaskLevel
end if
```

This algorithm works fine within the execution levels supported by DriverServicesLib, but it does not account for execution levels outside of the native driver model. For example, if you call this version of `CurrentExecutionLevel` from a deferred task or a Time Manager task the result is `kTaskLevel` (unless you happen to have interrupted some native driver processing). For some background on why this routine doesn't work properly, see DTS Q&A DV 43 InterfaceLib and Native Drivers.

In recent systems `CurrentExecutionLevel` has changed [2323165] to more accurately reflect the execution level of non-DriverServicesLib environments. Modern systems maintain a count of the number of times an interrupt (any interrupt, including native hardware interrupts and other, older, interrupt sources such as the Time Manager) has been dispatched. The `CurrentExecutionLevel` algorithm was modified to use that count, as shown in Listing 3.

Listing 3. The updated `CurrentExecutionLevel` algorithm.

```
if we're in the context of a native hardware interrupt then
  return kHardwareInterruptLevel
else if we're in the context of a secondary interrupt then
  return kSecondaryInterruptLevel
else
  if interrupt depth is zero then
    return kTaskLevel
  else
    return kHardwareInterruptLevel
  end if
end if
```

Therefore `CurrentExecutionLevel` is an accurate indication of the current execution level only if uses the new algorithm. Given that numerous machines still use the old algorithm (for example, all non-ROM-in-RAM computers), `CurrentExecutionLevel` is not a general purpose solution to this problem.

Note:
This raises the question, how do you detect which version of `CurrentExecutionLevel` is implemented? The code for doing this is available for download at the end of this technote. The code is complicated by the fact that the first systems to include the new `CurrentExecutionLevel` algorithm shipped as Mac OS version 9.0.4; the change was in the Mac OS ROM file (the NewWorld ROM) not in the System file itself.

TaskLevel

Mac OS 9.0 introduced a new system routine, `TaskLevel`, that was intended to assist in debugging. To emphasize this, `TaskLevel` is declared in "Debugging.h" and exported from DebugLib. The prototype and result mask constants are shown in Listing 4.

Listing 4. `TaskLevel` prototype and constants.

```
enum {
    k68kInterruptLevelMask      = 0x00000007,
    kInVBLTaskMask              = 0x00000010,
    kInDeferredTaskMask         = 0x00000020,
    kInSecondaryIntHandlerMask  = 0x00000040
};

extern pascal UInt32 TaskLevel(void);
```

Warning:
The result from `TaskLevel` is only an approximate answer to the question of what is the current execution level. It is helpful for debugging, but you should not use it for making non-debugging run-time decisions (such as whether to read a file synchronously or asynchronously).

The idea behind `TaskLevel` is that you can use it to add debugging code to your software to detect when it has been called inappropriately. For example, if you're writing a driver and certain sections allocate memory from the system heap, and thus only be called at system task level, you can use `TaskLevel` to detect if you've accidentally called that code at interrupt time. An example of this is shown in Listing 5.

Listing 5. An example of using `TaskLevel` for debugging.

```
static DrvQElPtr MyCreateDriveQueueElement(DeviceIdent id)
{
    DrvQElPtr result;

    assert(TaskLevel() == 0);

    result = (DrvQElPtr) NewPtrSysClear(sizeof(MyDrvQEl));
    if (result != nil) {
        ... fill out fields ...
    }

    return result;
}
```

Back to top

What Interrupt Routines Can't Do

Code running at "interrupt time" cannot do everything that system task code can do. The following list summarizes the operations that interrupt routines should not perform. An interrupt routine which violates any of these rules may cause a system crash:

- An interrupt routine must not allocate, move, or purge memory using the Mac OS Memory Manager.
- An interrupt routine cannot rely on the state of any unlocked handle.
- An interrupt routine must not call any Memory Manager routine which sets the low memory global `MemErr`.
- An interrupt routine must not call any Mac OS routines that violate the above.
- An interrupt routine must not do synchronous I/O. This includes File Manager, Device Manager, PPC Toolbox, and Open Transport I/O.
- For 68K code, an interrupt routine cannot access application global variables unless it sets up the application's A5 world properly. This technique is explained in the Accessing Application Globals in a VBL Task section of Inside Macintosh: Memory.
- For 68K code, an interrupt routine cannot call a routine from another code segment unless the segment is loaded in memory and linked into the code's jump table. In addition, the code must established the correct A5 world before calling across segments at interrupt time.
- As a special case of the above, some of the routines described in *Inside Macintosh* (for example, `BitAnd`, `HiWord`) are actually implemented as glue that is statically linked to your program. It's important to remember that this glue may be in another segment and, even though the routine itself does not move memory, the act of calling it might.
- CFM-68K code must comply with the requirements outlined in Technote 1084: "Running CFM-68K Code at Interrupt Time: Is Your Code at Risk?"

Back to top

Interrupt-Safe Routines by Manager

This section describes various interrupt-safe routines, grouped by manager.

IMPORTANT:
This list is intended only to document those routines which should always be safe to call at interrupt time. There may be other routines, not documented here, which are safe by virtue of their current implementation. You should not rely on such routines continuing to be interrupt safe.

Memory Manager

There are very few Memory Manager routines that you can safely call at interrupt time. The most common exceptions are `BlockMove` (including `BlockMoveData` and other variants) and `StripAddress`; these two routines may be safely made at all execution levels. 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 routines documented in Inside Macintosh: Memory that are safe. The entire suite of debugger routines are interrupt safe. This includes DebuggerEnter, DebuggerExit, DebuggerGetMax, DebuggerLockMemory, DebuggerPoll, PageFaultFatal, DebuggerUnlockMemory, SwapMMUMode, and Translate24to32.

The Virtual Memory Manager routines GetPageState, GetPhysical, DeferUserFN, UnholdMemory, and UnlockMemory are interrupt safe.

The Virtual Memory Manager routines HoldMemory, LockMemory, LockMemoryContiguous, and LockMemoryForOutput are interrupt safe if you guarantee that either page faults are allowed or, if paging is unsafe, that the routines will not cause a page fault. For example, it's safe to call LockMemory on memory that you can guarantee is held.

No other Memory Manager routines are interrupt safe, for one or more of the following reasons:

1. They clear the low-memory global MemErr, which is returned by the Memory Manager call MemError. Applications regularly use MemError to examine the result of the previous Memory Manager operation and may not properly detect a memory error if MemErr changes at interrupt time.
2. They allocate, move, or purge memory, or rely on the state of unlocked handles.
3. They examine data structures that can be in an inconsistent state at interrupt time.

IMPORTANT:
Developers sometimes think "Calling a routine that doesn't move memory (like DisposeHandle) should be safe as long as I save and restore the value of MemErr." **This is not true** because of point 3 above.

Specifically, 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 clears the low memory global MemErr, which is returned by the Memory Manager call MemError. Applications regularly uses MemError to examine the result of the previous Memory Manager operation, and may not properly detect a memory error if MemErr changes at interrupt time.

Note:
Unfortunately, there is some shipping software that calls StackSpace at interrupt time. Even more unfortunately, Apple has -- in the past -- shipped software that calls StackSpace at interrupt time.

Apple is committed to eliminating bugs like this from its system software, and DTS recommends that developers continue to rely on the results of MemError. However, the paranoid developer may wish to implement a wrapper for common Memory Manager routines, as shown below:

```
static OSErr MyNewHandle(Size byteCount, Handle *result)
{
    OSErr err;

    Assert(result != nil);
    err = noErr;
    *result = NewHandle(byteCount);
    if (*result == nil) {
        err = MemError();
        Assert(err != noErr);
        if (err == noErr) {
            err = memFullErr;
        }
    }
    return err;
}
```

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 (PRAM), using the poorly documented ReadXPRAM routine. Some Mac OS computers communicate with parameter RAM via interrupts. If you call ReadXPRAM, or any routine which calls ReadXPRAM, at interrupt time, the call may hang your system.

Device Manager

The core Device Manager traps (_Open, _Read, _Write, _Control, _Status, _Close) are interrupt safe in some cases. Some of these traps (_Open, _Read, _Write, _Close) are shared with the File Manager and the behavior is slightly different for Device Manager requests versus File Manager requests. The following rules summarize the situation:

- Synchronous routines are never interrupt safe.
- Asynchronous routines are interrupt safe, if they are legal at all.
- Immediate routines are interrupt safe if the receiving driver is prepared to handle immediate requests at interrupt time. Immediate routines are never legal for files.
- You should always open and close device drivers with `OpenDriver` and `CloseDriver`, which must be called at system task time.
- You should always open a file with one of the "OpenDF" routines (`FSpOpenDF`, `PBOpenDF`, `PBHOpenDF`). Asynchronous variants of these routines are interrupt safe.
- Asynchronous variants of the other "Open" routines (`PBOpen`, `PBHOpen`) are interrupt safe when applied to files. However, you should avoid these routines because they might unexpectedly open a device driver. For example, if you attempt to open a file called ".Sony", these routines might open the floppy device driver rather than the file.

The next section gives details on File Manager routines that are not shared with Device Manager.

If you're patching the Device Manager traps described above, you must ensure that your patch correctly handles interrupt-time requests. Your patch should not do interrupt-unsafe things unless it determines that the request is synchronous.

When implementing a device driver, you receive three types of requests: synchronous, asynchronous, and immediate. **If the driver can be called asynchronously, you must implement both synchronous and asynchronous requests as if they were asynchronous**, and not do things that are illegal at interrupt time. [This point is discussed in great detail in Technote 1067: "Traditional Device Drivers: Sync or Swim".] On the other hand, immediate requests always execute at the execution level at which the request was made, so if you know that your client made the request at system task time, you know you are running at system task time.

As a special case of this last point, a driver is always sent `accRun` control routines as an immediate request at system task time, so your driver can move or purge memory in response to an `accRun` call.

File Manager

All asynchronous File Manager routines are interrupt safe. For example, `PBOpenDFAsync` can be called at interrupt time.

File System Manager

The File System Manager service routines `GetFSInfo` and `SetFSInfo` are interrupt safe. Other File System Manager service routines (`InstallFS`, `RemoveFS`, `InformFSM`, `InformFFS`) are documented as not being interrupt safe.

A File System Manager plug-in should assume that it is running at interrupt time, and not violate the provisions of this Technote except where noted in the File System Manager documentation. As a consequence, most File System Manager utility routines must be interrupt safe. The routines documented not to be interrupt safe are `UTAllocateVCB` and `UTDisposeVCB`. Other File System Manager utility routines (for example, `UTCacheReadIP`) are interrupt safe but have other documented environmental restrictions.

Driver Services

The native driver support library (`DriverServicesLib`) provides a large number of routines that are "interrupt-safe." The execution level at which these routines may be called is defined in Designing PCI Cards and Drivers for Power Macintosh Computers, Table 9-2, starting on page 283.

When reading this table, you should note a number of important caveats:

- The column labelled "Software interrupt level" should be labelled "Secondary interrupt level."
- To work in the context of this technote, the column labelled "Hardware interrupt level" should be labelled "Native Hardware Interrupt Level."
- Routines that are labelled as allocating memory must be called at task level, and the underlying execution level must be system task level.
- Routines callable from native hardware interrupt level are also callable from hardware interrupt level.

In addition, the valid execution levels for `PrepareMemoryForIO` is covered in DTS Q&A DV 32: "PrepareMemoryForIO and Execution Levels."

Classic 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.

On the other hand, the Domain Name Resolver (DNR) is implemented as glue in your application. The `StrToAddr`, `AddrToName`, `HInfo`, and `MXInfo` routines are safe at interrupt time under MacTCP. However, these routines will fail (returning an error code) under Open Transport TCP/IP if they are first called at interrupt time. For this reason, DTS recommends that you do not calls these routines at interrupt time.

Open Transport

The latest release of Inside Macintosh: Networking with Open Transport has an extensive table of which Open Transport routines can be called from which execution levels.

Power Manager

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

> **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 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 system task time and hence it is safe to call most Toolbox routines. However, putting up user interface is tricky because you are running in the context of the front-most process.

Desktop Manager

All asynchronous Desktop Manager routines are interrupt safe. For example, the `PBDTAddAPPLAsync` routine can be called at interrupt time.

Gestalt

Inside Macintosh: Operating System Utilities has this to say about calling `Gestalt` at interrupt time:

> 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.

This statement is mostly correct. However, there are two important caveats:

1. Not all Apple-defined `Gestalt` selectors are interrupt safe, and there is no hard-and-fast rules for determining which are and which aren't.
2. Prior to Mac OS 8.5, the Gestalt Manager itself has a small concurrency hole (when it grows the `Gestalt` table) during which it may return incorrect information. In theory this makes `Gestalt` unsafe to call at interrupt time; in practice, the `Gestalt` table grows very rarely and Apple has not yet seen a case where this has caused problems.

In summary, our advice is that you should:

- avoid using `Gestalt` at interrupt time in new code,
- attempt to remove any interrupt-time usage of `Gestalt`, as convenient, when revising old code.

We do not believe that it is necessary for you to revise your code just to address this issue.

Sound Manager

`MACEVersion`, `SndGetSysBeepState`, `SndManagerStatus`, `SndPauseFilePlay`, `SndSetSysBeepState`, and `SndSoundManagerVersion` are all interrupt safe.

`SndDoImmediate` and `SndDoCommand` are interrupt safe if the command issued is interrupt safe. Specifically, a `bufferCmd` is not interrupt safe if it requires that the sound output channel be reconfigured. The sound output channel is reconfigured if the format of the sound changes from one buffer to the next (i.e., the sound changed from mono to stereo [or the reverse], 8-bit to 16-bit [or the reverse], or its compression format changed).

It is not safe (with one exception) to start playing a sound at interrupt time, but it is safe to continue playing a sound at interrupt time. The exception is that you can start playing a sound at interrupt time, if you have previously issued a `soundCmd` at task level on the same sound channel to allow the Sound Manager to prepare the sound channel for the type of sound that you will be playing at interrupt time.

> **IMPORTANT:**
> `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.

Multiprocessing Services

The existing documentation (Adding Multitasking to Applications Using Multiprocessing Services, version 2.1) mentions that the following routines are interrupt safe: `MPCurrentTaskID`, `MPYield`, `UpTime`, `MPSignalSemaphore`, `MPSetEvent`, and `MPNotifyQueue`.

IMPORTANT:
MPNotifyQueue is only interrupt safe if you have reserved space on the queue using MPSetQueueReserve.

In addition, the routines MPTaskIsPreemptive, MPBlockCopy, MPBlockClear, and MPDataToCode are interrupt safe, even though the existing documentation does not say that they are. This change will be rolled into a future version of the documentation. [2456896]

Time Manager

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

Process to Process Communications Toolbox

All asynchronous PPC Toolbox routines are interrupt safe.

Communications Toolbox

The Connection Manager routines CMRead, CMWrite, and CMStatus are interrupt safe; all other Connection Manager, Terminal Manager, File Transfer Manager, Communications Resource Manager, and Communications Toolbox Utilities routines are not.

Deferred Task Manager

Deferred task installation via DTInstall is interrupt safe. A deferred task runs at interrupt time with respect to most of the Mac OS toolbox and should follow the rules for interrupt time code.

Vertical Retrace Manager

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

Libraries

SetupA5, SetupA4, SetCurrentA5, SetCurrentA4, and so on 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 routines 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 implemented in a package (List Manager, Disk Initialization, Standard File, SANE, International Utilities, Apple Event Manager, PPC Browser, Edition Manager, Color Picker, Database Access Manager, Help Manager, and the Picture Utilities) at interrupt time. Package routines are not interrupt safe, since the package may not be in memory at that time.

Component Manager

Opening and closing a component is not safe to do at interrupt time, but many other component routines are interrupt safe. You should check the specifics of the component in question to determine exactly which functions can be called at interrupt time.

Event Manager

The only interrupt-safe Event Manager routines are PostEvent and PPostEvent. Other routines, specifically OSEventAvail, TickCount and GetKeys, are not interrupt safe.

IMPORTANT:
TickCount and GetKeys are not interrupt safe. This is because they support the Journaling Mechanism, as described in *Inside Macintosh I* , page 261. While the Journaling Mechanism is long obsolete -- leaving the core implementation of these routines interrupt safe -- it is legal for third party extensions to patch these routines with non-interrupt safe patches.

OSEventAvail has not been interrupt safe since System 7.0 because of a Help Manager patch. This could be considered a bug in Help Manager, however, the long standing nature of this bug means that it will not be fixed.

If you are writing interrupt-time code, you should use the alternatives shown in the Table 1.

Table 1. Interrupt-safe alternatives to TickCount and GetKeys.

Routine	Traditional Mac OS	Carbon
TickCount	LMGetTicks	TickCount [1]
GetKeys (modifiers only)	KeyMap ($174) [3]	GetCurrentKeyModifiers
GetKeys (other keys)	KeyMap ($174) [3]	none/GetKeys [2]

Notes:

1. The Carbon implementation of TickCount on traditional Mac OS calls LMGetTicks and is therefore interrupt safe.
2. GetKeys is interrupt-safe on traditional Mac OS when using CarbonLib 1.1 or later. It is always interrupt-safe on Mac OS X.
3. KeyMap is a low-memory global (at location $174) which contains the data returned by GetKeys.

QuickDraw

Virtually none of QuickDraw is interrupt safe. The exception is `SetCursor`, which is documented as interrupt safe. If you patch `SetCursor`, you should be sure that your patch is interrupt safe because it can and will be called at interrupt time.

> **IMPORTANT:**
> `SetCCursor` is not interrupt safe and never will be. `SetCCursor` is not interrupt safe because, amongst other things, the `CCrsr` data structure contains unlocked handles. Apple cannot just define it to be interrupt safe, because on real world systems `SetCCursor` is patched by interrupt-unsafe third party extensions.
>
> Apple is aware of the demand for an interrupt-safe mechanism for setting color cursors and is working on an alternate mechanism.

Do not be tricked into thinking that trivial QuickDraw routines -- such as `SetRect` or `Random` -- are interrupt safe: they are not! This is partly by definition and partly because it's possible for these routines to reside in pageable code fragments. If you call these routines at any time paging is unsafe, they could cause a fatal page fault.

Text Utilities

`EqualString` and `RelString` are interrupt safe, along with any other routines based on the `_CmpString` ($A03C) and `_RelString` ($A050) traps. These routines must be interrupt safe because they are used by parts of Mac OS (for example, File Manager and classic AppleTalk) that execute at interrupt time.

> **IMPORTANT:**
> These routines are not suitable for comparing user-visible text because they do not make use of any script or language information. To compare user-visible text, you should use one of the other (non-interrupt safe) routines declared in "StringOrder.h" (for example, `IdenticalString`, `CompareString`, and `StringOrder`).

> **Note:**
> `EqualString` and `RelString` *are* suitable for comparing the following system entities.
>
> - file names, as returned by the File Manager in Pascal string format (but only if you want the same order as used internally by HFS)
> - resource names
> - AppleTalk NBP entity names, types and zones

Unicode Converter

It is possible, with some restrictions, to call the Unicode Converter at interrupt time. If you have a specific product that needs this ability, please contact DTS for details.

Back to top

Summary of Interrupt-Safe Routines

Table 1 contains a listing 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:

Table 1. Listing of Interrupt-Safe Routines.

```
AddrToName *
AttachVBL
BatteryStatus
BlockMove
PBControlAsync
CMRead
CMStatus
CMWrite
DebuggerEnter
DebuggerExit
DebuggerGetMax
DebuggerLockMemory
DebuggerPoll
DebuggerUnlockMemory
DeferUserFN
Dequeue
DoVBLTask
Enqueue
EqualString
ExtendedToString
Format2Str
FormatRecToString
FormatStr2X
FormatX2Str
GetCurrentProcess
GetFrontProcess
GetFSInfo
GetNextProcess
GetPageState
GetPhysical
GetVBLQHdr
HInfo *
HoldMemory *
InsTime
InsXTime
LockMemory *
LockMemoryContiguous *
LockMemoryForOutput *
MACEVersion
MPBlockClear
MPBlockCopy
MPCurrentTaskID
MPDataToCode
MPNotifyQueue *
MPSetEvent
MPSignalSemaphore
MPTaskIsPreemptive
MPYield
MXInfo *
NMInstall
NMRemove
         Open Transport routines *
PBAllocContigAsync
PBAllocateAsync
PBCatMoveAsync
PBCatSearchAsync
PBCloseAsync *
PBCloseWDAsync
PBControlAsync
PBControlImmed *
PBCreateAsync
PBCreateFileIDRefAsync
PBDTAddAPPLAsync
PBDTAddIconAsync
PBDTDeleteAsync
PBDTFlushAsync
PBDTGetAPPLAsync
PBDTGetCommentAsync
PBDTGetIconAsync
PBDTGetIconInfoAsync
PBDTGetInfoAsync
PBDTRemoveAPPLAsync
PBDTRemoveCommentAsync
PBDTResetAsync
PBDTSetCommentAsync
PBDeleteAsync
PBDeleteFileIDRefAsync
PBDirCreateAsync
```

```
PBHOpenRFDenyAsync
PBHRenameAsync
PBHRstFLockAsync
PBHSetDirAccessAsync
PBHSetFInfoAsync
PBHSetFLockAsync
PBHSetVolAsync
PBLockRangeAsync
PBMakeFSSpecAsync
PBOpenAsync *
PBOpenDFAsync
PBOpenRFAsync
PBOpenWDAsync
PBReadAsync
PBReadImmed *
PBRenameAsync
PBResolveFileIDRefAsync
PBRstFLockAsync
PBSetAltAccessAsync
PBSetCatInfoAsync
PBSetEOFAsync
PBSetFInfoAsync
PBSetFLockAsync
PBSetFPosAsync
PBSetFVersAsync
PBSetForeignPrivsAsync
PBSetVInfoAsync
PBSetVolAsync
PBStatusAsync
PBStatusImmed *
PBShareAsync
PBUnlockRangeAsync
PBUnshareAsync
PBWriteAsync
PBWriteImmed *
PBXGetVolInfoAsync
PageFaultFatal
PostEvent
PPostEvent
PrimeTime
RelString
RmvTime
SameProcess
SetCursor
SetFSInfo
SetWUTime
SleepQInstall
SleepQRemove
SlotVInstall
SlotVRemove
SndDoCommand *
SndGetSysBeepState
SndManagerStatus
SndPauseFilePlay
SndSetSysBeepState
SndSoundManagerVersion
PBStatusAsync
StrToAddr *
StringToExtended
StripAddress
SwapMMUMode
Translate24to32
UnholdMemory
UnlockMemory
UpTime
UTAllocateFCB
UTReleaseFCB
UTLocateFCB
UTLocateNextFCB
UTIndexFCB
UTResolveFCB
UTAddNewVCB
UTLocateVCBByRefNum
UTLocateVCBByName
UTLocateNextVCB
UTAllocateWDCB
UTReleaseWDCB
UTResolveWDCB
```

```
PBExchangeFilesAsync
PBFlushFileAsync
PBFlushVolAsync
PBGetAltAccessAsync
PBGetCatInfoAsync
PBGetEOFAsync
PBGetFCBInfoAsync
PBGetFInfoAsync
PBGetFPosAsync
PBGetForeignPrivsAsync
PBGetUGEntryAsync
PBGetVInfoAsync
PBGetVolAsync
PBGetWDInfoAsync
PBGetXCatInfoAsync
PBHCopyFileAsync
PBHCreateAsync
PBHDeleteAsync
PBHGetDirAccessAsync
PBHGetFInfoAsync
PBHGetLogInInfoAsync
PBHGetVInfoAsync
PBHGetVolAsync
PBHGetVolParmsAsync
PBHMapIDAsync
PBHMapNameAsync
PBHMoveRenameAsync
PBHOpenAsync *
PBHOpenDFAsync
PBHOpenDenyAsync
PBHOpenRFAsync
```

```
UTFindDrive
UTAdjustEOF
UTSetDefaultVol
UTGetDefaultVol
UTEjectVol
UTCheckWDRefNum
UTCheckFileRefNum
UTCheckVolRefNum
UTCheckPermission
UTCheckVolOffline
UTCheckVolModifiable
UTCheckFileModifiable
UTCheckDirBusy
UTParsePathname
UTGetPathComponentName
UTDetermineVol
UTGetBlock
UTReleaseBlock
UTFlushCache
UTMarkDirty
UTTrashVolBlocks
UTTrashFileBlocks
UTTrashBlocks
UTCacheReadIP
UTCacheWriteIP
UTBlockInFQHashP
UTVolCacheReadIP
UTVolCacheWriteIP
VRemove
WakeUpProcess
```

References

Inside Macintosh: Memory

File System Manager SDK

Designing PCI Cards and Drivers for Power Macintosh Computers

Technote 1033: "Interrupts in Need of (a Good) Time"

Technote 1067: "Traditional Device Drivers: Sync or Swim"

Technote 1084: "Running CFM-68K Code at Interrupt Time: Is Your Code at Risk"

Technote 1094: "Virtual Memory Application Compatibility".

Technote ME 09: "Coping with VM and Memory Mappings

Back to top

Change History

February 1998	Originally written.
July 1998	Updated with new and revised material: ● an expanded discussion of "interrupt time" ● a note about CFM-68K ● a discussion of software interrupt level ● Open Transport information is now cross-referenced ● complete rewrite of the Device Manager section ● a new File System Manager section, with completely rewritten material ● added Event Manager and QuickDraw sections ● many stylistic improvement
November 1999	Updated with new material in the Event Manager section to list `OSEventAvail` as interrupt safe (which was a mistake, see the January 2000 revision) and discuss alternatives to non-interrupt safe routines.
December 1999	Updated to add the Text Utilities section.
January 2000	Updated to add the Unicode Converter section and correct the discussion on `OSEventAvail` [2418891].
February 2000	Updated to discuss Communications Toolbox.
April 2000	Updated to discuss Multiprocessing Services.
October 2000	Updated to discuss mechanisms used to determine the current execution level. Updated the Event Manager section to document that `GetKeys` is interrupt-safe under CarbonLib 1.1 and later.

Back to top

Downloadables

	Acrobat version of this Note (K)	Download
	New CurrentExecutionLevel detection code	Download

Back to top