Technical Note TN1137

Disabling Interrupts on the Traditional Mac OS

CONTENTS

This Technote describes how to disable interrupts on the traditional Mac OS. It also includes a long discussion of why you **should not disable interrupts**, and outlines other system services that you can use to avoid disabling interrupts.

This Note is directed at developers who are building kernel-level software, such as device drivers, or application software that makes heavy use of Mac OS "interrupt time." In general, application developers should not need to disable interrupts.

Updated: [Aug 31 1998]

Introduction

DTS recommends that third-party developers avoid disabling interrupts on Mac OS, although we recognize that there are circumstances for which disabling interrupts is the only solution. The purpose of this Note is to highlight the possible alternatives to disabling interrupts and -- if you decide that none of these meet your needs -- to minimize errors when performing this tricky task. This note should **not** be construed as an encouragement to disable interrupts needlessly.

The note is broken up into four sections:

- The first section, "Rationale," describes why we take this position on disabling interrupts.
- The second section, "Background Material," describes how the traditional Mac OS interrupt architecture works, including how interrupts are handled through the emulator on PowerPC-based computers. If you're familiar with the 680x0 interrupt architecture, you may want to skip this section.
- The third section, "Using `InterruptDisableLib`," describes the interrupt mask manipulation library which accompanies this note.
- The final section, "Alternatives," describes the many alternatives to disabling interrupts that already exist in the Mac OS. You should read this section to see if your task is amenable to any of these alternatives.

Back to top

Rationale

Why is DTS so against disabling interrupts? It is because:

- it's harmful for the system as a whole;
- it slows down your code; and
- it can often be avoided using existing system functionality.

This section will explore each area in turn.

Latency

Disabling interrupts increases the interrupt latency of the system. This is bad for system functions, like Sound Manager, that require good interrupt latency in order to operate correctly. While somewhat long in the tooth, DTS Technote HW 16 "I Was a Teenaged DMA Junkie" describes some background on this issue.

Performance Penalty

As described in the following section, there is no way to disable interrupts quickly on Mac OS. If you're running PowerPC code, you have to take a Mixed Mode Manager switch in order to disable interrupts. If you're building 680x0 software, it's possible that your code will be run on a real 680x0 microprocessor with virtual memory enabled, in which case the modification of the SR register is a privileged operation which the system must emulate for you. Both of these represent a performance penalty.

Avoidability

The Mac OS provides many low-level primitives that you can use to avoid disabling interrupts. While it's true that many of these routines eventually do disable interrupts, using them saves you some effort and allows Apple to improve the system "behind your back."

Back to top

Background Material

This section describes the traditional Mac OS interrupt architecture, which is modeled directly after the interrupt architecture on the 680x0 microprocessors. If you're already familiar with the 680x0 interrupt architecture, you may want to skip this section.

About 680x0 Interrupt Levels

The 680x0 SR register contains a three bit field that determines the current interrupt **mask**, a value from 0 to 7. If the priority of an incoming interrupt (the interrupt's **level**) is greater than the current interrupt mask, the 680x0 (or the emulated 680x0, if you're running on a PowerPC-based computer) will raise the interrupt mask to that level and service the interrupt.

The following table summarizes the common uses for the various interrupt levels:

Interrupt Level	Typical Usage
0	Normal application-level code
1	ADB
2	SCC, NuBus, PCI
3	Built-in Ethernet
4	Logic board DMA channel interrupts
5	na
6	na
7	Non-Maskable Interrupts (NMI), usually programmer's switch

Important:
The above table is for illustrative purposes only. Unless you have intimate knowledge of the hardware on which you're running, you should not explicitly use interrupt masks other than 7, which disable all maskable interrupts.

Note:
In general, an interrupt mask of *X* will disable all interrupts levels *X* and below. However NMIs are never masked, even when the interrupt mask is *7* .

The 680x0 SR register is a privileged register; it can only be accessed from code running in 680x0 supervisor mode. When you access the SR register from user mode, the 680x0 takes a privilege violation exception. The traditional Mac OS catches this exception and emulates the offending instruction. So, apart from the slow down caused by the exception, you can ignore this restriction and access the SR register from any 680x0 software.

For more information on SR register emulation, consult Technote 1094 "Virtual Memory Application Compatibility".

Interrupt Levels on a PowerPC

The raw PowerPC processor only has a single-interrupt-state bit: interrupts are either masked or they aren't. The interrupt mask bit is a privileged bit, so you must be running the native PowerPC processor in native supervisor mode to be able to access it. However, Mac OS runs all PowerPC code (except the nanokernel, see the next section) in user mode, so you cannot access the PowerPC interrupt mask bit from PowerPC code.

Mac OS Interrupt Architecture on PowerPC

As described above, the 680x0 and PowerPC microprocessors have quite different interrupt architectures. One of the key features of the Mac OS interrupt architecture on PowerPC-based computers is that it emulates the 680x0 interrupt architecture. This is necessary because a significant quantity of code (both in the traditional Mac OS and third party) needs to disable interrupts, including selectively masking interrupts at a specific level.

For example, consider the traditional Mac OS serial driver. As serial interrupts occur at level 2, the serial driver interrupt handler knows that, as long as it keeps the interrupt mask at 2 or higher, no other serial interrupt can occur. Many drivers use this assumption to provide concurrency control for their global data structures.

Note:
The above is an example only. On modern Mac OS computers, serial interrupts do not necessarily occur at interrupt level 2. See About 680x0 Interrupt Levels for details.

So, for compatibility purposes, all interrupts on Power Macintosh computers are prioritized as if they were on a 680x0-based computer. When external hardware interrupts the PowerPC processor, the interrupt is initially serviced by the **nanokernel**, which takes one of two actions depending on the state of the machine:

1. If the machine is currently running 680x0 software (by means of the **emulator**), the nanokernel signals the emulator that an interrupt of a specific priority has occurred and returns from the external interrupt handler. The next time the emulator finishes executing a 680x0 instruction (or basic block in the case of the Dynamic Recompiling emulator), it notices this interrupt. If the interrupt level is not masked, the emulator services it in the traditional fashion, by building a 680x0 exception frame and calling the 680x0 interrupt handler through the vector pointed to be the 680x0 Vector Base Register (VBR).
2. If, on the other hand, the machine is currently running PowerPC code, it switches back to the emulator context and takes a special exception to handle the interrupt.

The emulator executes 680x0 instructions atomically with respect to interrupts, as they were in the original 680x0 processors. This preserves the atomicity implied in 680x0 interrupt-handling code. The dynamic recompiling emulator may check for interrupts with less granularity due to the larger sections of native code it builds.

In summary, on a Power Macintosh, all interrupts are routed by the nanokernel through the emulator for servicing to achieve faithful emulation of 680x0 interrupts levels and to keep 680x0 instructions indivisible.

Theoretical Background for `InterruptDisableLib`

The above discussion yields two important consequences. First, native PowerPC code cannot access the native PowerPC interrupt mask because it's in a privileged register. Second, disabling interrupts from 680x0 software will effectively disable all the interrupts on the machine, because all interrupts are routed through the emulator. These two facts combine to yield the following result:

> **The only way to disable interrupts on Mac OS is to modify the interrupt mask in the 680x0 SR register. This is true even if your program is compiled for PowerPC.**

For this reason, code that wants to disable interrupts must contain two code paths. The 680x0 code path can modify the SR register directly. The PowerPC code path must use the "MixedMode.h" routine `CallUniversalProc` to call 680x0 code that modifies the SR register.

> **Note:**
> In normal circumstances, CFM-68K code acts exactly like CFM-PPC code: it must use Mixed Mode Manager to call any classic 68K code. However, when disabling interrupts, the CFM-68K code can take the same code path as the classic 680x0. Unlike CFM-PPC code, CFM-68K code can access the 680x0 SR register directly, and doing so avoids Mixed Mode Manager switches.

Back to top

Using `InterruptDisableLib`

A code sample called `InterruptDisableLib` is provided for you as an attachment to this technote. The code uses the technique discussed in the previous section to provide easy-to-use control over the 680x0 interrupt mask from classic 68K, PowerPC, and CFM-68K code. The code is structured as a complete library which you can drop in to your project, with C and Pascal interfaces, and a C implementation.

The library was compiled and tested using the Metrowerks CodeWarrior Pro C and Pascal compilers; however, you should be able to use the source with any C, C++, or Pascal compiler. To use it in your project, you need to take the following steps:

1. Add the "InterruptDisableLib.c" file to your program. If you're using an integrated development environment that supports C code, you will be able to add it directly to your project. Otherwise you may need to compile it separately and add it as an object file.
2. Include the appropriate interface file. For C/C++ programmers, you should include "InterruptDisableLib.h". Pascal programmers should use "InterruptDisableLib.p".
3. When you need to disable interrupts, do so using:

```
oldMask = SetInterruptMask(7);

// Interrupts are now disabled, do your stuff!

(void) SetInterruptMask(oldMask);
```

Library Reference

The library contains but two entry points:

```
extern pascal UInt16 GetInterruptMask(void);
extern pascal UInt16 SetInterruptMask(UInt16 newMask);
```

The `GetInterruptMask` function returns the current 680x0 interrupt mask as a value from 0 to 7. The `SetInterruptMask` function sets the current 680x0 interrupt mask as a value from 0 to 7. It also returns the prior interrupt mask.

Gotchas

This section contains some important rules. Rules must sometimes be broken, but you should think very carefully before breaking these!

- Never never never use the `GetInterruptMask` function to determine whether your code is running at "interrupt time." There are numerous ways a Mac OS computer can have interrupts enabled (an interrupt mask of 0) but still be running at "interrupt time." These include VBLs, Deferred Tasks, and PCI Secondary Interrupts.
- Never lower the interrupt mask inside an interrupt handler. For example, if your interrupt handler is entered with the interrupt mask set to 2, never lower the mask below that. The reason? A hardware driver may have set the interrupt mask to 1 to prevent it being reentered. If your interrupt happens at level 2, and then you lower the interrupt mask to 0, you will allow interrupts that the other driver is not expecting.
- Always restore the old interrupt mask. Never assume that you're running with a particular interrupt mask, and restore that mask by explicitly setting it. Always save the old mask and restore it when you're done.

Back to top

Avoidance

Before disabling interrupts, you should investigate the following system services to see if you can use them instead.

OS Utilities

The ancestors of all atomic operations on the Mac OS are the OS Utilities routines `Enqueue` and `Dequeue`. These routines are available on all Mac OS computers and provide simple atomic queue manipulation. They are, however, implemented as 680x0 code, so using them will cause a Mixed Mode Manager switch.

Deferred Tasks

If you place all your critical sections in deferred tasks, you can take advantage of the Deferred Task Manager's guarantee that all deferred tasks are serialized.

Open Transport Utilities

Open Transport provides a plethora of kernel-level services, including the following interrupt safe constructs:

- LIFO queues
- atomic bit and arithmetic operations
- atomic compare and swap
- Open Transport deferred tasks
- `OTGate` (provides critical section support)
- memory allocation

All of these OT primitives are implemented completely in native code on the PowerPC.

DriverServicesLib

If you're writing a PCI device driver, DriverServicesLib provides myriad low-level queue manipulation and atomic operations.

The DriverServicesLib queue manipulation routines are similar to the OS Utilities routines except that they have fast-code paths that avoid Mixed Mode Manager switches when adding to an empty queue or removing from a one-element queue.

680x0 Atomic Instructions

As described above, 680x0 instructions remain atomic even when emulated on the Power Macintosh. Thus, run-of-the-mill 680x0 instructions like addq and bset are all atomic operations on both 680x0- and PowerPC-based computers.

PowerPC Atomic Instructions

The PowerPC processor contains two special instructions, Load Reserved (lwarx) and Store Conditional (stwcx), which you can use to implement atomic operations. Most compilers provide access to these instructions via intrinsic functions.

> **Important:**
> DTS recommends that developers avoid using the PowerPC Load Reserved and Store Conditional instructions. There are two reasons for this. Firstly, these instructions are inherently processor-specific and reduce the portability of your code. Secondly, the behavior of these instructions varies between PowerPC CPU types. Accommodating all these variations is tricky. These instructions do not provide much utility beyond that provided by the Open Transport and DriverServicesLib atomic routines, and Apple ensures that these atomic routines are updated to do the right thing in all cases.

Back to top

Summary

DTS recommends that developers avoid disabling interrupts. Mac OS provides many alternatives to disabling interrupts. If none of these alternatives meet your needs, you can disable interrupts by setting the interrupt mask in the 680x0 SR register. To do this from PowerPC code, you must call 680x0 software using Mixed Mode Manager. If you do this, you should use the code from InterruptDisableLib to prevent common mistakes, and make sure you read the list of caveats in this Technote.

Back to top

References

Technote HW 16 "I Was a Teenaged DMA Junkie"

Technote 1094 "Virtual Memory Application Compatibility" discusses the SR instruction emulation done by the Virtual Memory Manager on 680x0-based computers.

PowerPC Microprocessor Family: The Programming Environments For 32-Bit Microprocessors discusses the native PowerPC interrupt model.

M68000 Family Programmer's Reference Manual discusses the 680x0 interrupt model.

Inside Macintosh: PowerPC System Software, Chapter 2 Mixed Mode Manager

Inside Macintosh: Operating System Utilities, Chapter 6 Queue Utilities

Inside Macintosh: Processes, Chapter 6 Deferred Task Manager

See the Open Transport web page for references to Open Transport technical documentation.

Designing PCI Cards and Drivers for Power Macintosh Computers

Back to top

Downloadables

	Acrobat version of this Note (K).	Download
	Binhexed Routine InterruptDisableLib (179K).	Download

Back to top
