
Porting Drivers to Mac OS X

Darwin



2009-05-06



Apple Inc.
© 2004, 2009 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

.Mac is a registered service mark of Apple Inc.

Apple, the Apple logo, Mac, and Mac OS are trademarks of Apple Inc., registered in the United States and other countries.

UNIX is a registered trademark of The Open Group

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE

ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Introduction **Introduction to Porting Drivers to Mac OS X** 7

See Also 7

Chapter 1 **Driver Porting Basics** 9

Does Your Driver Belong in the Kernel? 9
What's My Class? 9
Workloops vs. Interrupt Handlers 10
IOMemoryDescriptor vs. kvtophys 11
C++ Driver Model 12
Data Types 12
ioctl Handling 13
sysctl and syscall Handling 15
Interrupt Priority Levels (IPL/SPL) vs. IOLock Locks 15

Chapter 2 **I/O Kit Considerations** 17

IOLog vs. multi-level logging 17
Asynchronicity and Synchronous Returns 17
timeout, sleep and untimeout 19
Namespace Pollution 21
 Using Data Structures to Avoid Cross-Instance Namespace Pollution 21
 Using Macros to Avoid Cross-Driver Namespace Pollution 23

Chapter 3 **C++ Language Considerations** 25

Data Type Size Differences 25
Handling C to C++ Callbacks 25

Document Revision History 27

Listings

Chapter 2 **I/O Kit Considerations** 17

- Listing 2-1 Use of IOLock for sleep/timeout/untimeout 20
- Listing 2-2 Use of instance table for global variables 21

Introduction to Porting Drivers to Mac OS X

This book was intended to be used by developers who have existing drivers for other platforms, particularly classic Mac OS and other UNIX-based operating systems. Its goal is to provide you with useful ways to port these non-I/O Kit drivers to Mac OS X with minimal reengineering.

Before you begin porting a device driver to Mac OS X, you should read *I/O Kit Fundamentals*.

This book is only relevant if you are porting a driver that resides in the kernel. To find out if this applies to your driver, read "[Does Your Driver Belong in the Kernel?](#)" (page 9).

Important: This document is preliminary. Although it has received some technical review, there may be changes or additions to some of the information provided here.

See Also

For information on specific technology areas, you should consult the appropriate API reference in the I/O Kit section of Apple's Technical Publications website.

For more information on the Mac OS X Kernel and the I/O Kit, see the books *I/O Kit Fundamentals* and *Kernel Programming Guide*.

For more information on driver porting, two good sources of information are the *Darwin-drivers* and *Darwin-development* mailing lists. For more information, visit <http://www.lists.apple.com>.

INTRODUCTION

Introduction to Porting Drivers to Mac OS X

Driver Porting Basics

This chapter describes some of the fundamental differences between traditional UNIX-style drivers (including BSD and Linux) and Mac OS X's I/O Kit drivers. As with any subject as broad as driver porting, there are many details that are specific to the technology area in question. This chapter does not attempt to cover such issues. Instead, it focuses on the more general issues, leaving the implementation details as an exercise for the reader.

Does Your Driver Belong in the Kernel?

This document primarily applies to in-kernel device drivers. Not all device drivers in Mac OS X are in the kernel. For example, most human interface devices (keyboards, mice, and so on) and graphics and imaging devices (printers, scanners, and similar) are controlled by user-space drivers.

In general, your driver should be in the kernel if it is used concurrently across multiple applications or if its primary client is in the kernel. Examples include video drivers, disk or disk controller drivers, and so on.

There are a few cases where a driver has to reside in the kernel even though similar devices can reside in user space. In particular, any device on a PCI or AGP bus must be in the kernel because those busses do not export interfaces to user space for security reasons.

For the most part, PCI devices are things that need to be in the kernel anyway. There are a few exceptions, however, such as MIDI device drivers. While MIDI device drivers live in user space, PCI device drivers must reside in the kernel. Thus PCI cards with MIDI interfaces require two drivers: the actual I/O Kit driver for the device (subclassing from the PCI driver family) and a CFPlugIn in user space to provide a driver to CoreMIDI. Interfacing drivers across user-kernel boundaries is beyond the scope of this book. For help creating drivers that span user-kernel boundaries, you should contact Apple Developer Technical Support.

What's My Class?

Before you actually start porting code, you should create a stub driver using the correct base class for your type of device. The *Kernel Framework Reference* provides documentation for each of these base classes.

In general, you will need to use two classes in your drivers: a base class (whose name typically ends with "driver") and a nub class (whose name typically ends with "device").

The base class instance will represent the driver itself. It must then instantiate an instance of the nub class for each device that it is controlling. This nub abstraction provides a mechanism for other parts of the system to actually use your driver to communicate with the device. The mechanism for instantiating this nub varies from class to class.

For detailed information on this process, you should read *I/O Kit Fundamentals* and follow the Hello I/O Kit tutorial.

Workloops vs. Interrupt Handlers

Drivers in Mac OS X are designed around the concept of a workloop. (Many drivers on other UNIX-based platforms are written in a similar fashion, but not as part of the primary driver model.)

The idea of a workloop is simple. It is basically just a helper thread. When an interrupt occurs, instead of a handler routine being executed immediately, the kernel sets that thread to “ready to execute”. Thus, the interrupt handling occurs at a kernel thread priority instead of at an interrupt priority. This ensures that interrupts don’t get lost while handling routines have interrupts turned off. It also means that routines running in an interrupt service thread context do not have to obey the same rules as an actual interrupt handler; they can block, call IOLog, and so on.

Creating a workloop is relatively simple. The following code is an example of registering to receive two interrupts using a workloop:

```

/* class variables */
IOWorkLoop *myWorkLoop;
IOInterruptEventSource *interruptSource;
IOInterruptEventSource *DMAInterruptSource;

/* code in start() */
myWorkLoop = IOWorkLoop::workLoop();

if( myWorkLoop == NULL ) {    IOLog( "org_mklinux_iokit_swim3_driver::start:
Couldn't allocate "
    "workLoop event source\n" );
    return false;
}

interruptSource = IOInterruptEventSource::interruptEventSource(
    (OSObject*)this,
    (IOInterruptEventAction)&org_mklinux_iokit_swim3_driver::handleInterrupt,
    (Filter)&org_mklinux_iokit_swim3_driver::interruptPending,
    (IOService*)provider,
    (int)0 );

if ( interruptSource == NULL ) {
    IOLog( "org_mklinux_iokit_swim3_driver::start: Couldn't allocate "
        "Interrupt event source\n" );
    return false;
}

if ( myWorkLoop->addEventSource( interruptSource ) != kIOReturnSuccess ) {
    IOLog( "org_mklinux_iokit_swim3_driver::start - Couldn't add Interrupt"
        "event source\n" );    return false;}

DMAInterruptSource = IOInterruptEventSource::interruptEventSource(
    (OSObject*)this,
    (IOInterruptEventAction)&org_mklinux_iokit_swim3_driver::
        handleDMAInterrupt,    (IOService*)provider,    (int)1 );

if ( DMAInterruptSource == NULL ) {
    IOLog( "org_mklinux_iokit_swim3_driver::start: Couldn't allocate "
        "Interrupt event source\n" );
    return false;
}

```

```

if ( myWorkLoop->addEventSource( DMAInterruptSource ) != kIOReturnSuccess )
{
    IOLog( "org_mklinux_iokit_swim3_driver::start - Couldn't add "
          "Interrupt event source\n" );
    return false;
}

myWorkLoop->enableAllInterrupts();

```

The methods `handleInterrupt` and `handleDMAInterrupt` will now be called for the first and second device interrupts (offsets 0 and 1), respectively.

For the most part, this interrupt handler will behave much like any interrupt handler would in any other OS (except that you can call `printf` and `IOLog` safely). However, there are some exceptions, particularly in the area of interrupt priority.

You should note that this source example uses `IOWFilterInterruptEventSource` instead of `IOWInterruptEventSource`. This is strongly recommended for two reasons. First, if you are writing a driver for a multifunction PCI card, this will avoid having the OS call all of the drivers for all of the devices on that card. Second, if your card is installed into an environment where an interrupt line gets shared, this will significantly improve performance.

Unlike `IOWInterruptEventSource`, when you create an `IOWFilterInterruptEventSource` object, you pass in a filter function in addition to the handler.

In the filter function (`org_mklinux_iokit_swim3_driver::interruptPending` in this example), you should test to see whether your device generated the interrupt, and return `true` if your driver should handle the interrupt or `false` if it belongs to another device on the card. This generally involves polling a register in your device to see which interrupt flags are set, then storing the value for later use. Note that because the filter function runs in the primary hardware interrupt context, you must treat it like any other direct hardware interrupt—don't block, don't call `IOLog`, don't copy data around or allocate memory, and so on.

For more information, read *I/O Kit Fundamentals* and *Kernel Programming Guide*.


IOMemoryDescriptor vs. kvtophys

For any given location in a computer's memory, computer architectures define (at least) three distinct addresses: the physical address (the actual address line values as seen by the processor), the logical/virtual address (the address as seen by software), and the bus address (the address as seen by an arbitrary I/O device).

On most 32-bit hardware, the physical address is the same as the bus address. Thus, most people ignore the distinction. In Mac OS X, you must treat them as two different things, particularly on 64-bit hardware. For this reason, using physical addresses obtained with `kvtophys` is unsafe, and the function `kvtophys` has actually been removed from availability to KEXTs to prevent its improper use.

Instead, you should use an `IOMemoryDescriptor` for your memory allocation so that you can obtain the bus address associated with the block of memory and hand that information to the peripheral.

To do this, you use the method `IOMemoryDescriptor::getPhysicalSegment`. The first argument is an offset from the beginning of the descriptor. The second is the address where the segment length should be stored. If the segment length returned is less than the total descriptor length (which you can obtain using the method `IOMemoryDescriptor::getLength`), then you must get the physical address of the next segment, and so on, until you have dealt with the entire descriptor.

 **Warning:** You *must* call `IOMemoryDescriptor::Prepare` on the descriptor before using `getPhysicalSegment` to ensure that appropriate physical to bus address mappings are configured prior to initiating a bus transaction. Don't forget to call `IOMemoryDescriptor::Complete` after the transaction is completed. Failure to do so can cause data corruption and random crashes.

For more information about the `IOMemoryDescriptor` class, see the I/O Kit reference documentation, available from Apple's developer documentation website.

C++ Driver Model

I/O Kit drivers are written in C++. Most device drivers in other operating systems are written in C. This often poses interesting issues, primarily related to the way driver-specific data is stored. It also can lead to a number of other surprises. These are described further in "[C++ Language Considerations](#)" (page 25).

Data Types

Data types tend to be of different sizes in various driver models. To avoid any nasty surprises, you should always check the bit width of the various data types in the original OS, then explicitly use data types of the same width. For example, if type `int` on Linux is 32 bits, for maximum longevity, you should use the type `uint32_t` in Mac OS X.

Suggested types are:

```
uint8_t—unsigned 8-bit integer
uint16_t—unsigned 16-bit integer
uint32_t—unsigned 32-bit integer
uint64_t—unsigned 64-bit integer
int8_t—signed 8-bit integer
int16_t—signed 16-bit integer
int32_t—signed 32-bit integer
int64_t—signed 64-bit integer
```

The definition for these types can be included in both user-space and kernel-space code with the following:

```
#include <stdint.h>
```

You can also use the equivalent Mac-specific types, such as `UInt32`, which can be included with the following:

```
#include <libkern/OSTypes.h>
```

A note of caution: boolean variables can be problematic as there is no standard rule about their size. Rather than use built-in boolean types, you should generally stick with an explicit integer of your choice of sizes to avoid inadvertent alignment differences between your C and C++ code.

ioctl Handling

Mac OS X does not use `ioctl` support at the driver level. Instead, higher level driver families handle `ioctl` calls and turn them into explicit function calls. The details of these calls are dependent on the type of driver in question.

If the driver family you are using does not provide an `ioctl` that you need, you can either file a bug requesting that the `ioctl` be added or you can provide a similar solution through the use of a user client/device interface pair. These are described in more detail in the document *I/O Kit Device Driver Design Guidelines*, available from the Apple Technical Publications website.

If none of these options is acceptable, you can also sometimes subclass the BSD user client for a particular class of devices (such as `IOMediaBSDClient`) and add additional `ioctl` support. For example, to add an `ioctl` for a particular type of media, you would need to override the following methods:

- `start`—determine if the media is of the desired type
- `ioctl`—handle the `ioctl`

A brief code snippet follows:

```
bool FloppyMediaBSDClient::start(IOService *provider)
{
    IOMedia * media = (IOMedia *) provider;
    u_int64_t size;
    IOStorage *storage;

    /* Make sure our provider's start routine succeeds */
    if (super::start(provider) == false)
        return false;

    /* Make sure our provider is a block storage device */
    media = getProvider();
    storage = media->getProvider();
    this->driver = OSDynamicCast(IOBlockStorageDriver, storage);
    if (!this->driver)
        return false;

    /*
     * Determine if this is really the type of media we're looking for,
     * in this case by its size. This could also be done using a more
     * specific OSDynamicCast if we are looking for a particular driver
     * to be upstream.
     */
    size = media->getSize() / (u_int64_t) 512;

    switch (size) {
    case 0: // unformatted
    case 720: // 360k
    case 800: // 400k
```

```

        case 1440: // 720k
        case 1600: // 800k
        case 2880: // 1440k
        case 2881: // 1440k also
        case 3360: // 1680k
        case 5760: // 2880k
            dIOLog("Floppy Disk Media Detected.\n");
            break;
        default:
            dIOLog("Non-floppy disk media detected: %ld\n", (unsigned long)size);
            return false;
    }
    return true;
}

int FloppyMediaBSDClient::ioctl (
    dev_t      dev,
    u_long     cmd,
    caddr_t    data,
    int        flags,
    struct proc *  proc )
{
    //
    // Process a Floppy-specific ioctl.
    //

    int *buffer;
    int error = 0;
    IOReturn status = kIOReturnSuccess;
    int formatflags;

    switch(cmd) {
        case FD_VERIFY:
            buffer = NULL;
            break;
        case FD_FORMAT:
            buffer = (int *)data;
            if (!buffer) {
                dIOLog("ioctl (floppy): null buffer!\n");
                error=EINVAL;
                break;
            }
    }
    switch (cmd)
    {
        case FD_VERIFY:
        {
            IOLog("Got FD_VERIFY -- not supported yet.\n");
            error = ENOTTY;
            break;
        }
        case FD_FORMAT:
        {
            formatflags = *buffer;
            IOLog("Got FD_FORMAT -- not supported yet (flags = 0x%x).\n",
                formatflags);
            error = ENOTTY;
            break;
        }
    }
}

```

```

    }
    default:
    {
        //
        // A foreign ioctl was received. Ask our superclass' opinion.
        //
        IOLog("fd: unknown ioctl, calling parent.\n");
        error = super::ioctl(dev, cmd, data, flags, proc);
        break;
    }
}
return error; // (return error status)
}

```

sysctl and syscall Handling

Much like `ioctl` support, Mac OS X drivers do not generally use `sysctl` or `syscall` interfaces except those provided by their respective families. Instead, user client/device interface pairs are used. These are described in more detail in the document *I/O Kit Device Driver Design Guidelines*, available from the Apple Technical Publications website.

However, Mac OS X does provide ways of adding additional `sysctl` and `syscall` support. This is described in detail in the document *Kernel Programming Guide*.

Interrupt Priority Levels (IPL/SPL) vs. IOLock Locks

Many UNIX-based driver models use interrupt priority levels as a means of protecting critical sections in drivers using functions like things like `splhigh`, `splbio`, and `splx`. Using interrupt priority to protect critical sections doesn't work particularly well on SMP systems, and thus most operating systems are moving away from this design. However, these functions are still in common use.

Mac OS X does not support the use of interrupt priority levels for disabling interrupts for critical section protection. Instead, you should use locks, semaphores, or other synchronization mechanisms. If you compile your code using these functions, the code may compile, but will either not work properly (because the function in question is a no-op) or will result in a kernel panic, depending on the functions used.

Instead of using these functions, you should generally use `IOLock` mutex lock. These are described in the *Kernel Framework Reference*, and are briefly summarized below:

```

/* Allocate an I/O Lock */
IOLock *IOLockAlloc( void );

/* Free an I/O Lock */
void IOLockFree( IOLock * lock);

/* Lock an I/O Lock */
static __inline__ void IOLockLock( IOLock * lock);

/* Lock an I/O Lock if it doing so would not block. Returns true if
   lock was obtained. */
static __inline__ boolean_t IOLockTryLock( IOLock * lock);

```

```
/* Unlock an I/O Lock */
void IOLockUnlock( IOLock * lock);

/* Wait on condition specified by event (where event is usually
   generated by taking the address of a variable, much like
   timeout/untimeout in BSD */
static __inline__ int IOLockSleep(
    IOLock * lock,
    void *event,
    UInt32 interType);

/* Similar to IOLockSleep, only with a timeout specified */
static __inline__ int IOLockSleepDeadline(
    IOLock * lock,
    void *event,
    AbsoluteTime deadline,
    UInt32 interType);

/* Wake up an event waiting on the condition specified by event
   The boolean oneThread specifies whether to signal on the condition
   (wake the first waiting thread) or broadcast (wake all threads
   that are waiting).
   */
static __inline__ void IOLockWakeup(IOLock * lock,
    void *event,
    bool oneThread);
```

If you find that you can't live without a semaphore implementation, you can either implement one using an `IOLock` or use the Mach semaphores described in *Kernel Programming Guide*. The former is strongly recommended, as binary compatibility is not guaranteed for KEXTs that use Mach directly.

I/O Kit Considerations

The I/O Kit is a powerful and relatively straightforward driver environment. However, as with most things that are powerful, there are a few things that won't behave the way you might expect, particularly if you are used to writing drivers for other platforms.

This chapter describes some of the things you should consider when porting a driver from another operating system, including ways to avoid potential kernel panics, driver loading failures, and so on down the road

IOLog vs. multi-level logging

Some UNIX driver models provide additional support for logging, such as the use of multiple levels of verbosity. The I/O Kit does not provide such support, relying on the developer to implement such a scheme if it is desired.

A convenient way to simulate this is to replace calls to `printf` with calls to the a macro like this one:

```
#define dIOLog(level, a, b...) {if (org_mklinux_iokit_swim3_debug & \
    level) IOLog(a, ## b); }
```

and then define various level macros as powers of two. This provides even greater control over debugging than a level-based scheme, allowing you to have up to 32 distinct log options (or 64 if you declare the debug variable as a 64-bit type) that can be turned on or off individually by changing the value of a global variable (in this case, called `org_mklinux_iokit_swim3_debug`).

Note that if you are going to use this sort of debugging in a C driver core, you must either use a global variable, pass the variable's value as an argument, or pass a pointer to the variable as an argument, since the C code cannot access class variables directly.

Asynchronicity and Synchronous Returns

In many parts of the I/O Kit, the template class includes both synchronous and asynchronous methods. In these cases, it is absolutely necessary to implement both asynchronous and synchronous calls. Asynchronous calls usually include a callback for a completion routine. If you call the `complete` method on this completion routine from the same thread that called your function, you may get random kernel panics or stack corruption.

The problem is that most traditional driver architectures are not designed with asynchronous I/O in mind. There are a number of ways to retrofit asynchronous support into a driver. Each has its own difficulties.

The most obvious solution is to spawn a temporary helper thread to perform the operation. However, this runs into problems with concurrency. Specifically, the helper thread needs to be able to obtain data from the main thread, which means that the buffer must be in a shared location (for example as part of the class instance). However, some other thread could be in the same code at the same time, resulting in corruption of the request.

The obvious (but wrong) fix for this is to use a lock. Mach locks, however, do not like it when you lock them in one thread and release them in another. Instead, you should use an `IOCommandGate`.

In your class declaration, you should include a place to store the command gate:

```
IOCommandGate *myGate;
```

In your start routine, you must initialize the command gate. To do this, you might add code that looks like this:

```
myGate = new IOCommandGate();
```

You need a wrapper function to pass arguments to the `doSyncReadWrite` function. It might look like this:

```
struct asyncAction {
    IOMemoryDescriptor *buffer;
    UInt32 block;
    UInt32 nblks;
};

extern "C" static void org_mklinux_iokit_swim3_driver_doAsyncWrapper(
    void *selfref,
    void *actref,
    void *completionptr,
    void *)
{
    int retval;
    org_mklinux_iokit_swim3_driver *driver = selfref;

    retval = driver->doSyncReadWrite(myaction->buffer,
        myaction->block,
        myaction->nblks);

    IOStorage::complete(completion, retval, (retval ? 0 :
        (args.nblks * 512)));
}
```

In `doAsyncReadWrite`, you would do something like:

```
org_mklinux_iokit_swim3_driver::doAsyncReadWrite(IOMemoryDescriptor *buffer,
        UInt32 block, UInt32 nblks,
        IOStorageCompletion completion)
{
    struct asyncAction myaction;

    myaction.buffer = buffer;
    myaction.block = block;
    myaction.nblks = nblks;

    return (myGate->runAction(
        (Action)&org_mklinux_iokit_swim3_driver_doAsyncWrapper
        (void *)self,
        (void *)&myaction,
        (void *)&completion,
        NULL));
}
```

```
}

```

timeout, sleep and untimeout

Many traditional BSD-style drivers use the functions `sleep`, `timeout`, and `untimeout` as synchronization primitives akin to condition variables. You might have a block of code that looks something like this:

```
timeout((timeout_fcn_t) wakeup, event, (2 * timeout * HZ + 999) / 1000);
sleep((char *)event, PZERO);
```

where `event` is a pointer to an integer where the result of the wait will be stored, `wakeup` is a (bogus) function pointer called when the event occurs, the timeout is usually measured in some fraction of a second, and `PZERO` is the priority level of the current operation. This ensures that lower-priority interrupts do not wake this code prior to timeout. In such a system, interrupts are lower than `PZERO`, so they could still result in a wakeup.

Such code would also typically contain something like the following:

```
untimeout((timeout_fcn_t) wakeup, (void *) event);
```

in a different part of the code (generally in an interrupt handler). This part of the code wakes up the part of the code that is waiting for an event.

Mac OS X uses similar constructs in the BSD part of the kernel. However, these are not exposed to the I/O Kit. Instead, you need to use an alternative mechanism.

If you are not using timeouts, you could simply use an `IOCommandGate` instance. However, this is rarely the case in code ported from BSD.

There are two recommended solutions to this: `IOLock` locks and an `IOTimerEventSource`/`IOCommandGate` combination

`IOLock` locks

This is the easiest way to replace the `timeout/sleep/untimeout` combination. Instead of setting the timeout and sleeping, you calculate the time stamp when you want to wake, take a lock, and sleep on the lock.

`IOCommandGate` with an `IOTimerEventSource`

If you are rearchitecting your code, this provides some additional flexibility at the cost of complexity. This will be described in a forthcoming code example.

This code sample assumes that the use of priority is strictly to prevent stray wakeups. If your code is doing something more sophisticated with priority levels, you will have to substantially enhance this code.

This code also assumes that there is only one request in flight at any given time, and one thread doing a sleep wait at any given time. Otherwise, you will need to add some sort of queue in place of simply having a single variable to hold the return status of the wait.



Warning: If you use this code, you *must* change the names of *all* functions and global variables to something appropriate to your driver. You may also, if desired, use macros to change the names of these functions transparently.

Remember that the C function namespace is shared, and that if you pollute it, you will eventually run into some other driver that uses the same name.

For more information on namespace pollution, read about C naming conventions in the style chapter of *Kernel Programming Guide*

With those caveats, an alternative to sleep and wakeup follows:

Listing 2-1 Use of IOLock for sleep/timeout/untimeout

```
#include <kern/kern_types.h>    /* for THREAD_UNINT */
#include <IOKit/IOLocks.h>     /* for IOLock */
#include <osfmk/kern/clock.h>  /* for time manipulation functions */

/* instance-specific driver state structure */
struct driverstate_t {
    IOLock *lock;
    int wait_return;
    .
    .
    .
}

/* This function waits */
dosomething( ... , driverstate_t mydriverstate)
{
    IOLock *lock = mydriverstate->unitlock;
    int nanoseconds_to_delay = 1000000; /* 1 msec for an example
    AbsoluteTime absinterval, deadline;
    .
    .
    .
    // timeout((timeout_fcn_t) wakeup, event, (2*timeOut * HZ + 999) / 1000);
    // sleep((char *)event, PZERO);

    IOLockLock(lock);
    nanoseconds_to_absolutetime(nanoseconds_to_delay, (uint64_t *)&absinterval);
    clock_absolutetime_interval_to_deadline(absinterval, (uint64_t *)&deadline);
    IOLockSleepDeadline(lock, event, deadline, THREAD_UNINT);
    wakeup = mydriverstate->wait_return;
    IOLockUnlock(lock);
}

/* This function wakes up the waiting thread */
interrupt_handler( ... , driverstate_t mydriverstate)
{
    // untimeout((timeout_fcn_t) wakeup, (void *) event);
    IOLockLock(lock);
    mydriverstate->wait_return =
```

```

        IOLockUnlock(lock);
    }

```

Namespace Pollution

There are two major fundamental namespace problems with porting drivers from UNIX to Mac OS X: sharing between drivers and sharing between driver instances.

Namespace pollution is a problem for any driver developer that is using C code. The problem is that your function namespace is potentially shared with other drivers. If you name a C function or global variable with the same name as that of a function or global variable in someone else's driver, one of the two drivers will fail to load.

Namespace pollution also is a problem for drivers that depend on global variables if it is possible for more than one instance of the driver to be loaded at a time. Because C global variables are shared across driver instances, the two instances will end up clobbering each other's data. (Since the driver is already loaded, there will be no linking problems, unlike the case where two different drivers have variables or functions with the same name.)

This section describes solutions to both of these problems in detail.

Using Data Structures to Avoid Cross-Instance Namespace Pollution

The biggest problem with using a C core for an I/O Kit driver is the issue of a shared namespace. While this won't prevent multiple instances from being instantiated, it will mean that they will share any global variables across those multiple instances. For this reason, special care is needed when working with global variables.

Most UNIX-based driver architectures solve this by providing some sort of unit argument to their core functions. As a quick fix, the addition of a "used" flag in the data structure can be used to allow the C++ core to provide this same functionality, with some fixed limit to the number of concurrent instances. This is not an ideal long-term solution, however, and should only be used during early development.

To do this, however, multiple driver instances must cooperate. This requires a shared lock between them. Sharing the lock is easy. Initializing the lock without race conditions, however, requires a little more effort, specifically a static initializer. An example of this methodology is shown in [Listing 2-2](#) (page 21).

Listing 2-2 Use of instance table for global variables

```

/**** in the header ****/

/* Added "used" since there are no fixed interface numbers */
struct scsipi {
    int used = 0;
    .
    .
    .
};
struct scsipi interface[NSCSI];

kern_return_t read(int unit, ...)

```

```

{
.
.
.
}

class org_mklinux_driver_IOLockClass : public OSObject
{
    org_mklinux_driver_IOLockClass();
    ~org_mklinux_driver_IOLockClass();

    public:
        IOLock *lock;
};

/**** in the C++ wrapper ****/
extern "C" {
    IOLock *org_mklinux_driver_swim3_lock = NULL;
    int org_mklinux_driver_swim3_lock_refcount = 0;
    extern scsipi *interface;
}

/* declare a statically-initialized instance of the class so that
   its constructor will be called on driver load and its destructor
   will be called on unload. */
class org_mklinux_driver_IOLockClass org_mklinux_driver_swim3_lock;

class org_mklinux_driver_IOLockClass : public OSObject
{
    org_mklinux_driver_IOLockClass()
    {
        lock = IOLockAlloc();
    }
    ~org_mklinux_driver_IOLockClass()
    {
        IOLockFree(lock);
    }

    public:
        IOLock *lock;
};

int org_mklinux_driver_swim3::new_unit() {
    int i;
    IOLockLock(org_mklinux_driver_swim3_lock->lock);
    for (i=0; i<NSCSI; i++) {
        if (!interface[i]->used) {
            interface[i]->used = 1;
            IOLockUnlock(org_mklinux_driver_swim3_lock->lock);
            return i;
        }
    }
    IOLockUnlock(org_mklinux_driver_swim3_lock->lock);
    return -1;
}

void org_mklinux_driver_swim3::free_unit(int unit) {

```

```

    IOLockLock(org_mklinux_driver_swim3_lock->lock);
    interface[unit]->used = 0;
    IOLockUnlock(org_mklinux_driver_swim3_lock->lock);
}

org_mklinux_driver_swim3::init(){
    .
    .
    .
    /* unit should be an instance variable in your c++ wrapper class */
    unit = new_unit()
    if (unit == -1) {
        /* Maybe print a warning here */
        return false;
    }
}

org_mklinux_driver_swim3::free(){
    unit_free(unit)
}

```

A better long-term solution, however, is to declare the data structure as a member of the C++ wrapper class, then rewrite the functions to pass a pointer instead of an integer argument, and use pointer dereferencing (`mystructptr->field`) instead of structure dereferencing (`mystruct.field`).

Using Macros to Avoid Cross-Driver Namespace Pollution

To avoid confusion, you should always name your functions in ways that prevent namespace pollution. It is not always practical to name functions with names like `org_mklinux_iokit_swim3_foo`, though, as this quickly becomes unreadable. Instead, a more sane solution is to use macros to rename the functions at compile time.

For example, say you have functions called `bingo`, `nameo`, and `dog`. You want to rename them with the prefix `farmer_had_a_`. You might include macros in a project-wide header file that look like this:

```

#define dog(a, b, c) farmer_had_a_dog(a, b, c)
#define bingo(a, b, c) farmer_had_a_bingo(a, b, c)
#define nameo(a, b, c) farmer_had_a_nameo(a, b, c)

```

Now there is some risk that you will forget to include these defines for a function. There is a simple fix for that problem, though. First, instead of just including the macros, intersperse the prototypes for the functions in the same file. For example:

```

void farmer_had_a_dog(int and, int bingo, void *was_his, char *nameo);
#define dog(a, b, c) farmer_had_a_dog(a, b, c)

void farmer_had_a_bingo(int clap_i_n_g_o);
#define bingo(a, b, c) farmer_had_a_bingo(a, b, c)

void farmer_had_a_nameo(int dog);
#define nameo(a, b, c) farmer_had_a_nameo(a, b, c)

```

Now this is only a partial solution. To make it a complete solution, add the compiler flag `-Wmissing-prototypes` to the gcc compiler options (CFLAGS) in your driver's Makefile or in its Project Builder plist. If you see notices of missing prototypes, you'll know that you forgot one (or more).

For more information about namespace pollution, see *Kernel Programming Guide* in Apple's Developer Documentation.

C++ Language Considerations

Because your driver will probably be written as a combination of C and C++ code, there are some potential pitfalls that you should try to avoid. Many of these are described in this chapter.

The I/O Kit as a whole is written in a subset of C++. The following features are not allowed:

- exceptions
- templates
- multiple inheritance
- non-trivial constructors
(memory allocation and similar should be done in the `init` function)
- standard run-time type identification (RTTI)
(RTTI-like functionality provided with `OSDynamicCast`)
- initialization lists

For more information, see *I/O Kit Fundamentals* and the Object Creation and Destruction section of *I/O Kit Device Driver Design Guidelines*.

Data Type Size Differences

One of the most common errors when porting a C driver core to the I/O Kit is assuming that a data type in C will have the same size as a data type with a similar name in C++. This is not always the case.

A good rule is to always make sure you use *exactly* the same data type (with the same name) on both sides when passing data between the C and C++ portions of your driver. This can save you lots of headaches and unnecessary debugging in the future.

Handling C to C++ Callbacks

C code cannot call class methods directly. This poses a bit of a problem if you are writing code where a C++ method needs to be passed as a callback into your C core code.

To get around this issue, you should create an intermediate relay function within a C++ file. This function is a standard C function, but compiled using the C++ compiler. This function can then be called by C functions and easily make calls into C++ class methods.

The following rules must be followed:

- The C++ function must be declared `static`.
- One of its argument must be a pointer to an instance of your class.
- Calls to class functions must then be made with respect to that pointer.

For example, you might write a relay function that looks like this:

```
static int org_mklinux_iokit_swim3_driver_dosomething(
    org_mklinux_iokit_swim3_driver *self, int something_arg)
{
    dIOLog(DEBUG_GENERAL, "dosomething: enter (0x%x)\n",
        (unsigned int)self);

    self->dosomething(something_arg);
}
```

For your callback, you would need to pass two things into the C core code: the address of the class instance (the value of the `this` pointer), and the address of this function. After that, the C code can call the class function `dosomething` by doing something like this:

```
ret = (*funcptr)(classptr, argument);
```

where `funcptr` is a pointer to the relay function, `classptr` is a pointer to the class instance whose `dosomething` method is to be called, and `argument` is the argument to be passed to the `dosomething` method.

Document Revision History

This table describes the changes to *Porting Drivers to Mac OS X*.

Date	Notes
2009-05-06	Corrected bugs in code samples.
2006-10-03	Added header information for integer types.
2004-04-01	New document that helps UNIX/Linux device driver developers bring their drivers to Mac OS X.
	Initial Revision (Preliminary)

REVISION HISTORY

Document Revision History