Hacking the Mac OS X Kernel for Unsupported Machines

Ryan Rempel[1]
Other World Computing[2]
osxguru@macsales.com

Abstract

To get Mac OS X running on machines that Apple no longer supports, some changes to the behavior of the kernel and certain IOKit kernel extensions were required. Since most of the relevant code is open source, it would have been possible to substitute custom components for the ones that Apple supplies. However, this would have required recompilation and redistribution for every minor version change in Mac OS X. Fortunately, the IOKit system provides a sophisticated run-time environment that often allows your own kernel extensions to inherit Apple's code, and fix only the things that need to be fixed. The techniques involved can be useful for anyone doing driver development, even on supported systems.

Overview

When Apple originally released Mac OS X, it would not run on my trusty Power Macintosh 7300. The "public beta" had been able to run on the 7300.[3] Sadly, the necessary drivers had been removed for the official release of Mac OS X. However, Apple had included those drivers in its open-source initiative (the Darwin operating system). Thus, it seemed as though it might be possible to resurrect Mac OS X on some unsupported systems by updating the drivers.

It turned out that updating the missing drivers was not quite enough. There were additional problems in the Mac OS X kernel and kernel extensions which had to be worked around in one way or another. The techniques used in doing so provide useful case studies in taking advantage of the sophisticated Mac OS X device driver system (the IOKit system).

In some cases, it is possible to "out-match" Apple's drivers — even drivers built into the kernel — with your own. Sometimes you can even use the Apple driver as a superclass, so that you need only implement

[1] *About the Author*: Ryan Rempel is a lawyer who taught himself kernel programming in order to get Mac OS X running on his 7300. Being a graduate student when Mac OS X was released, he had more time to write device drivers than money to buy a new computer. The 7300 is now 8 years old and is still his primary workstation. (It has had a few upgrades in the meantime).

[2] I would like to thank Other World Computing (http://eshop.macsales.com) for their generous financial support over a period of several years as I have worked to keep Mac OS X running on unsupported systems — it is a tribute to their dedication to their customers.

[3] So long as a G3 or G4 CPU upgrade had been installed.

a few methods to fix the problem. This is possible because of certain conventions which are generally used when writing Mac OS X device drivers. Even in cases where the conventions are not perfectly followed, it is often possible to find an appropriate place to intervene in the device driver system in order to solve a problem.

Out-matching Apple's Drivers: The case of the unreadable NVRAM

In the original release of Mac OS X, there was a bug in the routines which read and write NVRAM. In fact, this bug persists to this day. See if you can spot it:

```
IOReturn AppleNVRAM::read (IOByteCount offset, UInt8 *buffer,
            IOByteCount length)
{
  …

  case kNVRAMTypePort:
    for (cnt = 0; cnt < length; cnt++) {
      *_nvramPort = (offset + length) >> 5;
      eieio();
      buffer[cnt] = _nvramData[((offset + length) & 0x1F) << 4];
    }
    break;

  …
}
```

Listing 1. Extract from <xnu/iokit/Drivers/platform/drvAppleNVRAM/AppleNVRAM.cpp>

Like many kernel bugs (in my experience), it is a problem which is reasonably easy to see once you know exactly which part of the code is causing the problem.[4] Of course, it often takes quite a while to get to that point!

The effect of this problem is that reading and writing NVRAM would not work on the 7300 – 9600 series. That, in turn, explains why Apple's engineers did not catch the problem: since the 7300 – 9600 were no longer supported, they were no longer testing on those machines.[5] So it was something that would need to be fixed in order to run Mac OS X on the 7300.

If a similar problem were found in Linux or FreeBSD, the procedure to fix it would be reasonably clear:

- Develop and test the bug fix (in this case, merely a two-line change).
- Produce a patch against the official source code.
- Submit it to the project.
- Bug the maintainers until it is incorporated in a new release.

For Mac OS X, things are a little different. It is, of course, possible to submit patches to Apple, and Apple will sometimes incorporate them. However, this is less likely where the problem only affects unsupported

[4] "`offset + length`" should, of course, be "`offset + count`".

[5] People sometimes ask me if I have any insight into why Apple makes the decisions it does about model support. One can, of course, speculate about certain marketing considerations (Apple does have interest in selling new machines, and a degree of uniformity among the supported models does simplify marketing claims regarding new features which are hardware dependent). From a purely technical point of view, the main advantage in dropping model support would appear to be that it limits the number of configurations which need to be tested.

machines (the word "unsupported" does mean something, after all). Even if Apple did accept a patch, it would not entirely solve the problem. The bug would still be present on all the existing installation CD's, and it wouldn't be possible for people to simply download a new CD (the way they might download a new Linux or FreeBSD CD).[6] So the problem needs to be fixed independently, without relying on a new release of the operating system as a whole.

Since the Mac OS X kernel is (mostly) open source, it would be possible to recompile and redistribute a version of the kernel that fixed the problem. However, this would be awkward — every time Apple upgraded the kernel (even a minor upgrade), you would have to redistribute a corresponding "patched" kernel. It would be much better to solve the problem dynamically, without requiring any changes to the Apple-supplied kernel. Fortunately, Mac OS X has a mechanism for adding code to the kernel dynamically — we can write kernel extensions.

The notion of a kernel extension is by no means unique to Mac OS X. Other operating systems have similar mechanisms. What is unique to Mac OS X is the ease with which you can arrange for the kernel to use your kernel extension to fix problems in the underlying operating system. The key is that Mac OS X is unusually introspective about its driver model. Drivers are objects with a common superclass (IOService), and cooperate in a matching process to determine which driver will control which device.

At boot time, the kernel constructs a "device tree", which contains information about the various devices located either on the motherboard or in PCI slots. The type of information available differs to some extent depending on the device in question, but generally includes such things as a name, a model (or "compatible" model), a device type, information about interrupts and memory locations, and other things which a driver needs to know. The devices are organized in a tree structure, based on the way in which they are connected physically (or sometimes logically).

The kernel then traverses each device in the device tree and figures out which driver ought to be used to control that device. This process builds a registry structure in the kernel which records which drivers are attached to which devices. The pattern is that a driver attaches to a device, and then the driver publishes nubs which represent the devices "below" its device in the tree structure. The kernel then traverses those devices, choosing a driver for them, which then continues the process. One ends up with a tree structure with alternating devices and drivers.

So, to fix the problem reading and writing NVRAM, I needed to write a new NVRAM driver, and then ensure that it was selected by the kernel in preference to the NVRAM driver that was built into the kernel.

The first task was to figure out where in the device tree the NVRAM driver occurs. We saw earlier that the C++ class[7] in question was `AppleNVRAM`. You can explore the device tree with the IORegistryExplorer, or with the ioreg command in the terminal, to find where this fits in. The following output from the ioreg command shows where the nvram "device" and its driver fit into the registry scheme.[8]

[6] Of course, Apple distributes bug fixes through the "System Update" mechanism, but that doesn't help if the problem prevents booting from the Install CD in the first place.

[7] Device drivers in Mac OS X are written in a somewhat restricted dialect of C++ (no multiple inheritance, no templates, etc.)

[8] I have edited the output to eliminate other devices, and to show properties only for NVRAM and AppleNVRAM.

```
+-o Root  <class IORegistryEntry>
  +-o AAPL,7500  <class IOPlatformExpertDevice>
    +-o ApplePowerSurgePE  <class ApplePowerSurgePE>
    | +-o bandit@F2000000  <class IOPlatformDevice>
    | | +-o AppleMacRiscPCI  <class AppleMacRiscPCI>
    | |   +-o gc@10  <class IOPCIDevice>
    | |    | +-o AppleGrandCentral  <class AppleGrandCentral>
    | |    |   +-o nvram@1D000  <class AppleGrandCentralDevice>
    | |    |    | | {
    | |    |    | |   "IODeviceMemory" = …
    | |    |    | |   "reg" = <0001d000000000100001f00000000200>
    | |    |    | |   "name" = <"nvram">
    | |    |    | |   "existing" = <0000000000002000>
    | |    |    | |   "device_type" = <"nvram">
    | |    |    | |   "AAPL,phandle" = <ff83e168>
    | |    |    | | }
    | |    |    | |
    | |    |    | +-o AppleNVRAM  <class AppleNVRAM>
    | |    |    |     {
    | |    |    |       "IOClass" = "AppleNVRAM"
    | |    |    |       "IOProviderClass" = "AppleMacIODevice"
    | |    |    |       "IONameMatched" = "nvram"
    | |    |    |       "IONameMatch" = ("nvram")
    | |    |    |     }
```

Listing 2. Partial output from ioreg -S -l

You can see the alternation between the representation of devices and drivers. The `IOPlatformExpertDevice` class represents the machine as a whole. The `ApplePowerSurgePE` class is then the driver which was selected for that device (it is known as the "platform expert" and is responsible for certain types of overall coordination among the drivers). The `ApplePowerSurgePE` class then creates the "bandit" `IOPlatformDevice` (among others not shown here). The `AppleMacRiscPCI` class is then selected as the driver for "bandit", and so on.

Ultimately, we arrive at the "nvram" device, which has properties which indicate its name, device_type, and information about how to access the device in memory. The `AppleNVRAM` class has been selected as the driver. So what I needed to do was write a kernel extension with a fixed version of the class, and then set things up so that my class would be selected in the boot process instead of the `AppleNVRAM` class built into the kernel.

Writing the code for the `PatchedAppleNVRAM` class (as I called it) was an easy exercise. The `AppleNVRAM` class is short and simple, so I merely duplicated the code[9] and fixed the two lines that needed to be fixed.[10] Now, how to get the kernel to use my driver?

The kernel begins the process of selecting drivers by collecting information about all the drivers which are available to it (either built into the kernel itself, or in kernel extensions).[11] In kernel extensions, this information is set out in the "Info.plist" file. For drivers built into the kernel, you can see the information

[9] Making use of the appropriate copyright attributions, I hasten to add!

[10] The current version of `PatchedAppleNVRAM` actually does a little more than this now, but originally it was just the two-line change.

[11] Originally, a fair number of drivers were built into the kernel itself. However, subsequent releases of Mac OS X have moved virtually all of the drivers into kernel extensions instead. This saves some kernel memory, since only those kernel extensions which are needed for a particular model are loaded into memory.

in the kernel source code, or you can deduce it from the properties you see when running the ioreg command. In the case of `AppleNVRAM`, you can see the key properties in the listing above: IOClass, IOProviderClass, and IONameMatch.

The kernel's process for selecting drivers has a "passive" and an "active" component. In the passive component, the kernel first looks at the class of the object which represents the device. In this case, the class of the object representing the NVRAM device is `AppleGrandCentralDevice`. This is the "nub" that was published by the `AppleGrandCentral` driver. Drivers will publish nubs subclassed (often from `AppleMacIODevice`) to include any particular quirks needed to represent its devices. The IOProviderClass property in the driver then represents the kind of nubs which that driver might attach to — the kernel will only consider nubs with the specified class (or a subclass). The kernel will then apply various further matching schemes which vary somewhat depending on the class of the nub. In this case, the `AppleNVRAM` driver has an IONameMatch property, which matches nubs which have a name (or model, or compatible) property of "nvram". So, that is how the kernel selects the `AppleNVRAM` class as a possible driver for the nvram device.

Once the kernel has identified all the possible drivers for a device, the matching process then enters an "active" stage. The kernel calls the "probe" method in each potential driver in turn, passing a reference to the device which is being matched. The driver can then do some investigation of the device to see whether it is really the kind of device which the driver can deal with. The driver can alter what is known as the "probe score". If more than one driver thinks that it can handle a device, then the kernel picks the one which has assigned itself the highest probe score.

So getting the kernel to select my `PatchedAppleNVRAM` driver in preference to the `AppleNVRAM` driver built into the kernel really isn't very hard. All I had to do was set up the appropriate part of my Info.plist to look something like this:

```
<key>IOKitPersonalities</key>
<dict>
  <key>PatchedAppleNVRAM</key>
  <dict>
    <key>CFBundleIdentifier</key>
    <string>oldworld.support.PatchedAppleNVRAM</string>
    <key>IOClass</key>
    <string>PatchedAppleNVRAM</string>
    <key>IONameMatch</key>
    <array>
      <string>nvram</string>
    </array>
    <key>IOProbeScore</key>
    <integer>200</integer>
    <key>IOProviderClass</key>
    <string>AppleMacIODevice</string>
  </dict>
</dict>
```

Listing 3. Partial contents of Info.plist for PatchedAppleNVRAM.kext

Note the IOProbeScore property. If all you need to do is adjust the probe score, then you can do it in the Info.plist file, rather than writing a method in the driver. So this Info.plist file tells the kernel to match my driver with the NVRAM device, and since its IOProbeScore is higher than the probe score for the `AppleNVRAM` driver built into the kernel, my driver wins!

There is an additional property required in the Info.plist file — the OSBundleRequired property must be set to "Root". The reason is that the driver matching process occurs in two stages as Mac OS X boots. First, the bootloader (called BootX[12]) loads the kernel and any kernel extensions which have identified themselves as being needed during the early part of the boot process. BootX then launches the kernel, and the kernel does the first round of driver matching in order to be able to mount the root-device. The startup process then continues according to the startup scripts on the root-device. This includes a second round of driver matching, which considers all the kernel extensions. The OSBundleRequired property is how the `PatchedAppleNVRAM` kernel extension tells BootX that it needs to be loaded for the initial round of driver matching. If it were not loaded at that stage, then the built-in `AppleNVRAM` driver would be selected, and `PatchedAppleNVRAM` would never have a chance.

The cool thing about this is how easy it is to do. In fact, it is an entirely orthodox application of the underlying principles of the Mac OS X driver system. No trickery or hackery required![13] Furthermore, Apple did not do anything particularly unusual in the `AppleNVRAM` driver itself to make it possible for it to be side-stepped in this manner. Instead, they simply coded the `AppleNVRAM` driver according to the usual conventions of the IOKit driver system. Merely by following convention, it became possible to supply a kernel extension that would out-match the original code in the kernel.

Fiddling with the Boot Process: Booting from an unmodified Install CD

Along with such bug fixes as the PatchedNVRAM.kext, there are also various drivers which needed to be updated (or, in some cases, created) for the unsupported machines. In total, there are about 30 kernel extensions involved, with approximately 45,000 lines of code.

Having written (or updated) kernel extensions to provide the necessary drivers (or fix bugs) for an unsupported machine, there was still a practical problem remaining. You install Mac OS X by booting from the Install CD, but my unsupported drivers were not on the CD. Thus, an attempt to boot from the Install CD in the usual way would still result in a kernel panic (since the kernel cannot find a "platform expert" driver to match the `ApplePlatformExpertDevice` which represents the root of the device tree).

There are a variety of ways that one might deal with this. One option would be to do the install on a supported machine, add the needed drivers, and then transfer the hard drive to an unsupported machine. This can work, because an installation of Mac OS X does not vary depending on the machine used to do the installation.[14] However, this method would be cumbersome, and I did not want to assume that people would have a supported machine to use for this purpose.

Originally, my instructions for installing Mac OS X on an unsupported machine involved copying the Install CD to a partition on your hard drive, modifying that partition, and then booting from it in order to perform the install. It worked, but it was a cumbersome 14-step process, and not everyone has a separate partition handy.

To come up with something a little friendlier, some research was required into the Mac OS X boot process. For these purposes, the Mac OS X boot process can be divided into four parts.

[12] This is a Mac OS X bootloader, not to be confused with the LinuxPPC bootloader of the same name.
[13] Well, perhaps a little bit of trickery — see the next section.
[14] There are some small exceptions, mostly relating to DVD support.

- Open Firmware loads patches from NVRAM (if any).[15]
- Open Firmware loads BootX.
- BootX loads the kernel and the kernel extensions required to access the root device.
- The kernel mounts the root device and initiates the startup process from that device.

Ultimately, what we want is for the kernel to use the Install CD as the root device, since that is what is going to actually get the install process going. We also want to have BootX load the kernel and kernel extensions from the CD. However, we want it to load some extra kernel extensions that are not on the CD.

It turns out that three are three variables in NVRAM which can be used to control each part of the boot process separately. The "boot-device" variable controls which drive Open Firmware will try to load BootX from.[16] By default, BootX will then load the kernel and kernel extensions from the boot-device. However, if you specify a "boot-file", then BootX will load the kernel and kernel extensions from there (instead of the boot-device). The kernel will then, by default, use the partition the boot-file was on as the root device. However, that can be adjusted as well, by specifying a different root device in the "boot-command" variable.[17] Thus, the various elements of the boot process can be fine-tuned according to our needs.

This made it possible to set things up in the following way. I would copy the kernel and kernel extensions from the Install CD to a temporary location on the target partition (that is, the partition the user wanted to install Mac OS X on). I would then add the unsupported kernel extensions to the extensions copied from the CD. I would set the "boot-device" variable to point to the Install CD, since it had BootX installed in the required manner. However, the "boot-file" variable would be set to the temporary location of the kernel and kernel extensions (which had been copied to the target volume). Finally, the "boot-command" variable set the root device back to the CD. That way, the boot process would start from the CD, pick up the kernel and kernel extensions copied to the target volume (including the additional ones), and then return to the CD to continue as the root device.

By this point, I had written an application to manage this process (which ultimately became XPostFacto). It only required three clicks (select the target volume, select the Install CD, and click the "Install" button) — everything else was now automated.[18] The application manages such things as the conversion between volumes and Open Firmware paths,[19] reading and writing NVRAM, presenting the user interface and

[15] The main purpose of these patches is to fix bugs in the Open Firmware implementation on the unsupported machines. The "Old World" machines made limited use of Open Firmware prior to Mac OS X, and thus certain bugs were not particularly significant. The Mac OS X boot process makes greater use of Open Firmware and thus needs those bugs to be fixed. Fortunately, the required NVRAM patches (written in the Forth programming language) had been mostly completed by Apple, and made open-source as part of the Darwin project. I did tweak them in certain respects to conserve NVRAM space (which is limited on "Old World" machines).

[16] On "New World" machines, you can point the boot-device to a specific BootX file. On "Old World" machines, you can only point to a specific partition, and even then Open Firmware will actually use the first bootable partition it finds on the drive, rather than the one you specify. Also, on "Old World" machines there are special techniques involved in installing BootX (you need to modify the partition table in certain ways to tell Open Firmware how to load BootX).

[17] On "New World" machines, you would specify the root device in the "boot-args" rather than the "boot-command".

[18] Just to provide a sense of the scale of the XPostFacto application, it is about 24,000 lines of code. It is written using the MacApp framework, which is now itself unsupported by Apple!

[19] The NVRAM variables which control the boot process are specified in terms of Open Firmware paths, and it turns out that there is no trivial means to accurately determine the Open Firmware path of a mounted volume from Mac OS 9. Thus, it was necessary to work out ways of determining Open Firmware paths in different scenarios (SCSI, ATA, Firewire, PCI cards, etc.).

managing user preferences, installing and updating the kernel extensions, etc. Initially it only ran in Mac OS 9 (in order to set up an initial Mac OS X installation), but I eventually ported it to run in Mac OS X as well (in order to manage updates in the kernel extensions and updates from one version of Mac OS X to the next).

Copying the kernel and kernel extensions from the Install CD worked reasonably well until Apple stopped putting the kernel extensions on the Install CD. Instead, the Install CD relied purely on an extensions cache for booting (the Extensions.mkext), and installed the actual kernel extensions from an archive. Of course, I could copy the extensions cache to the target volume and point the "boot-file" there. However, BootX would not load both the extensions cache and the additional unsupported extensions. It would either use the cache or the extensions themselves, but not both — otherwise, what would be the purpose of having a cache?

Fortunately, BootX itself was open source, so if it did not do what I wanted, I could modify it to suit my purposes better. So, I compiled (and installed) a custom version of BootX which would recognize when an unsupported installation was being attempted, and in those cases load both the extensions cache copied from the CD as well as the additional unsupported kernel extensions. This meant that the boot-device would now be the target volume (to pick up my custom BootX). Otherwise, the boot settings remained similar to the previous scheme.[20]

Thus, the flexibility of the Mac OS X boot process made it possible to boot an unsupported system from an unmodified Mac OS X Install CD.

Subclassing Apple's Drivers: The case of the crashing CD-ROM

Another difficulty with the original release of Mac OS X on unsupported systems was that whenever you inserted a CD-ROM into a SCSI CD-ROM device, the system would panic.[21] This made the install process a little difficult — BootX would load the kernel and start the boot process, but as soon as the kernel tried to mount the Install CD, boom!

It turned out that the problem was in the `IOSCSICDDrive` class, specifically the `readTOC` method. The method took a buffer pointer as an argument, and then stored the pointer in a structure which was ultimately released. The problem was that the buffer was released as well, but the `readTOC` method hadn't bumped its retain count to deal with that. So it was a basic memory management problem. The solution was to bump the buffer's retain count before using it (this time, a one-line fix).

As with the `AppleNVRAM` example, it would have been possible to create a "corrected" version of the class, and then manipulate the Info.plist so that the kernel would select the corrected class rather than the one built into the kernel. However, in this case there would have been difficulties with simply out-matching the Apple driver. With `AppleNVRAM`, it wasn't really necessary to worry about future

[20] Now that I write this, it occurs to me that it would have been possible to dispense with copying the kernel and kernel extensions from the Install CD at this point. I could have programmed the custom BootX to load the kernel and kernel extensions directly from the CD, and then also load the additional extensions from the target volume. I do not believe that approach occurred to me at the time. However, there are some advantages to copying the kernel and kernel extensions from the CD. There are cases in which BootX cannot read from the CD (using the Open Firmware drivers) whereas the kernel can (using the Mac OS X drivers). Copying the kernel and kernel extensions is required in such a case. It also helps when there are no Open Firmware drivers for the installation drive (a Firewire DVD on an unsupported machine, for instance).

[21] Once again, the reason Apple's engineers didn't catch this one right away is no doubt that virtually all the supported machines used IDE CD-ROM devices.

modifications that Apple might make to the original class. `AppleNVRAM` was a very short class that did just one thing, and possibility that it would be changed significantly by Apple seemed remote. Thus, it was reasonably prudent to just ignore the original version and use the corrected version.

The `IOSCSICDDrive` class, on the other hand, was more complex. It could well change at some point, which would require that any substitute class be recompiled and redistributed to take the changes into account. Thus, the ideal solution would be to replace not the whole class, but just the one method that was causing problems.

Fortunately, we don't have to write drivers from scratch — it is often possible to subclass and inherit from Apple's drivers. Here is what the header for the subclass looked like (with some code removed which related to a different bug):

```
#include "IOSCSICDDrive.h"
#include "IOSCSICDDriveNub.h"

class PatchedIOSCSICDDrive : public IOSCSICDDrive {

    OSDeclareDefaultStructors(PatchedIOSCSICDDrive)

public:

    virtual bool deviceTypeMatches(UInt8 inqBuf[],UInt32
            inqLen,SInt32 *score);

    virtual IOReturn readTOC(IOMemoryDescriptor * buffer);

};
```

Listing 4. Partial contents of PatchedIOSCSICDDrive.h

Note that we include the header for the superclass and then declare the replacement as a subclass of it.[22] In some cases, the only method we need to implement is the one which is causing a problem (i.e., the `readTOC` method). In this case, I also implemented the `deviceTypeMatches` method.

Here is the corresponding implementation (again, omitting some code which dealt with a different bug):

```
IOReturn
PatchedIOSCSICDDrive::readTOC (IOMemoryDescriptor *buffer)
{
  // This works around a bug that may be present in our
  // superclass (it may release buffer without retaining it).
  // But the bug may be fixed, so we check for that.

  buffer->retain();
  int bufferRetainCount = buffer->getRetainCount();
  IOReturn result = super::readTOC(buffer);
  if (buffer->getRetainCount() == bufferRetainCount) {
    buffer->release();
  }
  return result;
}
```

[22] In this case, it was necessary to copy the superclass header from the Darwin source code. In cases where Apple anticipates that subclassing will be necessary, you can find headers in the Kernel.framework.

```
bool
PatchedIOSCSICDDrive::deviceTypeMatches (UInt8 inqBuf[], UInt32
            inqLen, SInt32 *score)
{
  bool retVal = super::deviceTypeMatches (inqBuf, inqLen, score);
  if (retVal) *score = 20000;
  return retVal;
}
```

Listing 5. Partial contents of PatchedIOSCSICDDrive.cpp

In this case, instead of setting the IOProbeScore in the Info.plist, I implemented the `deviceTypeMatches` method (which is called by the `probe` method in the superclass). The effect is to out-match the Apple driver (i.e. the superclass itself) when the kernel is selecting drivers.[23]

Note that the `readTOC` method in the subclass does not actually re-implement the whole of the `readTOC` method in the superclass. The reason this works is that the subclass has access to the buffer pointer being passed in as an argument, and can thus bump the retain count of the buffer before calling the `readTOC` method in the superclass. The code then checks whether `super::readTOC` has changed the retain count. If not, then the code releases the buffer — otherwise, our bug fix would actually become a bug once the superclass was fixed.

The net effect is that it was possible fix the bug in a few lines of code, by subclassing from Apple's driver and "wrapping" the problematic method with some error-checking.

Dealing with Families: The case of the disappearing SCSI devices

An analogous problem affected the initial release of Mac OS X 10.3. Sometimes (but not always) SCSI devices would not appear when the machine was started up. Starting up in verbose mode seemed to improve the situation, so it appeared as though it might be affected by timing issues in the boot process.

The problem turned out to be an interaction between the `IOBlockStorageDriver` class and the `IOBlockStorageServices` class. To understand it, we need to delve further into the structure of the device registry which the kernel creates as it is matching devices with drivers. When we looked at the `AppleNVRAM` bug, we noted that there was an nvram device published to the device registry, and then a driver selected to match it. This conformed to the general convention of alternating between devices and drivers in the levels of the registry. The reason this is a useful convention is that it permits a kind of loose coupling between devices and drivers. In general, it is not a good idea for one driver to directly create an instance of another driver. Instead, it should publish a device, and let the kernel figure out what driver is appropriate for that device. These "devices" do not necessarily always correspond to hardware as such — in some cases, they are merely pseudo-devices, for the express purpose of achieving a loose coupling between drivers.

Apple makes particularly extensive use of these conventions in the drivers that form part of the storage-related families. Consider the registry structures created for the internal SCSI bus on the 7300:[24]

[23] It would probably have been possible to achieve the same effect by setting the IOProbeScore in the Info.plist — I am not sure why I didn't do that here.

[24] I have edited the output to fit better on the page, in cases where the class name is the same as the entry name.

```
+-o mesh@18000  <class AppleGrandCentralDevice>
  +-o meshSCSIController  <class meshSCSIController>
    +-o IOSCSIParallelDevice@0  <class IOSCSIParallelDevice>
      +-o IOSCSIParallelInterfaceProtocolTransport
         +-o IOSCSIPeripheralDeviceNub
           +-o IOSCSIPeripheralDeviceType00
              +-o IOBlockStorageServices
                +-o IOBlockStorageDriver
                  +-o IBM DDRS-39130W Media  <class IOMedia>
                    +-o IOMediaBSDClient
                    +-o IOApplePartitionScheme
                      +-o Apple@1  <class IOMedia>
                      | +-o IOMediaBSDClient
                      +-o Macintosh@2  <class IOMedia>
                      | +-o IOMediaBSDClient
                      +-o Macintosh@3  <class IOMedia>
                      | +-o IOMediaBSDClient
                      +-o Macintosh@4  <class IOMedia>
                      | +-o IOMediaBSDClient
                      +-o Patch Partition@5  <class IOMedia>
                      | +-o IOMediaBSDClient
                      +-o untitled@6  <class IOMedia>
                        +-o IOMediaBSDClient
```

Listing 6. Partial ioreg output related to SCSI drive

The entry named "mesh" represents the SCSI controller on the motherboard. The `meshSCSIController` is the class that knows how to talk to that device. The `IOSCSIParallelDevice` represents the SCSI device at ID 0. One of the jobs of the `meshSCSIController` driver is to create `IOSCSIParallelDevice` nubs for any devices it finds on the SCSI bus. But `meshSCSIController` doesn't create a driver for those nubs — that is the kernel's job. Generally speaking, each driver class only needs to know about the device above it (its provider) and the devices below it (its children). It does not need to know about the other drivers in the chain.

In the case of the disappearing SCSI drives, the registry structures noted above would get cut off at the `IOBlockStorageDriver` — the `IOMedia` classes would never arrive. Examining the `IOBlockStorageDriver` code, there appeared to be two paths which would lead to it creating `IOMedia` nubs. There was a `message` method which objects higher in the chain could call to tell the `IOBlockStorageDriver` object that media had arrived or departed (consider a removable ZIP drive, for instance). That would, however, not catch cases where the media had arrived before the `IOBlockStorageDriver` was created.[25] To deal with that case, the `start` method in `IOBlockStorageDriver` called `checkForMedia`, which in turn asked the provider to `reportMediaState`. If the media state was reported to have changed, then the code would deal with that by creating the `IOMedia` nub.

[25] The Mac OS X kernel is multi-threaded, so it would be possible for the registry structures to be created while the SCSI controller was still talking to the SCSI device.

The `checkForMedia` method in `IOBlockStorageDriver` looked like this:

```
IOReturn
IOBlockStorageDriver::checkForMedia(void)
{
    IOReturn result;
    bool currentState;
    bool changed;

    IOLockLock(_mediaStateLock);

    result = getProvider()->
            reportMediaState(&currentState,&changed);

    …

    if (changed) {
        result = mediaStateHasChanged (currentState ?
            kIOMediaStateOnline : kIOMediaStateOffline);
    }

    IOLockUnlock(_mediaStateLock);
    return(result);
}
```

Listing 7. IOBlockStorageDriver::checkForMedia

The implementation of `reportMediaState` in `IOBlockStorageServices` looked like this:

```
IOReturn
IOBlockStorageServices::reportMediaState ( bool * mediaPresent,
            bool * changed )
{
    *mediaPresent       = fMediaPresent;
    *changed            = false;

    return kIOReturnSuccess;
}
```

Listing 8. IOBlockStorageServices::reportMediaState

This is another case where the problem is pretty obvious once you juxtapose the correct bits of code. The `reportMediaState` method does not actually keep track of whether the media state has changed or not — it always returns false. But that means that the `checkMediaState` method never does anything useful, because it only does something if `reportMediaState` says that the media state has changed. Thus, the `IOBlockStorageDriver` class could not properly respond to media that was detected before the `IOBlockStorageDriver` object was created. It could only successfully respond to the `message` method afterwards (since that code path did not involve `checkMediaState`). That was why the problem was sensitive to timing issues in the boot process — if the detection of the media was delayed slightly until after the `IOBlockStorageDriver` object was set up, then the SCSI devices would appear.[26]

[26] This illustrates one of the occasional frustrations created by the Mac OS X approach to device drivers — there are so many different objects involved that it is sometimes difficult to keep track of exactly who is responsible for what.

To fix this, one could either adjust the `IOBlockStorageDriver` class or the `IOBlockStorageServices` class. I chose to do the former, subclassing from `IOBlockStorageDriver` and implementing a fixed version of the `checkMediaState` method (I made it compare the reported media state with its own previously recorded state, rather than relying on the value returned in `changed`).

It turned out that Apple's engineers had figured out the problem as well, and were implementing a solution in Mac OS X 10.3.4. However, they did it at the other end. Instead of making `checkMediaState` keep track of `changed`, they fixed `IOBlockStorageServices` so that it would return the correct value.

This ended up clashing with the way that I had fixed things, so I had to change my kernel extension to determine whether the Apple fix had been installed or not. It would probably have been possible to check whether 10.3.4 had been installed globally, but instead I decided to check the version of the specific kernel extension which contained `IOBlockStorageServices`. The code looked like this:

```
IOService*
PatchedBlockStorageDriver::probe (IOService *provider, SInt32
            *score)
{
    kmod_info_t *scsiAMF = kmod_lookupbyname
            ("com.apple.iokit.IOSCSIArchitectureModelFamily");
    if (!scsiAMF) return NULL;

    UInt32 targetVersion, installedVersion;
    VERS_parse_string ("1.3.3", &targetVersion);
    VERS_parse_string (scsiAMF->version, &installedVersion);
    if (installedVersion >= targetVersion) return NULL;

    return super::probe (provider, score);
}
```

Listing 9. Code to check installed version of kernel extension[27]

This way, we could again get out of the way once Apple had fixed the underlying problem.

There was one more wrinkle in the re-implementation of `checkForMedia`. Note that the original code used a lock variable (`_mediaStateLock`) to serialize `checkForMedia`. If the lock variable were only used in `checkForMedia`, then I could have created my own in the subclass. However, it was also used elsewhere, so it seemed that it might be important to use the same lock variable that the superclass used. The problem was that `_mediaStateLock` was declared as a private variable in the header. Thus my subclass could not access it.

To solve this, I made a copy of the header file, and edited it to make the `_mediaStateLock` protected rather than private. The compiler now believed that my subclass could access it. It appears that the

---

In fact, the `IOBlockStorageDriver` class and the `IOBlockStorageServices` class are in two different kernel extensions (IOStorageFamily.kext and IOSCSIArchitectureModelFamily.kext).

[27] I have omitted the code for `VERS_parse_string`. The full code can be found at this URL: <http://cvs.opendarwin.org/index.cgi/projects/XPostFacto/Extensions/PatchedBlockStorageDriver/PatchedBlockStorageDriver.cpp?rev=1.3>

public/protected/private classification does not affect the code at run-time. Thus, it seems possible to "work around" that issue by editing headers where necessary. (At least, it worked in this case).[28]

Dealing with Mac OS X Upgrades: More kernel magic

The big advantage of being able to subclass and out-match the original driver is that the resulting kernel extension can continue to work as Apple updates Mac OS X. Since we are merely adding kernel extensions, rather than modifying what Apple installs, it doesn't matter when the user updates the kernel and kernel extensions to a new version of Mac OS X — our kernel extensions just keep working.

The fact that kernel extensions generally keep working across major Mac OS X updates (i.e. 10.2 to 10.3 etc.) actually requires a certain amount of kernel magic. Ordinarily, a driver written in C++ would be very sensitive to changes in its superclass. It is quite possible (and sometimes necessary) to make changes in the storage requirements of the superclass (i.e. new member variables) or its methods (especially virtual methods) that would require subclasses to be recompiled. This would be awkward for drivers, since one would have to redistribute and reinstall different versions of the drivers based on the version of Mac OS X installed.

Fortunately, Mac OS X uses some conventions and some kernel magic in order to ease the problem. By convention, many IOKit drivers include an "expansion" variable, which is a pointer to a structure that the driver creates at run-time. This allows a superclass to add variables to the expansion structure without changing the size of the class itself. Many IOKit drivers also include unused virtual methods. This allows new virtual methods to be added, again without changing the size of the class. The kernel does some magic when linking subclasses in order to adjust them to correspond with an updated superclass (this magic even dealt with changes in the ABI when moving from gcc 2.95 to gcc 3).

Because of these conventions and kernel magic, it is generally possible to maintain a fair degree of forward compatibility for kernel extensions in Mac OS X. For instance, if you compile a kernel extension against the Mac OS X 10.2 headers, it is often possible to have it work with Mac OS X 10.2 through 10.4.[29]

However, there can be some issues to deal with when Apple upgrades Mac OS X. It is often the case that certain kernel methods will have been removed for one reason or another, so that code which relied on those methods will no longer link. In those cases, it is often possible to reimplement that code using other methods so that it can work with both the new version of Mac OS X and the previous version. If not, then it may be necessary to compile two versions of the kernel extension.[30]

One also needs to watch for cases where your code is creating objects with `new` and the size of those objects has changed. When Mac OS X 10.3 was coming out, I was unable to get the developer previews to boot on unsupported machines at all — the machine would panic, and I could not make very much sense out of the backtrace.[31] After Mac OS X 10.3 was released, I realized that the size of the `IOPMrootDomain` class had changed. The problem was that my platform expert was creating one with

[28] Instead of changing `private` to `protected`, another option would have been to add a "`friend`" designation (i.e. `friend PatchedBlockStorageDriver;`) to allow the subclass to access private variables.
[29] In fact, until recently I was compiling the XPostFacto kernel extensions for use with Mac OS X 10.0 through 10.4, but have now dropped support for 10.0 and 10.1 to simplify things a little.
[30] You would also need to compile multiple versions if you want to explicitly use new capabilities in the later headers.
[31] Unfortunately, Apple no longer supplies source code for the kernel or kernel extensions when making developer previews available, which makes those previews of limited assistance for debugging kernel extensions.

`new` (i.e. `new IOPMrootDomain;`). This was, of course, now the wrong size (since the size was fixed at compile-time by the previous headers).

Of course, this created a dilemma — if I compiled against the new headers, I would run successfully on Mac OS X 10.3, but would crash on previous versions of Mac OS X. Fortunately, there is an alternate way to create IOKit objects that does not fix there size at compile-time. Instead of `new IOPMrootDomain`, I could use the following handy invocation:

```
(IOPMrootDomain *) IOPMrootDomain::metaClass->alloc ();
```

Listing 10. Handy way to create IOKit objects without worrying about size changes.

Essentially, this asks the IOPMrootDomain metaclass to create an IOPMrootDomain object. That way, the size of the object can change and it won't matter – the metaclass will always know to create an object of the correct size.

Duck Typing: Pretending to be an Apple class when you can't subclass

One of the advantages of subclassing from an Apple driver is that anything which is expecting to see the Apple driver will generally be happy to deal with a subclass. There are a few circumstances in which it is not possible to subclass. However, there are still some Mac OS X driver conventions that are helpful in that case.

One case in which you cannot subclass is when you do not have a header file for the Apple class. While many of the Mac OS X drivers are open source, not all of them are. One case which I came across is the `ApplePMU` driver. It was originally open source, but by Mac OS X 10.2 it had become closed source. The problem was that the closed source version did not work properly with some of the unsupported machines. Without source code, it was difficult to track down exactly why it didn't work. So I simply created a substitute class (`OpenPMU`) based on the last open-source version of `ApplePMU`. Since I did not have a header file for the closed-source `ApplePMU`, I could not subclass.

The other reason I could not subclass is that the ApplePMU.kext did not have an OSBundleCompatibleVersion entry in its Info.plist. When you load a kernel extension, the kernel needs to link your kernel extension with symbols from the kernel and other kernel extensions. The Info.plist in your kernel extension indicates which symbols it requires, by specifying kernel extensions (or parts of the kernel) in its OSBundleLibraries property. The OSBundleLibraries property specifies particular versions of those kernel extensions (or parts of the kernel) which your kernel extension requires. Of course, the kernel extension you require may have been updated in a way that is backwards-compatible with the version you specified. If so, it will use an OSBundleCompatibleVersion entry to indicate the earliest version that it is still compatible with. However, if there is no OSBundleCompatibleVersion entry for a kernel extension, then the kernel will not link your kernel extension to any of its symbols. Thus, without an OSBundleCompatibleVersion entry in ApplePMU.kext, I could not subclass from `ApplePMU`.

Since I could not subclass, `OpenPMU` instead descends from the same superclass as `ApplePMU`. The potential difficulty is that several other drivers need to talk to the PMU driver. If they are expecting to see an `ApplePMU` object, and they get an `OpenPMU` object, then how will then know how to talk to it?

In some cases, the driver is based on a "family" superclass, and the clients know how to talk to the superclass. That was not the case for ApplePMU, however — there was no family superclass that clients were expecting.

Fortunately, Apple has provided another mechanism to decouple method calling from specific class types where necessary. Every Mac OS X driver understands the `callPlatformFunction` method, because it is defined in the `IOService` class (the superclass for all Mac OS X drivers). The arguments to the `callPlatformFunction` method are an `OSSymbol` (which identifies what to do), and then a bunch of void pointers which provide the parameters. So, as long as `OpenPMU` responds to the same symbols as `ApplePMU`, clients can use `callPlatformFunction` without caring which it is. If it walks like a duck, it is a duck[32] (at least, as long as the clients are all cooperating with the convention).

Here is how `OpenPMU` implements the `callPlatformFunction` method. It may or may not be identical to the current `ApplePMU` implementation — since `ApplePMU` is now closed source, who knows?

```
IOReturn
OpenPMUInterface::callPlatformFunction( const OSSymbol
            *functionName, bool waitForFunction, void *param1,
            void *param2, void *param3, void *param4 )
{
    // Dispatches the right call:
    if (functionName->isEqualTo(kSendMiscCommand)) {
        …
    }
    else if (functionName->isEqualTo(kRegisterForPMUInterrupts))
            {
        …
    }
    else if (functionName->isEqualTo(kDeRegisterClient)) {
        …
    }
    else if (functionName->isEqualTo(kSetLCDPower)) {
        setLCDPower(param1 != NULL);
        return kIOReturnSuccess;
    }
    else if (functionName->isEqualTo(kSetHDPower)) {
        setHDPower(param1 != NULL);
        return kIOReturnSuccess;
    }
    else if (functionName->isEqualTo(kSetMediaBayPower)) {
        setMediaBayPower(param1 != NULL);
        return kIOReturnSuccess;
    }
    else if (functionName->isEqualTo(kSetIRPower)) {
        setIRPower(param1 != NULL);
        return kIOReturnSuccess;
    }
    else if (functionName->isEqualTo(kSleepNow)) {
        putMachineToSleep ();
        return kIOReturnSuccess;
    }

    // If we are here it means we failed to find the correct call
    // so we pass the parametes to the function above:
```

[32] I have been reading lately about the Ruby programming language, for which the phrase "duck typing" is used to get this concept across. Objective-C has similar constructs, and I understand that the Mac OS X driver model grew out of a previous model that had been based on Objective-C. It certainly would have been fun to be able to program with Objective-C in the kernel!

```
    return super::callPlatformFunction (functionName,
            waitForFunction, param1, param2, param3, param4);
}
```

Listing 11. Partial contents of OpenPMU::callPlatformFunction

There is one additional wrinkle. How does a client know to talk to the `OpenPMU` driver at all? It could look for the PMU hardware nub and then see what driver has attached. However, the conventional approach is to call `waitForService (serviceMatching ("ApplePMU"))` to ask the kernel to find an `ApplePMU` object (or a subclass). The problem is that `OpenPMU` isn't a subclass — thus, it won't get picked up that way.

Fortunately, `waitForService` actually calls each object in the registry to ask it whether it is an `ApplePMU` subclass. So, we can pretend that we are.

```
bool
OpenPMU::passiveMatch (OSDictionary *matching, bool changesOK)
{
  // The problem is that other drivers call waitForService
  // (serviceMatching ("ApplePMU")). That would be OK if we could
  // inherit from ApplePMU, but we can't, since ApplePMU.h is not
  // available. So, instead, we just override passiveMatch so that
  // this will match ApplePMU. This works OK so long as the other
  // drivers use callPlatformFunction for talking to ApplePMU.

  OSString *str = OSDynamicCast (OSString, matching->getObject
            (gIOProviderClassKey));
  if (str && str->isEqualTo ("ApplePMU")) return true;
  return super::passiveMatch (matching, changesOK);
}
```

Listing 12. Pretending to be an ApplePMU driver

This way, clients expecting `ApplePMU` will successfully find `OpenPMU`, and will be able to call anything implemented by its `callPlatformFunction` method.

Adjusting Properties: When you need to tell a few white lies

Alongside its own member variables, each Mac OS X driver has a property table which is derived from `IOService` (the superclass for all Mac OS X drivers). Because the methods for accessing (and changing) those properties are public methods, it is reasonably easy to change the properties when necessary.

This was required when Apple dropped support for the "Beige G3" in Mac OS X 10.3 (Panther). The stated system requirements for Panther were a Power Macintosh with built-in USB.[33] If you tried to boot the Panther Install CD on one of the newly unsupported machines, you would end up in the Mac OS X Installer.[34] It would, however, then inform you that your machine was unsupported, and refuse to install.

[33] There wasn't actually anything magical about USB — it was simply a handy way to distinguish between the models which they were supporting and the models which were no longer supported. In fact, it happened to correspond with the dividing line between "Old World" and "New World" machines, which have differences in NVRAM structure and in the early parts of the boot process.

[34] Actually, you wouldn't get quite that far with the Beige G3, as there was a video-related bug which would cause a panic first (since fixed).

But wait! If you actually can get as far as booting the Mac OS X Install CD and having the Installer tell you that your machine is not supported, then clearly something is actually working. In fact, Apple had not removed very much of the driver code required to run Panther on the newly unsupported machines. Instead, they had merely gotten the Installer to check to see if the machine was unsupported, and refuse to install.

How did the Installer know what kind of machine it was running on? The `IOPlatformExpertDevice` at the root of the device-tree has a "model" property (and/or a "compatible" property) which identifies the machine as a whole. I theorized that the Installer was probably looking at that property to determine whether the installation process should continue or not. Thus, if we could change that property to something the Installer did not recognize, then the Installer might let the installation process continue.

The most orthodox way to do this would have been to subclass the platform expert, `GossamerPE`. However, the problem was that `GossamerPE`, like `ApplePMU`, was no longer open source. I could have based a platform expert on the last open source version (as I had with `ApplePMU`). However, since the closed-source `GossamerPE` continued to work, my preference was to continue to use it.

The alternative was to write a driver which would work alongside `GossamerPE`, rather than replacing it. I wrote a driver called `GossamerDeviceTreeUpdater` whose personality (in its Info.plist file) looked a lot like the personality for `GossamerPE`. The difference was the presence of a non-default IOMatchCategory. The Mac OS X kernel will match a single driver to each device per match category. Thus, if you use the default match category (by not specifying one), you can out-match the Apple driver. However, if you specify a non-default match category, then you can get your driver to load alongside the Apple driver. The Info.plist file for `GossamerDeviceTreeUpdater` kernel extension ended up looking something like this:

```
<key>GossamerDeviceTreeUpdater</key>
<dict>
    <key>CFBundleIdentifier</key>
    <string>com.macsales.iokit.GossamerDeviceTreeUpdater</string>
    <key>IOClass</key>
    <string>GossamerDeviceTreeUpdater</string>
    <key>IOMatchCategory</key>
    <string>GossamerDeviceTreeUpdater</string>
    <key>IONameMatch</key>
    <array>
        <string>AAPL,Gossamer</string>
        <string>AAPL,PowerMac G3</string>
        <string>AAPL,PowerBook1998</string>
        <string>AAPL,PowerBook1999</string>
        <string>PowerBook1,1</string>
        <string>PowerBook2,1</string>
        <string>iMac</string>
    </array>
    <key>IOProviderClass</key>
    <string>IOPlatformExpertDevice</string>
</dict>
```

Listing 13. Extract from Info.plist file for GossamerDeviceTreeUpdater.kext

This personality is basically the same as the personality for `GossamerPE`, except for the IOMatchCategory.[35] The IOMatchCategory tells the kernel to load this driver alongside other drivers, rather than replacing them.

The code in `GossamerDeviceTreeUpdater` then simply waits until the entire driver matching process is completed, and then changes the name, model and compatible properties in the `IOPlatformExpertDevice` to trick the Installer into going ahead with the install. The reason it waits is that (at least in Panther) there were some drivers which looked at the compatible property directly and adjusted their behaviour accordingly. Thus, it was necessary to keep the original value until that process was finished. At that point, the change could be made.

It worked nicely, and along with a couple of other small fixes permitted Mac OS X 10.3 (and now 10.4) to be installed on the Beige G3's.

Another case where it was necessary to adjust some properties arose in Mac OS X 10.3, which had a tendency to treat all SCSI drives as if they were removable. This was partly an aesthetic problem (the Finder put an "eject" button beside removable drives), and partly a substantive problem (the system mounted and unmounted removable drives when users logged in and out, which caused problems in certain configurations).

It turned out that the problem lay in the "Physical Interconnect Protocol" property set by one of the drivers — many SCSI controllers would set it to "Internal/External" rather than "Internal", because the SCSI chain might include either internal or external drives. The solution, then, was simply to modify that property in the appropriate registry entry. Because the `setProperty` method is public, you can actually do this from anywhere. You do not necessarily need to modify the driver — you can simply ask it to change a property, and it will obey. You can do this from another driver, or even from user-space code (if you are the root user).

Thus, in cases where all that is required is a change to the properties of an object in the device tree, the change can be quite easily implemented.

Intervening at the Correct Point: What to do when the conventions fail

The conventions adopted by Apple in the IOKit driver system mean that it is often possible to adjust the behaviour of the kernel or a kernel extension at a very granular level. In the NVRAM case, we were able to write a small driver that would out-match one specific driver built into the kernel. In the two storage examples, we were able to subclass an existing class in order to fix it, and out-match the original class so that our driver would be selected. Generally speaking, what I like to do is figure out the smallest, most sustainable change I can make in order to achieve the necessary result.

However, there are some occasions where the conventions of the driver system don't quite work out as you would like, and larger interventions are necessary. One example is some further work which was required on NVRAM in Mac OS X 10.4.

The underlying problem was not directly related to NVRAM. The issue was that the Mac OS X Installer would enforce an "8 GB limit" when installing onto an ATA drive on an "Old World" machine. The Installer would check to see whether a partition was wholly within the first 8 GB of the drive. If not, it would refuse to install onto that partition.

[35] Also, the still-supported models are left out of the IONameMatch array.

The reason the Installer enforced an "8 GB limit" was that there was a bug in the Open Firmware ATA driver for the built-in ATA bus on "Old World" machines. The driver could not read past the 8 GB mark on a drive. This meant that it would not be safe to install Mac OS X there. It might work, so long as everything required by BootX (the kernel, the kernel extensions, kernel caches and BootX itself) was within the first 8 GB. But there was no guarantee that they would be located there, or that they would all stay there when updates occurred.[36]

The difficulty was that the Installer was a little over-zealous in enforcing the 8 GB limit. It would enforce the limit even on ATA drives attached to PCI cards, which was not necessary (since the PCI cards had their own Open Firmware drivers, not typically subject to the 8 GB limit). This was a nuisance, but the problem became more serious in Mac OS X 10.4. The Installer started to think that every drive was an ATA drive. Worse yet, it thought that every partition extended past 8 GB (even if it did not). This made installing Mac OS X 10.4 on "Old World" machines very tricky indeed.[37]

If the Mac OS X Installer were open source, then it would have been reasonably easy to figure out why the Installer was behaving in this way — one could have determined what it was looking at that caused it to think that every drive was ATA, and every partition extended past 8 GB. It might then have been possible to work around the problem in some way. Since the Installer is not open source, it is more difficult to figure out precisely what is going wrong.

However, we do know that the problem does not affect "New World" machines — the Mac OS X Installer never applies the 8 GB limit to New World machines. So what if we were to pretend to be a "New World" machine? This would be problematic in one respect, since the Installer would then permit installations where the 8 GB limit was a real problem. However, it would at least let us install in cases where the 8 GB limit was not a problem.

I theorized that the Installer was distinguishing between "Old World" and "New World" machines based on the hw.epoch sysctl (hw.epoch is 0 for "Old World" and 1 for "New World"). Thus, if I could change the result returned by that sysctl, I could trick the Installer into believing that it was on a "New World" machine.

A little searching through the kernel source code revealed that the hw.epoch sysctl was set according to the result returned from `PEGetPlatformEpoch()`.[38] That, in turn, was implemented in the platform expert, by calling `getBootROMType()`.[39] This looked convenient — I was writing the platform expert anyway, so I could simply override `getBootROMType` (thankfully, a virtual function), and return the value I wanted.[40]

However, it was not quite that simple — there were side effects to consider. NVRAM structures differ between "Old World" and "New World" machines, and the kernel code responsible for NVRAM

[36] The problem did not affect Mac OS 9 because it used the Mac OS ROM to access the ATA drive, rather than the Open Firmware driver. In theory, it might have been possible to fix the Open Firmware driver with an NVRAM patch, but it may be that a patch was not feasible for one reason or another (NVRAM space on Old World machines is limited). It might also have been possible (in theory) to embed an ATA driver into BootX, but that may have been impractical, and would have required some kind of guarantee that BootX would be wholly within the first 8 GB, since it would need to be loaded by Open Firmware.

[37] All the "Old World" machines were unsupported by the time of Mac OS X 10.4, so Apple would not have tested for this problem.

[38] See <xnu/bsd/kern/kern_mib.c>.

[39] See <xnu/iokit/Kernel/IOPlatformExpert.cpp>

[40] Actually, I wasn't writing all the platform experts, but I could handle the existing platform experts by changing the value of the member variable returned by `getBootROMType`.

structures called `getBootROMType` in order to distinguish between them. Thus, by returning the "wrong" value from `getBootROMType`, I would completely mess up reading and writing NVRAM.

One possibility which I considered was trying to replace the code which responds to the hw.epoch sysctl, so that I could have it pretend to be a "New World" machine while `getBootROMType` continued to report that it was "Old World". This might have been possible by modifying the structure which defines how sysctl responds to the hw.epoch selector (the `sysctl__hw_epoch` symbol in the kernel). However, this would not have been a complete solution. The reason is that it is not only the structure of NVRAM that is different between "New World" and "Old World" machines. The content is also slightly different, and that content would be defined in user-space (and would thus rely on the sysctl to know whether it was on an "Old World" or "New World" machine). Of course, I could have compensated for this in my own application (since it would be able to tell whether the machine was only "pretending" to be New World). However, I preferred to have a more general solution if possible.[41]

So I had to look for another point at which to intervene. The ideal point to intervene would have been the `IODTNVRAM` class, which is the driver for the /options node in the device tree. The `IODTNVRAM` class is responsible for actually understanding the content and structure of the NVRAM buffer. It works with the `IONVRAMController` class[42] to read and write the buffer, but the content and structure of the buffer is the responsibility of `IODTNVRAM`. Thus, it is the `IODTNVRAM` class that knows about the differences between "Old World" and "New World" NVRAM — the `IONVRAMController` class just sees a buffer that needs to be written to (or read from) the hardware.

So if I could replace the `IODTNVRAM` class with a custom subclass, I could get it to "do the right thing" even though `getBootROMType` would be returning the "wrong" value. Unfortunately, this was a case in which the drivers were not following the usual conventions. Generally speaking, the platform expert would not create an `IODTNVRAM` instance directly. Instead, it would publish a "nub" which drivers could match (as described earlier). Ordinarily, things would be set up so that `IODTNVRAM` would "win", but the outcome could be altered if necessary without changing the platform expert.

Unfortunately, that's not how the platform expert does things in this case. Instead, it does this:

```
void IODTPlatformExpert::processTopLevel (IORegistryEntry *rootEntry) {
    …

    // Publish an IODTNVRAM class on /options.
    options = rootEntry->childFromPath("options", gIODTPlane);
    if (options) {
      dtNVRAM = new IODTNVRAM;
      if (dtNVRAM) {
        if (!dtNVRAM->init(options, gIODTPlane)) {
          dtNVRAM->release();
          dtNVRAM = 0;
        } else {
          dtNVRAM->attach(this);
          dtNVRAM->registerService();
        }
```

Listing 14. How not to instantiate a driver, if you want to maintain flexibility[43]

[41] I believe there was also some question about whether the `sysctl__hw_epoch` symbol was exported by the kernel, but I don't remember how hard I looked into that.
[42] The `PatchedAppleNVRAM` driver mentioned earlier is a subclass of `IONVRAMController`.
[43] The code is from <xnu/iokit/Kernel/IOPlatformExpert.cpp>.

Because the platform expert directly creates the `IODTNVRAM` instance with "`new`", they are tightly coupled. The /options node does not participate in the matching process, and there is no simple way of substituting a different driver for `IODTNVRAM`.

One way to deal with this would have been to subclass the platform expert (which, in some cases, I was doing anyway), and override the `processTopLevel` method so that it would instantiate a different driver. However, there were problems with this approach. One problem was that the `dtNVRAM` member variable had been declared "`private`". It was important to use the `dtNVRAM` variable, because various other `IODTPlatformExpert` methods used it — I couldn't just leave it `NULL`. Thus, I would need to construct a subclass of `IODTNVRAM` and then store it in the `dtNVRAM` variable.

I could have edited the IODTPlatformExpert.h header to make `dtNVRAM` protected instead of private (or made my subclass a `friend`). However, there was a more fundamental problem with subclassing in this case. The problem was that `IODTNVRAM` had been changing across versions of Mac OS X, and new member variables and virtual functions had been added without having reserved any space for them in the original headers. This meant that it would not be possible to write an `IODTNVRAM` subclass that worked across multiple versions of Mac OS X, and I was reluctant to rely on a driver that would require separate versions unless there was no alternative.

To avoid having to subclass `IODTNVRAM`, I could have overridden all of the methods in `IODTPlatformExpert` that used the `dtNVRAM` member variable (rendering it irrelevant). The problem was that some of the `IODTPlatformExpert` methods which used `dtNVRAM` were not declared `virtual`. Unlike the `public/protected/private` issue, this was not something that could be solved by editing headers. If the method wasn't in the vtable, then there was no way to override it in a subclass of `IODTPlatformExpert`.

So I appeared to be stuck. Because this part of the driver system was not following the usual conventions, I did not have a really good place to intervene in order to accomplish what I wanted.

However, I did have full control over the ultimate process of reading and writing a buffer to and from the hardware — that was something which my `PatchedAppleNVRAM` class was responsible for. It occurred to me that I could code `PatchedAppleNVRAM` to emulate New World NVRAM on an Old World machine. Ordinarily, it wasn't the job of `PatchedAppleNVRAM` to actually understand the NVRAM structures — it was merely meant to read and write a buffer prepared by `IODTNVRAM`. However, `PatchedAppleNVRAM` could be coded to understand the NVRAM structures if necessary, and convert "on the fly" from Old World to New World and vice versa. Essentially, it would read the "Old World" structures from the hardware, but then rewrite the buffer to conform to the "New World" structures before returning the buffer to the caller. Conversely, it would take the caller-supplied buffer meant for "New World" machines, and rewrite it to conform to "Old World" structures before writing it to hardware. That way, `IODTNVRAM` could think that it was on a "New World" machine, but the hardware would receive the correct "Old World" structures.[44]

In this case, `PatchedAppleNVRAM` was the entirely wrong place to intervene from every point of view except that it was easier to do. Logically speaking, it was `IODTNVRAM` or `IODTPlatformExpert` that needed the intervention. However, it was more practical to intervene with `PatchedAppleNVRAM` because that part of the system followed the conventions better.

[44] I also adjusted along the way for certain necessary differences in the content of NVRAM.

There was, however, one additional problem. I had forgotten that not all of the unsupported machines used `PatchedAppleNVRAM`. Some of them were PMU-based machines, and used the `ApplePMUNVRAMController` to read and write NVRAM. For this reason, changing `PatchedAppleNVRAM` did not help on those machines. This was a logical consequence of having intervened at the "wrong" point — if I had intervened with `IODTNVRAM` or `IODTPlatformExpert`, then the same intervention could have worked on the PMU-based machines as well.

So I needed to replace `ApplePMUNVRAMController` with my own driver, much as I had replaced `PatchedAppleNVRAM`. Writing the code was not difficult. However, getting the system to use my driver was a little trickier, because once again the conventions were not being followed. The PMU NVRAM driver was not being selected using the usual matching process — instead, like the `IODTNVRAM` driver, it was tightly coupled to the driver that created it (in this case, `ApplePMU`). As mentioned earlier, I did not have the source code for current versions of the `ApplePMU` driver. However, previous versions had created the NVRAM controller as follows:

```
bool ApplePMU::allocateInterfaces()
{
    …

    // gets the provider (which property we should look at):
    IOService *myProvider = getProvider();

    // Do not create the OpenPMUNVRAMController if the
    // no-nvram property is set.
    if (!myProvider->getProperty("no-nvram")) {
        ourNVRAMinterface = new ApplePMUNVRAMController;

        if (ourNVRAMinterface != NULL) {
            if (ourNVRAMinterface->init(0,this) &&
                ourNVRAMinterface->attach(this) &&
                ourNVRAMinterface->start(this)) {
                …
```

Listing 15. Another example of how not to instantiate a driver

The `ApplePMU` class and the `ApplePMUNVRAMController` class were tightly-coupled, because the `ApplePMU` class created the latter directly, rather than creating a nub that would participate in the matching process.

For some machines, I was substituting the `OpenPMU` class for `ApplePMU` anyway, so in those cases the problem was easy to fix. However, I did not want to use `OpenPMU` in cases where `ApplePMU` still worked, since `ApplePMU` was no longer open source, and I couldn't be sure what I was missing.

Fortunately, the `ApplePMU` and `ApplePMUNVRAMController` classes could be decoupled reasonably simply in this case. While the instance was stored in `ourNVRAMInterface`, this was only for the purpose of cleaning up when the driver was destroyed. Thus, it was possible to leave `ourNVRAMInterface` as `NULL` — it was not used elsewhere in the `ApplePMU` class.[45] Furthermore, the `ApplePMU` class checked for a "no-nvram" property in its provider. Thus, I could write a class that would wait for the provider to show up, and put a no-nvram property there. `ApplePMU` would then not create an instance of `ApplePMUNVRAMController`. My driver could then create and attach a

[45] At least, not in the last open-source version, and it turned out that the closed-source version hadn't changed in a way that broke this.

different driver. Because the conventions had been at least partially preserved in this case, it was possible to intervene without too much trouble.

Conclusion

Getting Mac OS X to run on some unsupported systems was, in part, a matter of updating the device drivers necessary for that hardware. However, there were also some issues which needed to be corrected in the kernel and kernel extensions supplied by Apple. The techniques used for doing so illustrate the flexibility and sophistication of the Mac OS X driver model, and the wisdom of abiding by its conventions where possible.[46]

[46] Additional information about the XPostFacto application (which implements these techniques) can be found at Other World Computing's web site (http://eshop.macsales.com/OSXCenter/). The source code for XPostFacto (itself an open source application) can be found at the OpenDarwin web site (http://www.opendarwin.org/).