Universal Binary Programming Guidelines, Second Edition

Contents

Chapter 4   Guidelines for Specific Scenarios   47

4

Chapter 5   Preparing Vector-Based Code   63

Appendix A   Rosetta   65

Appendix B   Architecture-Independent Vector-Based Code   75

Appendix C   Application Binary Interface   81

Appendix D   Using PowerPlant   83

5

Tables, Figures, and Listings

Appendix B   Architecture-Independent Vector-Based Code   75

Appendix D   Using PowerPlant   83

Introduction to Universal Binary Programming Guidelines

Universal Binary Programming Guidelines will assist experienced developers to build and modify their Mac OS X applications to run as universal binaries. **Universal binaries** run natively on Macintosh computers using PowerPC or Intel microprocessors and deliver optimal performance for both architectures in a single package.

This document is designed to help developers determine exactly how much work needs to be done and provides useful tips for general as well as specific code modification scenarios. It describes the prerequisites for building code as a universal binary and shows how to do so using Xcode 2.2. It also discusses the differences between the Intel and PowerPC architectures that can affect code behavior and provides guidelines for ensuring that universal binary code builds correctly.

This version of *Universal Binary Programming Guidelines* represents a significant update since its introduction at the Apple Worldwide Developers Conference in June, 2005. It brings together all the information that developers need to make the transition to Intel-based Macintosh computers. This version includes pointers to newly revised tools documentation—*Xcode 2.3 User Guide*, *GCC Porting Guide*, *Cross-Development Programming Guide*, and more—as well as improved guidelines and tips. Anyone who has an older version of *Universal Binary Programming Guidelines* will want to replace it with this version.

Who Should Read This Document?

Any developer who currently has an application that runs in Mac OS X will want to read this document to learn how to modify their code so that it runs natively on all current Apple hardware. Developers who have not yet written an application for the Macintosh, but are planning to do so, will want to follow the guidelines in the document to ensure that their code can run as a universal binary.

Organization of This Document

This document is organized into the following chapters:

- "Building a Universal Binary" (page 13) shows how to use Xcode 2.2 to build native and universal binaries, describes build options, and provides troubleshooting information for code that doesn't run properly on an Intel-based Macintosh computer.

■ "Architectural Differences" (page 23) outlines the major differences between the x86 and PowerPC architectures. Understanding the differences will help you to write portable code.

■ "Swapping Bytes" (page 29) describes byte-ordering differences in detail, provides a list of byte-swapping routines, and discusses strategies for a number of scenarios that require you to swap bytes. This is a must-read chapter for all Mac OS X developers. It will help you understand how to avoid byte-ordering issues when transferring data and data files between architectures.

■ "Guidelines for Specific Scenarios" (page 47) contains tips for a variety of situations that are not common to most applications.

■ "Preparing Vector-Based Code" (page 63) discusses the options available for those developers who have high-performance computing needs.

This document contains the following appendixes:

■ "Rosetta" (page 65) describes the translation process that allows PowerPC binaries to run on an Intel-based Macintosh computer.

■ "Architecture-Independent Vector-Based Code" (page 75) uses matrix multiplication as an example to show how to write vector code with a minimum amount of architecture-specific coding.

■ "Application Binary Interface" (page 81) provides information on where to find details.

■ "Using PowerPlant" (page 83) discusses the changes you need to make to use PowerPlant on an Intel-based Macintosh computer.

Assumptions

The document assumes the following:

■ Your application runs in Mac OS X.

Your application can use any of the Mac OS X development environments: Carbon, Cocoa, Java, or BSD UNIX.

If your application runs in the UNIX operating system but not specifically in Mac OS X, you should first read *Porting UNIX/Linux Applications to Mac OS X*.

If your application runs only in the Windows operating system, you should first read *Porting to Mac OS X from Windows Win32 API*.

If you are new to Mac OS X, you should take a look at *Mac OS X Technology Overview*.

■ You know how to use Xcode.

Currently Xcode is the only GUI tool available that compiles code to run universally.

If you are unfamiliar with Xcode, you might want to take a look at *Xcode 2.3 User Guide*.

If you have been using CodeWarrior, you should read *Porting CodeWarrior Projects to Xcode*.

Conventions

The term **x86** is a generic term used in some parts of this book to refer to the class of microprocessors manufactured by Intel. This book uses the term x86 as a synonym for IA-32 (Intel Architecture 32-bit).

Building a Universal Binary

Architectural differences between Macintosh computers that use Intel and PowerPC microprocessors can cause existing PowerPC code to behave differently when built and run natively on a Macintosh computer that uses an Intel microprocessor. The extent to which architectural differences affect your code depends on the level of your source code. Most existing code is high-level source code that is not specific to the processor. If your application falls into this category, you'll find that creating a universal binary involves adjusting code in a few places. Cocoa developers may need to make fewer adjustments than Carbon developers whose code was ported from Mac OS 9 to Mac OS X.

Most code that uses high-level frameworks and that builds with GCC 4.0 in Mac OS X v10.4 will build with few, if any, changes on an Intel-based Macintosh computer. The best approach for any developer in that situation is to build the existing code as a universal binary, as described in this chapter, and then see how the application runs on an Intel-based Macintosh. Find the places where the code doesn't behave as expected and consult the sections in this document that cover those issues.

Developers who use AltiVec instructions in their code or who intentionally exploit architectural differences for optimization or other purposes will need to make the most code adjustments. These developers will probably want to consult the rest of this document before building a universal binary. AltiVec programmers should read "Preparing Vector-Based Code" (page 63).

This chapter describes how to use Xcode 2.2 to create a universal binary, provides troubleshooting information, and lists relevant build options. You'll find that the software development workflow on an Intel-based Macintosh computer is exactly the same as the software development workflow on a PowerPC-based Macintosh.

Build Assumptions

Before you build your code as a universal binary, you must ensure that:

■ Your application already builds for Mac OS X. Your application can use any of the Mac OS X development environments: Carbon, Cocoa, Java, or BSD UNIX.

■ Your application uses the Mach-O executable format. Mach-O binaries are the only type of binary that run natively on an Intel-based Macintosh computer. If you are already using the Xcode compilers and linkers, your application is a Mach–O binary. Carbon applications based on the Code Fragment Manager Preferred Executable Format (PEF) must be changed to Mach-O.

■ Your Xcode target is a native Xcode target. If it isn't, in Xcode you can choose Project > Upgrade All Targets in Project to Native.

- Your code project is ported to GCC 4.0. Xcode uses GCC 4.0 for targeting Intel-based Macintosh computers. You may want to look at the document *GCC Porting Guide* to assess whether you need to make any changes to your code to allow it to compile using GCC 4.0.

- You installed the Mac OS X v10.4 universal SDK. The installer places the SDK in this location:

  `/Developer/SDKs/MacOSX10.4u.sdk`

Building Your Code

If you have already been using Xcode to build applications on a PowerPC-based Macintosh, you'll see that building your code on an Intel-based Macintosh computer is accomplished in the same way. By default, Xcode compiles code to run on the architecture on which you build your Xcode project. Note that your Xcode target must be a native target.

> **Tip:** CodeWarrior users can read Xcode from a CodeWarrior Perspective for a discussion of the similarities and differences between the two. This information can help you to put your CodeWarrior experience to work in Xcode.

When you are in the process of developing your project, you'll want to use the following settings for the Default and Debug configurations:

- Keep the Architectures settings set to `$(NATIVE_ARCH)`.

- Change the Mac OS X Deployment Target settings to `Mac OS X 10.4`.

- Make sure the SDKROOT setting is `/Developer/SDKs/MacOSX10.4u.sdk`.

You can set the SDK root for the project by following these steps:

1. Open your project in Xcode 2.2 or later.

   Make sure that your Xcode target is a native target. If it isn't, you can choose Project > Upgrade All Targets in Project to Native.

2. In the Groups & Files list, click the project name.

3. Click the Info button to open the Info window.

4. In the General pane, in the Cross-Develop Using Target SDK pop-up menu, choose Mac OS X 10.4 (Universal).

   If you don't see Mac OS X 10.4 (Universal) as a choice, look in the following directory to make sure that the universal SDK is installed:

   `/Developer/SDKs/MacOSX10.4u.sdk`

   If it's not there, you'll need to install this SDK before you can continue.

5. Click Change in the sheet that appears.

The Debug build configuration turns on ZeroLink, Fix and Continue, and debug-symbol generation, among other settings, and turns off code optimization.

When you are ready to test your application on both architectures, you'll want to use the Release configuration. This configuration turns off ZeroLink and Fix and Continue. It also sets the code-optimization level to optimize for size. As with the Default and Debug configurations, you'll want to set the Mac OS X Deployment Target to `Mac OS X 10.4` and the SDK root to `MacOSX10.4u.sdk`. To build a universal binary, the Architectures setting for the Release configuration must be set to build on Intel and PowerPC.

You can change the Architectures setting by following these steps:

1. Open your project in Xcode 2.2 or later.

2. In the Groups & Files list, click the project name.

3. Click the Info button to open the Info window.

4. In the Build pane (see Figure 1-1), choose Release from the Configuration pop-up menu.

Figure 1-1　　The Build pane

2006-07-24　|　© 2005, 2006 Apple Computer, Inc. All Rights Reserved.
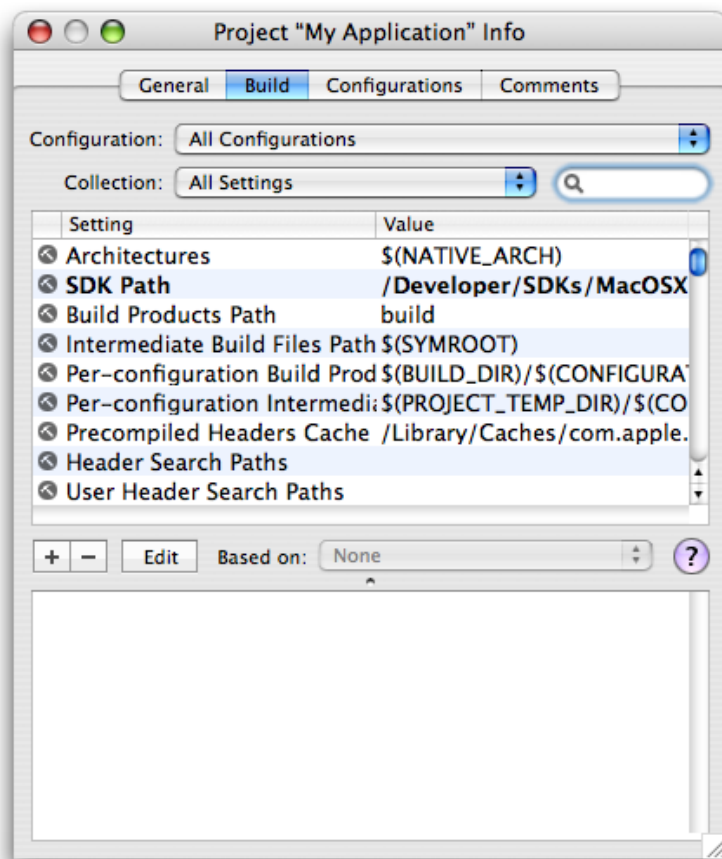
5. Select the Architectures setting and click Edit. In the sheet that appears, select the PowerPC and Intel options, as shown in Figure 1-2.

Figure 1-2   Architectures settings
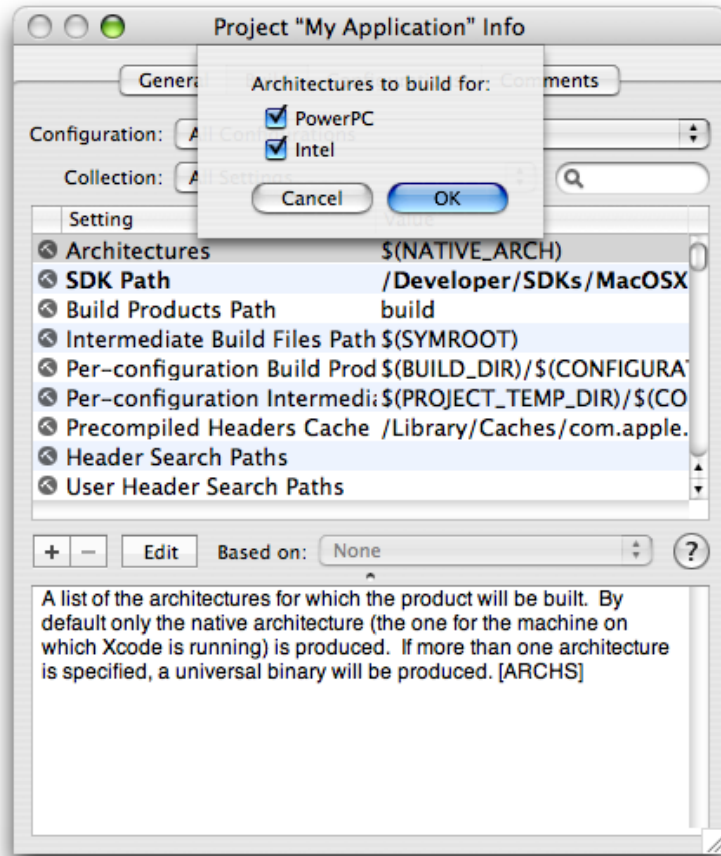
6. Close the Info window.

7. Build and run the project.

If your application doesn't build, see "Debugging" (page 17).

If your application builds but does not behave as expected when you run it as a native binary on an Intel-based Macintosh computer, see "Troubleshooting Your Built Application" (page 17).

If your application behaves as expected, don't assume that it also works on the other architecture. You need to test your application on both PowerPC Macintosh computers and Intel-based Macintosh computers. If your application reads data from and writes data to disk, you should make sure that you can save files on one architecture and open them on the other.

> **Note:** Xcode has per-architecture SDK support. For example, you can target Mac OS X versions 10.3 and 10.4 for PowerPC while also targeting Mac OS X v10.4 and later for Intel-based Macintosh computers.

For information on default compiler settings, architecture-specific options, and Autoconf macros, see

For information on building with version-specific SDKs for PowerPC (Mac OS X v10.3, v10.2, and so forth) while still building a universal binary for both PowerPC and Intel-based Macintosh computers, see the following resources:

- Using Cross Development in Xcode, in *Xcode 2.3 User Guide*.

- Cross Development and Universal Binaries in the *Cross-Development Programming Guide* provides details on to create executable files that contain object code for both Intel-based and PowerPC-based Macintosh computers.

Debugging

Xcode uses GDB for debugging, so you'll want to review the Debugging chapter in *Xcode 2.3 User Guide*. Xcode provides a powerful user interface to GDB that lets you step through your code, set breakpoints and view variables, stack frames, and threads.

Debugging with GDB—an Open Source document that explains how to use GDB—is another useful resource that you'll want to look at. It provides a lot of valuable information, including how to get a list of breakpoints for debugging.

If you are moving code to GCC 4.0, you can find remedies for most linking issues and compiler warnings by consulting *GCC Porting Guide*. You can find additional information on the GCC options you can use to request or suppress warnings in Section 3.8 of the *GNU C/C++/Objective-C 4.0.1 Compiler User Guide*.

Troubleshooting Your Built Application

Here are the most typical behavior problems you'll observe when your application runs natively on an Intel-based Macintosh computer:

- The application crashes.

- There are unexpected numerical results.

- Color is displayed incorrectly.

- Text is not displayed properly—characters from the Last Resort font or unexpected Chinese or Japanese characters appear.

- Files are not read or written correctly.

- Network communication does not work properly.

The first two problems in the list are typically caused by architecture-dependent code. On an Intel-based Macintosh computer, an integer divide-by-zero exception results in a crash, but on PowerPC the same operation returns zero. In these cases, the code must be rewritten in an architecture-independent manner. "Architectural Differences" (page 23) discusses the major differences between Macintosh computers that use PowerPC and Intel microprocessors. That chapter can help you determine which code is causing the crash or the unexpected numerical results.

The last four problems in the list are most often caused by byte-ordering differences between architectures. These problems are easily remedied by taking the byte order into account when you read and write data. The strategies available for handling byte ordering, as well as an in-depth discussion of byte-ordering differences, are provided in "Swapping Bytes" (page 29). Keep in mind that Mac OS X ensures that byte-ordering is correct for anything it is responsible for. Apple-defined resources (such as menus) won't result in problem behavior. Custom resources provided by your application, however, can result in problem behavior. For example, if images in your application seem to have a cyan tint, it's quite likely that your application is writing alpha channel data to the blue channel. For this specific issue, depending on the APIs that you are using, you'd want to consult the sections "GWorlds" (page 51), "Pixel Data " (page 58), or other graphics-related sections in "Guidelines for Specific Scenarios" (page 47).

Apple engineers prepared a lot of code to run natively on an Intel-based Macintosh computer—including the operating system, most Apple applications, and Apple tools. The guidelines in this book are the result of their work. In addition to the more common issues discussed in "Architectural Differences" (page 23) and "Swapping Bytes" (page 29), the engineers identified a number of narrowly focused issues. These are described in "Guidelines for Specific Scenarios" (page 47). You will want to at least glance at this chapter to see if your code can benefit from any of the information.

Determining Whether a Binary Is Universal

You can determine whether an application has a universal binary by looking at the Kind entry in the General section of the Info window for the application (see Figure 1-3). To open the Info window, click the application icon and press Cmd-I.

Figure 1-3   The Chess application has a Universal binary
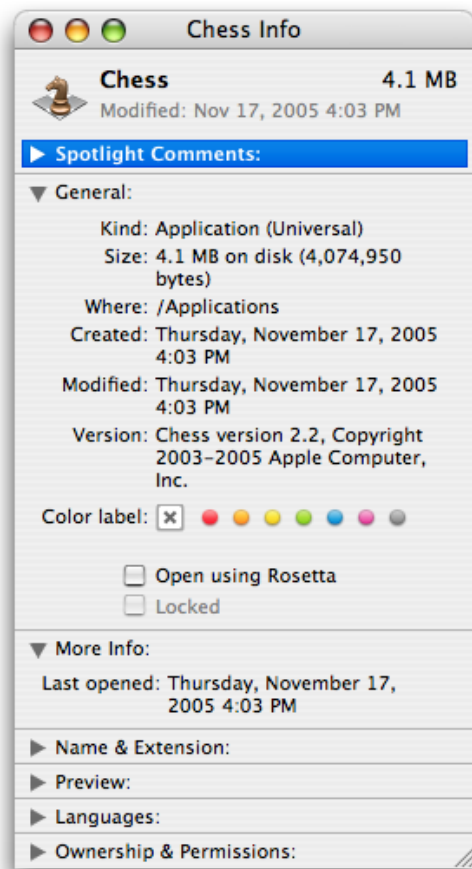
On an Intel-based Macintosh computer, when you double-click an application that doesn't have an executable for the native architecture, it might launch. Whether or not it launches depends on how compatible the application is with Rosetta. For more information, see "Rosetta" (page 65).

Build Options

This section contains information on the build options that you need to be aware of when using Xcode 2.2 and later on an Intel-based Macintosh computer. It lists the default compiler options, discusses how to set architecture-specific options, and provides information on using GNU Autoconf macros.

Default Compiler Options

In Xcode 2.2 and later on an Intel-based Macintosh computer, the defaults for compiler flags that differ from standard GCC distributions are listed in Table 1-1.

Table 1-1   Default values for compiler flags on an Intel-based Macintosh computer

Compiler flag	Default value	Specifies to
`-mfpmath`	`sse`	Use SSE instructions for floating-point math.
`-msse2`	On by default	Enable the MMX, SSE, and SSE2 extensions in the Intel instruction set architecture.

Architecture-Specific Options

Most developers don't need to use architecture-specific options for their projects.

In Xcode, to set one flag for an Intel-based Macintosh and another for PowerPC, you use the `PER_ARCH_CFLAGS_i386` and `PER_ARCH_CFLAGS_ppc` build settings variables to supply the architecture-specific settings.

For example to set the architecture-specific flags `-faltivec` and `-msse3`, you would add the following build settings:

```
PER_ARCH_CFLAGS_i386 = -msse3
PER_ARCH_CFLAGS_ppc = -faltivec
```

Similarly, you can supply architecture-specific linker flags using the `OTHER_LDFLAGS_i386` and `OTHER_LDFLAGS_ppc` build settings variables.

You can pass the `-arch` flag to `gcc`, `ld`, and `as`. The allowable values are `i386` and `ppc`. You can specify both flags as follows:

```
-arch ppc -arch i386
```

For more information on architecture-specific options, see Building Universal Binaries in *Xcode 2.3 User Guide*.

Autoconf Macros

If you are compiling a project that uses GNU Autoconf and trying to build it for both PowerPC-based and Intel-based Macintosh computers, you need to make sure that when the project configures itself, it doesn't use Autoconf macros to determine the endian type of the runtime system. For example, if your project uses the Autoconf `AC_C_BIGENDIAN` macro, the program won't work correctly when it is run on the opposite architecture from the one you are targeting when you configure the project. To correctly build for both PowerPC-based and Intel-based Macintosh computers, use the compiler-defined `__BIG_ENDIAN__` and `__LITTLE_ENDIAN__` macros in your code.

For more information, see Using GNU Autoconf in *Porting UNIX/Linux Applications to Mac OS X*.

See Also

These resources provide information related to compiling and building applications, and measuring performance:

- *Xcode 2.3 User Guide* contains all the instructions needed to compile and debug any type of Xcode project (C, C++, Objective C, Java, AppleScript, resource, nib files, and so forth).

- *GCC Porting Guide* provides advice for how to modify your code in ways that make it more compatible with GCC 4.0.

- *GNU C/C++/Objective-C 4.0.1 Compiler User Guide* provides details about the GCC implementation. Xcode uses the GNU compiler collection (GCC) to compile code.

  The assembler (`as`) used by Xcode supports AT&T System V/386 assembler syntax in order to maintain compatibility with the output from GCC. The AT&T syntax is quite different from Intel syntax. The major differences are discussed in the GNU documentation.

- *C++ Runtime Environment Programming Guide* provides information on the GCC 4.0 shared C++ runtime that is available in Panther 10.3.9 and later.

- *Porting UNIX/Linux Applications to Mac OS X*. Developers porting from UNIX and Linux applications who want to compile a universal binary, will want to read the section Compiling for Multiple Architectures.

- *Kernel Extension Concepts* contains information about debugging KEXTs on Intel-based Macintosh computers.

- Performance tools. Shark, MallocDebug, ObjectAlloc, Sampler, Quartz Debug, Thread Viewer, and other Apple-developed tools (some command-line, others use a GUI) are in the `/Developer` directory. Command-line performance tools are in the `/usr/bin` directory.

- *Code Size Performance Guidelines* and *Code Speed Performance Guidelines* discuss optimization strategies for a Mach-O executable.

Architectural Differences

The PowerPC and the x86 architectures have some fundamental differences that can prevent code written for one architecture from running properly on the other architecture. The extent to which you need to change your PowerPC code so that it runs natively on an Intel-based Macintosh computer depends on how much of your code is processor specific. This chapter describes the major differences between architectures, organized alphabetically by topic. You can use the information to identify the parts of your code that are likely to be problematic.

Alignment

All PowerPC instructions are 4 bytes in size and must be 4-byte aligned. x86 instructions are variable in size (from 1 to >10 bytes), and as a consequence do not need to be aligned.

Bit Fields

The value of a signed, 1-bit bit field is either `0`, `1`, or `-1`, depending on the compiler, architecture, optimization level, and so forth. Code that compares the value of a bit field to 1 may not work if the bit field is signed, so you will want to use unsigned 1-bit bit fields. Keep in mind that the order of bit fields in memory can be reversed between architectures.

For more information on issues related to endian format, see "Swapping Bytes" (page 29). See also "Archived Bit Fields" (page 47) and "Structures and Unions" (page 26).

Byte Order

Microprocessor architectures commonly use two different byte-ordering methods (little-endian and big-endian) to store the individual bytes of multibyte data formats in memory. This difference becomes critically important if you try to read data from files that were created on a computer that uses a different byte ordering than yours. You also need to consider byte ordering when you send and receive data through a network connection and handle networking data. The difference in byte ordering can produce incorrect results if you do not account for this difference. For example, the order of bytes in memory of a scalar type is architecture-dependent, as shown in Listing 2-1 (page 24).

Listing 2-1   Code that illustrates byte-ordering differences

```
unsigned char charVal;
unsigned long value = 0x12345678;
unsigned long *ptr = &value;
charVal = *(unsigned char*)ptr;
```

On a processor that uses little-endian addressing the variable `charVal` takes on the value 0x78. On a processor that uses big-endian addressing the variable `charVal` takes on the value 0x12. To make this code architecture-independent, change the last line in Listing 2-1 to the following:

```
charVal = (unsigned char)*ptr;
```

For a detailed discussion of byte ordering and strategies that you can use to account for byte-ordering differences, see "Swapping Bytes" (page 29).

Calling Conventions

The x86 C-language calling convention (application binary interface, or ABI) specifies that arguments to functions are passed on the stack. The PowerPC ABI specifies that arguments to functions are passed in registers. Also, x86 has far fewer registers, so many local variables use the stack for their storage. Thus, programming errors, or other operations that access past the end of a local variable array or otherwise incorrectly manipulate values on the stack may be more likely to crash applications on x86 systems than on PowerPC.

For detailed information about the IA-32 ABI, see *Mac OS X ABI Function Call Guide*. This document describes the function-calling conventions used in all the architectures supported by Mac OS X. See also "Application Binary Interface" (page 81).

Code on the Stack: Disabling Execution

Intel processors include a bit that prevents code from being executed on the stack. On Intel-based Macintosh computers, this bit is always set to `On`.

Data Type Conversions

For some data type conversions, such as casting a string to a long and converting a floating-point type to an integer type, the PowerPC and x86 architectures perform differently. When the microprocessor converts a floating-point type to an integer type, it discards the fractional part of the value. The behavior is undefined if the value of the integral part cannot be represented by the integer type.

Listing 2-2 shows an example of the sort of code that is architecture-dependent. You would need to modify this code to make it architecture-independent. On a PowerPC microprocessor, the variable x shown in the listing is equal to `7fffffff` or `INTMAX`. On an x86 microprocessor, the variable x is equal to `80000000` or `INTMIN`.

Listing 2-2   Architecture-dependent code

```
int main (int argc, const char * argv[])
{
    double  a;
    int     x;

    a = 5000000.0 * 6709000.5;  // or any really big value
    x = a;
    printf("x = %08x \n",x);
    return 0;
}
```

Data Types

A `long double` is 16 bytes on both architectures, but only 80 bits are significant in `long double` data types on Intel-based Macintosh computers.

A `bool` data type is a single byte on an x86 system, but four bytes on a PowerPC architecture. This size difference can cause alignment problems. You should use fixed-size data types to avoid alignment problems. (The `bool` data type is not the Carbon `Boolean` type, which is a fixed size of 1 byte.)

Existing document formats that include the `bool` data type as part of a data structure that is written directly to disk can be problematic because the data structure might not be laid out the same on both architectures. If you update the data structure definition to use the `UInt32` data type or another fixed-size four-byte data type, the structure should then be portable, although you must swap bytes appropriately.

Divide-By-Zero Operations

An integer divide-by-zero operation is fatal on an x86 system but the operation continues on a PowerPC system, where it returns zero. (A floating point divide-by-zero behaves the same on both architectures.) If you get a crash log that mentions `EXC_I386_DIV (divide by zero)`, your program divided by zero. Mod operations perform a divide, so a mod-by-zero operation produces a divide-by-zero exception. To fix a divide-by-zero exception, find the place in your program corresponding to that operation. Then add code that checks for a denominator of zero before performing the divide operation.

For example, change this:

```
 int a = b % c;    // Divide by zero can happen here;
```

to this:

```
int a;
if (c != 0) {
    a = b % c;
 } else {
    a = 0;
}
```

Extensible Firmware Interface (EFI)

Intel-based Macintosh computers use extensible firmware interface (EFI). EFI provides a flexible and adaptable interface between Mac OS X and the platform firmware. This change should be transparent to most developers, but may affect some, such as those who write boot drivers.

For more information on the EFI specification, see http://www.intel.com/technology/efi/

Floating-Point Equality Comparisons

The results of a floating-point equality comparison are architecture-dependent. Whether the comparison works depends on a number of things, including the compiler, the surrounding code, all compiler flags in use (particularly optimization flags), and the current floating-point mode for the thread. If your floating-point comparison is currently working on PowerPC, you may need to inspect it on an Intel-based Macintosh computer.

You can use the GCC flag `-Wfloat-equal` to receive a warning for floating-point equality comparisons. For details on this option, see Section 3.8 of the *GNU C/C++/Objective-C 4.0.1 Compiler User Guide*

Structures and Unions

The fields in a structure can be sensitive to their defined order. Structures must either be properly ordered or accessed by the field name directly.

When a union has components that could be affected by byte order, use a form similar to that shown in Listing 2-3. Code that sets `wch` and then reads `hi` and `lo` as the high and low bytes of `wch` will work correctly. The same is true for the reverse direction. Code that sets `hi` and `lo` and then reads `wch` will get the same value on both architectures. For another example, see the `WideChar` union that's defined in the `IntlResources.h` header file.

Listing 2-3   A union whose components can be affected by byte order

```
union WChar{
    unsigned short wch;
    struct {
#if __BIG_ENDIAN__
        unsigned char hi;
        unsigned char lo;
#else
        unsigned char lo;
        unsigned char hi;
#endif
    } s;
}
```

See Also

The ISO standard for the C programming language—ISO/IEC 9899—is a valuable reference that you can use to investigate code portability issues, many of which may not be immediately obvious. You can find this reference in a number of locations on the web, including:

http://www.iso.org/

Swapping Bytes

Two primary byte-ordering methods (or **endian formats**) exist in the world of computing. An endian format specifies how to store the individual bytes of multibyte numerical data in memory. **Big-endian byte ordering** specifies to store multibyte data with its most significant byte first. **Little-endian byte ordering** specifies to store multibyte data with its least significant byte first. The PowerPC processor uses big-endian byte ordering. The x86 processor family uses little-endian byte ordering. By convention, multibyte data sent over the network uses big-endian byte ordering.

If your application assumes that data is in one endian format, but the data is actually in another, then it will interpret the data incorrectly. You will want to analyze your code for routines that read multibyte data (16 bits, 32 bits, or 64 bits) from, or write multibyte data to, disk or to the network, as these routines are sensitive to byte-ordering format. There are two general approaches for handling byte ordering differences: swap bytes when necessary or use XML or another byte-order-independent data format such as those offered by Core Foundation (CFPreferences, CFPropertyList, CFXMLParser).

Whether you should swap bytes or use a byte-order-independent data format depends on how you use the data in your application. If you have an existing file format to support, the binary-compatible solution is to accept the big-endian file format you have been using in your application, and write code that swaps bytes when the file is read or written on an Intel-based Macintosh. If you don't have legacy files to support, you could consider redesigning your file format to use XML (extensible markup language), XDR (external data representation), or NSCoding (Objective C) to represent data.

This chapter describes why byte ordering matters, gives guidelines for swapping bytes, describes the byte-swapping APIs available in Mac OS X, and provides solutions for most of the situations where byte ordering matters.

Why Byte Ordering Matters

The example in this section is designed to show you why byte ordering matters. Take a look at the C data structure defined in Listing 3-1. It contains a four-byte integer, a character string, and a two-byte integer. The listing also initializes the structure.

Listing 3-1   A data structure that contains multibyte and single-byte data

```
typedef struct {
    uint32_t myOptions;
    char     myStringArray [7];
    short    myVariable;
} myDataStructure;
```

```
myDataStructure aStruct;

aStruct.myOptions = 0xfeedface;
strcpy(aStruct.myStringArray, "safari");
aStruct.myVariable = 0x1234;
```

Figure 3-1 compares how this data structure is stored in memory on big-endian and little-endian systems. In a big-endian system, memory is organized with the address of each data byte increasing from most significant to least significant. In a little-endian system, memory is organized with the address of each data byte increasing from the least significant to the most significant.

Figure 3-1   Big-endian byte ordering compared to little-endian byte ordering

As you look at Figure 3-1, note the following:

- Multibyte data, such as the 32-bit and 16-bit variables shown in the figure, are stored differently between big-endian and little-endian systems. As you can see in the figure, big-endian systems store data in memory so that the most significant byte of the data is stored in the address with the lowest value. Little-endian systems store data in memory so that the most significant byte of the data is in the address with the highest value. Hence, the least significant byte of the `myOptions` variable (`0xce`) is stored in memory location `0x00000003` on the big-endian system while it is stored in memory location `0x00000000` on the little-endian system.

- Single-byte data, such as the `char` values in the `myStringArray` character array, are stored in the same memory location on either system regardless of the byte ordering format of the system.

- Each system pads bytes to maintain four-byte data alignment. Padded bytes in the figure are designated by a shaded box that contains an asterisk.

The byte ordering of multibyte data in memory matters if you are reading data written on one architecture from a system that uses a different architecture and you access the data on a byte-by-byte basis. For example, if your application is written to access the second byte of the `myOptions` variable, then when you read the data from a system that uses the opposite byte ordering scheme, you'll end up retrieving the first byte of the `myOptions` variable instead of the second one.

Suppose the example data values that are initialized by the code shown in Listing 3-1 are generated on a little-endian system and saved to disk. Assume that the data is written to disk in byte-address order. When read from disk by a big-endian system, the data is again laid out in memory as shown in Figure 3-1. The problem is that the data is still in little-endian byte order even though it is interpreted on a big-endian system. This difference causes the values to be evaluated incorrectly. In this example, the value of the field `myOptions` should be `0xfeedface`, but because of the incorrect byte ordering it is evaluated as `0xcefaedfe`.

> **Note:** The terms *big-endian* and *little-endian* come from Jonathan Swift's eighteenth-century satire *Gulliver's Travels*. The subjects of the empire of Blefuscu were divided into two factions: those who ate eggs starting from the big end and those who ate eggs starting from the little end.

Guidelines for Swapping Bytes

The following guidelines, along with the strategies provided later in this chapter, will help ensure optimal byte-swapping code in your application.

- Keep data structures in native byte-order while in memory. Only swap bytes when you read data from disk or write it to disk.

- When possible, let the compiler do the work for you. For example, when you use function calls such as the Core Foundation function `CFSwapInt16BigToHost`, the compiler determines whether the function call does something for the processor you are targeting. If the code does nothing, the compiler won't call the function. Letting the compiler do the work is more efficient than using `#ifdef` statements.

- If you must access a large file, consider arranging the data in a way that limits the byte swapping that you must perform. For example, you can arrange the most frequently accessed data contiguously in the file. Then, you need to read and swap bytes only for that chunk of data instead of for the entire data file.

- Use the `__BIG_ENDIAN__` and `__LITTLE_ENDIAN__` macros only if you must. Do not use macros that check for a specific processor type, such as `__i386__` and `__ppc__`.

- Choose a consistent byte-order approach and stick with it. That is, if you are reading and writing data from disk on a regular basis, choose the endian format you want to use. This eliminates the need for you to check the byte ordering of the data, and then to possibly have to swap the byte order.

- Be aware of which functions return big-endian data, and use them appropriately. These include the BSD Sockets networking functions, the `DNSServiceDiscovery` functions (for example, TCP and UDP ports are specified in network byte order), and the ColorSync profile functions (for which all data is big-endian). The `IconFamilyElement` and `IconFamilyResource` data types (which also include the data types `IconFamilyPtr` and `IconFamilyHandle`) are always big-endian.

There may be other functions and data types that are not listed here. Consult the appropriate API reference for information on data returned by a function. For more information see "Network-Related Data" (page 36).

■ Keep in mind that swapping bytes comes at a performance cost so swap them only when absolutely necessary.

Byte-Swapping Routines

The APIs that provide byte-swapping routines are listed below. For most situations it's best to use the routines that match the framework you're programming in. The Core Foundation and Foundation APIs have functions for swapping floating-point values, while the other APIs listed do not.

■ POSIX (Portable Operating System Interface) byte ordering functions (`ntohl`, `htonl`, `ntohs`, and `htons`) are documented in *Mac OS X Man Pages*.

■ Darwin byte ordering functions and macros are defined in the header file `libkern/OSByteOrder.h`. Even though this header is in the kernel framework, it is acceptable to use it from high-level applications.

■ Core Foundation byte-order functions are defined in the header file `CoreFoundation/CFByteOrder.h` and described in the *Byte-Order Utilities Reference*. For details on using these functions, see the Byte Swapping article in *Memory Management Programming Guide for Core Foundation*.

■ Foundation byte-order functions are defined in the header file `Foundation/NSByteOrder.h` and described in *Foundation Framework Reference*.

■ The Core Endian API is defined in the header file `CarbonCore/Endian.h` and described in *Core Endian Reference*.

> **Note:** When you use byte-swapping routines, the compiler optimizes your code so that the routines are executed only if they are needed for the architecture on which your code is running.

Byte-Swapping Strategies

The strategy for swapping bytes depends on the format of the data; there is no universal routine that can take care of all byte ordering differences. Any program that needs to swap data must know the data type, the source data endian order, and the host endian order.

This section lists byte-swapping strategies, organized alphabetically, for the following data:

Constants

Constants that are part of a compiled executable are in host byte order. You need to swap bytes for a constant only if it is part of data that is not maintained natively or if the constant travels between hosts. In most cases you can either swap bytes ahead of time or let the preprocessor perform any needed math by using shifts or other simple operators.

If you are defining and populating a structure that must use data of a specific endian format in memory, use the `OSSwapConst` macros and the `OSSwap*Const` variants defined in the `libkern/OSByteOrder.h` header file. These macros can be used from high-level applications.

Custom Apple Event Data

An Apple event is a high-level event that conforms to the Apple Event Interprocess Messaging Protocol. The Apple Event Manager sends Apple events between applications on the same computer or between applications on remote computers. You can define your own Apple event data types, and send and receive Apple events using the Apple Event Manager API.

Mac OS X manages system-defined Apple event data types for you, handling them appropriately for the currently executing code. You don't need to perform any special tasks. When the data that your application extracts from an Apple event is system-defined, the system swaps the data for you before giving the event to your application to process. You will want to treat system-defined data types from Apple events as native endian. Similarly, if you put native-endian data into an Apple event that you are sending, and it is a system-defined data type, the receiver will be able to interpret the data in its own native endian format.

However, you must account for byte-ordering differences for the custom Apple event data types that you define. You can accomplish this in one of the following ways:

■   Write a byte-swapping callback routine (also known as a flipper) and provide it to the system. Whenever the system determines that your Apple event data needs to be byte swapped it invokes your flipper to ensure that the recipient of the data gets the data in the correct endian format. For details, see "Writing a Callback to Swap Data Bytes" (page 39).

■   Choose one endian format to use, regardless of architecture. Then, when you read or write your custom Apple event data, use big-to-host and host-to-big routines, such as the Core Foundation Byte Order Utilities functions `CFSwapInt16BigToHost` and `CFSwapInt16HostToBig`.

Custom Resource Data

In Mac OS X, the preferred way to supply resources is to provide files in your application bundle that define resources such as image files, sounds, localized text, and archived user-interface definitions. The resource data types discussed in this section are those defined in Resource Manager-style files supported by Carbon. The Resource Manager was created prior to Mac OS X. If your application uses Resource Manager-style resource files, you should consider moving towards Mac OS X–style resources in your application bundle instead.

Resources typically include data that describes menus, windows, controls, dialogs, sounds, fonts, and icons. Although the system defines a number of standard resource types (such as `'moov'`, used to specify a QuickTime movie, and `'MENU'`, used to define menus) you can also create your own private resource types for use in your application. You use the Resource Manager API to define resource data types and to get and set resource data.

Mac OS X keeps track of resources in memory and allows your application to read or write resources. Applications and system software interpret the data for a resource according to its resource type. Although you'll typically let the operating system read resources (such as your application icon) for you, you can also call Resource Manager functions directly to read and write resources.

Mac OS X manages the system-defined resources for you, handling them appropriately for the currently executing code. That is, if your application runs on an Intel-based Macintosh, Mac OS X swaps bytes so that your application icon, menus, and other standard resources appear correctly. You don't need to perform any special tasks. But if you define your own private resource data types for use in your application, you need to account for byte-ordering differences between architectures when you read or write resource data from disk.

You can use either of the following strategies to handle custom Resource Manager-style resource data. Notice that these are the same strategies used to handle custom Apple event data:

■ Provide a byte-swapping callback routine for the system to invoke whenever the system determines your resource data must be byte swapped. For details, see "Writing a Callback to Swap Data Bytes" (page 39).

■ Always write your data using the same endian format, regardless of the architecture. Then, when you read or write your custom resource data, use big-to-host and host-to-big routines, such as the Core Foundation Byte Order Utilities `CFSwapInt16BigToHost` and `CFSwapInt16HostToBig`.

> **Note:** If you are revising old code that marks resources with a preload bit, you should remove the preload bit from any resources that must be byte swapped. In Mac OS X, the preload bit is almost always unnecessary. If you cannot remove the preload bit, you should swap the resource data after you read the resource. You will not be able to use a flipper callback to swap bytes automatically because in Mac OS X a preload bit causes the resources to be read before any of the application code runs.

Floating-Point Values

Core Foundation defines a set of functions and two special data types to help you work with floating-point values. These functions allow you to encode 32- and 64-bit floating-point values in such a way that they can later be decoded and byte swapped if necessary. Listing 3-2 shows you how to encode a 64-bit floating-point number and Listing 3-3 shows how to decode it.

Listing 3-2   Encoding a 64-bit floating-point value

```
double d = 3.0;
CFSwappedFloat64 swappedDouble;
// Encode the floating-point value.
swappedDouble = CFConvertFloat64HostToSwapped(d);
// Call the appropriate routine to write swappedDouble to disk,
// send it to another process, etc.
write(myFile, &swappedDouble, sizeof(swappedDouble));
```

The data types CFSwappedFloat32 and CFSwappedFloat64 contain floating-point values in a canonical representation. A CFSwappedFloat data type is not itself a floating-point value, and should not be directly used as one. You can however send one to another process, save it to disk, or send it over a network. Because the format is converted to and from the canonical format by the conversion functions, there is no need for explicit swapping. Bytes are swapped for you during the format conversion if necessary.

Listing 3-3   Decoding a 32-bit floating-point value

```
float f;
CFSwappedFloat32 swappedFloat;
// Call the appropriate routine to read swappedFloat from disk,
// receive it from another process, etc.
read(myFile, &swappedFloat, sizeof(swappedFloat));
f = CFConvertFloat32SwappedToHost(swappedFloat)
```

The NSByteOrder.h header file defines functions that are comparable to the Core Foundation functions discussed here.

Integers

The system library byte-access functions, such as OSReadLittleInt16 and OSWriteLittleInt16, provide generic byte swapping. These functions swap bytes if the native endian format is different from the endian format of the destination. They are defined in the libkern/OSByteOrder.h header file.

Note: The OSReadXXX and OSWriteXXX functions provide higher performance than the OSSwapXXX functions or any other functions in the higher-level frameworks.

Core Foundation provides three optimized primitive functions for swapping bytes— CFSwapInt16, CFSwapInt32, and CFSwapInt64. All of the other swapping functions use these primitives to accomplish their work. In general you don't need to use these primitives directly.

Although the primitive swapping functions swap unconditionally, the higher-level swapping functions are defined in such a way that they do nothing when swapping bytes is not required—in other words, when the source and host byte orders are the same. For the integer types, these functions take the forms CFSwapXXXBigToHost, CFSwapXXXLittleToHost, CFSwapXXXHostToBig, and CFSwapXXXHostToLittle, where XXX is a data type such as Int32. For example, on a little-endian machine you use the function CFSwapInt16BigToHost to read a 16-bit integer value from a network whose data is in network byte order (big-endian). Listing 3-4 demonstrates this process.

Listing 3-4   Swapping a 16-bit integer from big-endian to host-endian

```
SInt16  bigEndian16;
SInt16  swapped16;
// Swap a 16-bit value read from the network.
swapped16 = CFSwapInt16BigToHost(bigEndian16);
```

Suppose the integers are in the fields of a data structure. Listing 3-5 demonstrates how to swap bytes.

Listing 3-5   Swapping integers from little-endian to host-endian

```
// Swap the bytes of the values if necessary.
aStruct.int1 = CFSwapInt32LittleToHost(aStruct.int1)
aStruct.int2 = CFSwapInt32LittleToHost(aStruct.int2)
```

The code swaps bytes only if necessary. If the host is a big-endian architecture, the functions used in the code sample swap the bytes in each field. The code does nothing when run on a little-endian machine—the compiler ignores the code.

Network-Related Data

Network-related data typically uses big-endian format (also known as **network byte order**), so you may need to swap bytes when communicating between the network and an Intel-based Macintosh computer. You probably never had to adjust your PowerPC code when you transmitted data to, or received data from, the network. On an Intel-based Macintosh computer you must look closely at your networking code and ensure that you always send network-related data in the appropriate byte order. You must also handle data received from the network appropriately, swapping the bytes of values to the endian format appropriate to the host microprocessor.

You can use the following POSIX functions to convert between network byte order and host byte order. (Other byte-swapping functions, such as those defined in the `OSByteOrder.h` and `CFByteOrder.h` header files, can also be useful for handling network data.)

■   network to host:

```
uint32_t ntohl (uint32_t netlong);

uint16_t ntohs (uint16_t netshort);
```

■   host to network:

```
uint32_t htonl (uint32_t hostlong);

uint16_t htons (uint16_t hostshort);
```

These functions are documented in *Mac OS X Man Pages*.

The `sin_saddr.s_addr` and `sin_port` fields of a `sockaddr_in` structure should always be in network byte order. You can find out the appropriate endian format of any argument to a BSD networking function by reading the man page documentation.

When advertising a service on the network, you use `getsockname` to get the local TCP or UDP port that your socket is bound to, and then pass `my_sockaddr.sin_port` unchanged, without any byte swapping, to the `DNSServiceRegister` function.

In CoreFoundation code, you can use the same approach. Use the `CFSocketCopyAddress` function as shown below, and then pass `my_sockaddr.sin_port` unchanged, without any byte swapping, to the `DNSServiceRegister` function.

```
CFDataRef addr = CFSocketCopyAddress(myCFSocketRef);
struct sockaddr_in my_sockaddr;
memmove(&my_sockaddr, CFDataGetBytePtr(addr), sizeof(my_sockaddr));
DNSServiceRegister( ... , my_sockaddr.sin_port, ...);
```

When browsing and resolving, the process is similar. The `DNSServiceResolve` function and the BSD Sockets calls such as `gethostbyname` and `getaddrinfo` all return IP addresses and ports already in the correct byte order so that you can assign them directly to your `struct sockaddr_in` and call `connect` to open a TCP connection. If you byte-swap the address or port, then you program will not work.

The important point is that when you use the DNSServiceDiscovery API with the BSD Sockets networking APIs, you do not need to swap anything. Your code will work correctly on both PowerPC and Intel-based Macintosh computers as well as on Linux, Solaris, and Windows.

OSType-to-String Conversions

You can use the functions `UTCreateStringForOSType` and `UTGetOSTypeFromString` to convert an `OSType` data type to or from a `CFString` object (`CFStringRef` data type). These functions are discussed in *Uniform Type Identifiers Overview* and defined in the `UTType.h` header file, which is part of the Launch Services framework.

When you use four-character literals, keep in mind that "abcd" != 'abcd'. Rather 'abcd' == 0x61626364. You must treat 'abcd' as an integer and not string data, as 'abcd' is a shortcut for a 32-bit integer. (A `FourCharCode` data type is a `UInt32` data type.) The compiler does not swap this for you. You can use the shift operator if you need to deal with individual characters.

For example, if you currently print an `OSType` or `FourCharCode` type using the standard C `printf`-style semantics, use

```
printf("%c%c%c%c", (char) (val >> 24), (char) (val  >> 16),
                    (char) (val >> 8), (char) val)
```

instead of the following:

```
printf("%4.4s", (const char*) &val)
```

Unicode Text Files

Mac OS X often uses UTF-16 to encode Unicode; a `UniChar` data type is a double-byte value. As with any multibyte data, Unicode characters are sensitive to the byte ordering method used by the microprocessor. A byte order mark written to the beginning of a file informs the program reading the data which byte ordering method was used to write the data. The Unicode standard states that in the absence of a byte order mark (BOM) the data in a Unicode data file is to be taken as big-endian. Although a BOM is not mandatory, you should make use of it to ensure that a file written on one architecture can be read from the other architecture. The program can then act accordingly to make sure the byte ordering of the Unicode text is compatible with the host.

Table 3-1 lists the standard byte order marks for UTF-8, UTF-16, and UTF-32. (Note that the UTF-8 BOM is not used for endian issues, but only as a tag to indicate that the file is UTF-8.)

Table 3-1   Byte order marks

Byte order mark	Encoding form
EF BB BF	UTF-8
FF FE	UTF-16/UCS-2, little endian
FE FF	UTF-16/UCS-2, big endian
FF FE 00 00	UTF-32/UCS-4, little endian
00 00 FE FF	UTF-32/UCS-4, big endian

In practice, when your application reads a file, it does not need to look for a byte order mark nor does it need to swap bytes as long as you follow these steps to read a file:

1. Map the file using `mmap` to get a pointer to the contents of the file (or string).

   Reading the entire file into memory ensures the best performance and is a prerequisite for the next step.

2. Generate a `CFString` object by calling the function `CFStringCreateWithBytes` with the `isExternalRepresentation` parameter set to `true`, or call the function `CFStringCreateWithExternalRepresentation` to generate a `CFString`, passing in an encoding of `kCFStringEncodingUnicode` (for UTF-16) or `kCFStringEncodingUTF8` (for UTF-8).

   Either function interprets a BOM swaps bytes as necessary. Note that a BOM should not be used in memory; its use is solely for data transmission (files, pasteboard, and so forth).

In summary, with respect to Unicode files, your application performs best when you follow these guidelines:

■ Accept the BOM when taking UTF-16 or UTF-8 encoded files from outside the application.

■ Use native-endian `UniChar` data types internally.

■ Generate a BOM when writing UTF-16 to a file. Ideally, you only need to generate a BOM for an architecture that uses little-endian format, but it is also acceptable to generate a BOM for an architecture that uses big-endian format.

■ When you put data on the Clipboard, make sure that `'utxt'` data does not have a BOM. Only `'ut16'` data should have a BOM. If you use Cocoa to put an `NSString` object on the pasteboard, you don't need to concern yourself with a BOM.

For more information, see "UTF & BOM," available from the Unicode website:

http://www.unicode.org/faq/utf_bom.html

The Apple Event Manager provides text constants that you can use to specify the type of your data. As of Mac OS X v10.4, only two text constants are recommended:

- `typeUTF16ExternalRepresentation`, which specifies Unicode text in 16-bit external representation with optional byte order mark (BOM). The presence of this constant guarantees that either there is a BOM or the data is in UTF-16 big-endian format.

- `typeUTF8Text`, which specifies 8-bit Unicode (UTF-8 encoding).

The constant `typeUnicodeText` indicates `utxt` text data, in native byte ordering format, with an optional BOM. This constant does not specify an explicit Unicode encoding or byte order definition.

The Scrap Manager provides the flavor type constant `kScrapFlavorTypeUTF16External` which specifies Unicode text in 16-bit external representation with optional byte order mark (BOM).

Values in an Array

The routine in Listing 3-6 shows an approach that you can use to swap the bytes of values in an array. On a big-endian system, the compiler optimizes away the entire function; you don't need to use `#ifdef` statements to swap these sorts of arrays.

Listing 3-6   A routine for swapping the bytes of the values in an array

```
static inline void SwapUInt32ArrayBigToHost(UInt32 *array, UInt32  count) {
    int i;

    for(i = 0; i < count; i++) {
        array[i] = CFSwapInt32BigToHost(array[i]);
    }
}
```

Writing a Callback to Swap Data Bytes

You can provide a byte-swapping callback routine, also referred to as a **flipper**, to the system for custom resource data, custom pasteboard data, and custom Apple event data. When you install a byte-swapping callback, you specify which domain that the data type belongs to. There are two data domains—Apple event and resource. The resource data domain specifies custom pasteboard data or custom resource data. If the callback can be applied to either domain (Apple event and resource), you can specify that as well.

The Core Endian API defines a callback that you provide to swap bytes for custom resource and Apple event data. You must provide one callback for each type of data you want to swap bytes. The prototype for the `CoreEndianFlipProc` callback is:

```
typedef CALLBACK_API (OSStatus, CoreEndianFlipProc)
    (OSType dataDomain,
    OSType dataType,
    short id,
    void *dataPtr,
    UInt32 dataSize,
    Boolean currentlyNative,
    void *refcon
);
```

The callback takes the following parameters:

■ `dataDomain`—An `OSType` value that specifies the domain to which the flipper callback applies. The value `kCoreEndianResourceManagerDomain` signifies that the domain is resource or pasteboard data. The value `kCoreEndianAppleEventManagerDomain` signifies that the domain is Apple event data.

■ `dataType`—The type of data that needs the callback to swap bytes for. This is the four-character code of the resource type, pasteboard type, or Apple event.

■ `id`—The resource id of the data type. This field is ignored if the `dataDomain` parameter is not `kCoreEndianResourceManagerDomain`.

■ `dataPtr`—On input, points to the data to be flipped. On output, points to the byte swapped data.

■ `dataSize`—The size of the data pointed to by the `dataPtr` parameter.

■ `currentlyNative`—A Boolean value that indicates the direction to swap bytes. The value `true` specifies the data pointed to by the `dataPtr` parameter uses the byte ordering of the currently executing code. On a PowerPC Macintosh, `true` specifies that the data is in big-endian format. On an Intel-based Macintosh, `true` specifies that the data is in little-endian format.

■ `refcon`—A 32-bit value that contains, or refers to, data needed by the callback.

The callback returns a result code that indicates whether bytes are swapped successfully. Your callback should return `noErr` if the data is byte swapped without error and the appropriate result code to indicate an error condition—`errCoreEndianDataTooShortForFormat`, `errCoreEndianDataTooLongForFormat`, or `errCoreEndianDataDoesNotMatchFormat`. The result code you return is propagated through the appropriate manager (Resource Manager (`ResError`) or Apple Event Manager) to the caller.

You do not need to swap bytes for quantities that are not numerical (such as strings, byte streams, and so forth). You need to provide a callback only to swap bytes data types for which the order of bytes in a word or long word are important. (For the preferred way to handle Unicode strings, see "Unicode Text Files" (page 37).)

Your callback should traverse the data structure that contains the data and swap bytes for:

■ All counts and lengths so that array indexes are associated with the appropriate value

■ All integers and longs so that when you read them into variables of a compatible type, you can operate correctly on the values (such as numerical, offset, and shift operations)

The Core Endian API provides these functions for working with your callback:

■ `CoreEndianInstallFlipper` registers your callback for the specified data type (custom resource or custom Apple Event). After you register a byte-swapping callback for an application-defined resource data type, then any time you call a Resource Manager function that operates on that resource type, the system invokes your callback if it is appropriate to do so. (If your callback operates on pasteboard data, the system also invokes the callback at the appropriate time.) Similarly, if you specify Apple event as the domain for your callback, then any time you call an Apple Event Manager function that operates on that data type, your callback is invoked when it is appropriate to do so.

■ `CoreEndianGetFlipper` obtains the callback that is registered for the specified data type. You can call this function to determine whether a flipper is available for a given data type.

■   `CoreEndianFlipData` invokes the callback associated with the specified data type. You shouldn't need to call this function, because the system invokes your callback whenever it's needed.

As an example, look at a callback for the custom resource type (`'PREF'`) defined in Listing 3-7. The `MyPreferences` structure is used to store preferences data on disk. The structure contains a number of values and includes two instances of the `RGBColor` data type and an array of `RGBColor` values.

Listing 3-7   A declaration for a custom resource

```
#define kMyPreferencesType        'PREF'

struct MyPreferences {
            SInt32          fPrefsVersion;

            Boolean         fHighlightLinks;
            Boolean         fUnderlineLinks;

            RGBColor        fHighlightColor;
            RGBColor        fUnderlineColor;
            SInt16          fZoomValue;

            char            fCString[32];

            SInt16          fCount;
            RGBColor        fPalette[];
};
```

You can handle the `RGBColor` data type by writing a function that swaps bytes in an `RGBColor` data structure, such as the function `MyRGBSwap`, shown in Listing 3-8. This function calls the Core Endian macro `EndianS16_Swap` to swap bytes for each of the values in the `RGBColor` data structure. The function doesn't need to check for the currently executing system because the function is never called unless the values in the `RGBColor` data type need to have their bytes swapped. The `MyRGBSwap` function is called by the byte-swapping callback routine (shown in Listing 3-9 (page 41)) that's provided to handle the custom `'PREF'` resource (that is defined in Listing 3-7 (page 41)).

Listing 3-8   A flipper function for RGBColor data

```
static void MyRGBSwap (RGBColor *p)
{
    p->red = Endian16_Swap(p->red);
    p->blue = Endian16_Swap(p->blue);
    p->green = Endian16_Swap(p->green);
}
```

Listing 3-9 shows a byte-swapping callback for the custom `'PREF'` resource. An explanation for each numbered line of code appears following the listing. Note that the flipper checks for data that is malformed or is of an unexpected length. If the data passed into the flipper routine is a shorter length than the flipped type is normally, or (for example) contains garbage data instead of an array count, the flipper must be careful not to read or write data beyond the end of the passed-in data. Instead, the routine returns an error.

Listing 3-9   A flipper for the custom 'PREF' resource

```
#define kCurrentVersion     0x00010400
```

```
static OSStatus MyFlipPreferences (OSType dataDomain,                      // 1
                  OSType dataType,                                         // 2
                  short id,                                                // 3
                  void * dataPtr,                                          // 4
                  UInt32 dataSize,                                         // 5
                  Boolean currentlyNative,                                 // 6
                  void* refcon)                                            // 7
{
    UInt32  versionNumber;

    OSStatus status = noErr;
    MyPreferences* toFlip = (MyPreferences*) dataPtr;                      // 8
    int count, i;

    if (dataSize < sizeof(MyPreferences))
        return errCoreEndianDataTooShortForFormat;                        // 9
    if (currentlyNative)                                                   // 10
    {
        count = toFlip->fCount;
        versionNumber = toFlip->fPrefsVersion;
        toFlip->fPrefsVersion = Endian32_Swap (toFlip->fPrefsVersion);
        toFlip->fCount = Endian16_Swap (toFlip->fCount);
        toFlip->fZoomValue = Endian16_Swap (toFlip->fZoomValue);
    }
    else                                                                  // 11
    {
        toFlip->fPrefsVersion = Endian32_Swap (toFlip->fPrefsVersion);
        versionNumber = toFlip->fPrefsVersion;
        toFlip->fCount = Endian16_Swap (toFlip->fCount);
        toFlip->fZoomValue = Endian16_Swap (toFlip->fZoomValue);
        count = toFlip->fCount;
    }
    if (versionNumber != kCurrentVersion)                                 // 12
                return errCoreEndianDataDoesNotMatchFormat;

    MyRGBSwap (&toFlip->fHighlightColor);                                 // 13
    MyRGBSwap (&toFlip->fUnderlineColor);                                 // 14

    if (dataSize < sizeof(MyPreferences) + count * sizeof(RGBColor))
        return errCoreEndianDataTooShortForFormat;                        // 15

    for(i = 0; i < count; i++)
    {
        MyRGBSwap (&toFlip->fPalette[i]);                                 // 16
    }

    return status;                                                       // 17
}
```

Here's what the code does:

1. The system passes to your callback the domain to which the callback applies. You define the domain when you register the callback using the function `CoreEndianInstallFlipper`.

2. The system passes to your callback the resource type you defined for the data. In this example, the resource type is `'PREF'`.

3. The system passes to your callback the resource ID of the data type. If the data is not a resource, this value is `0`.

4. The system passes to your callback a pointer to the resource data that needs to have its bytes swapped. In this case, the pointer refers to a `MyPreferences` data structure.

5. The system passes to your callback the size of the data pointed to by the pointer described in the previous step.

6. The system passes to your callback `true` if the data in the buffer passed to the callback is in the byte ordering of the currently executing code. On a PowerPC Macintosh, when `currentlyNative` is `true`, the data is in big-endian order. On a Macintosh that uses an Intel microprocessor, when `currentlyNative` is `true`, the data is in little-endian order. Your callback needs to know this value, because if your callback uses a value in the data buffer to decide how to process other data in the buffer (for example, the `count` variable shown in the code), you must know whether that value needs to be flipped before the value can be used by the callback.

7. The system passes to your callback a pointer that refers to application-specific data. In this example, the callback doesn't require any application-specific data.

8. Defines a variable for the `MyPreferences` data type and assigns the contents of the data pointer to the newly-defined `toFlip` variable.

9. Checks the static-length portion of the structure. If the size is less than it should be, the routine returns the error `errCoreEndianDataTooLongForFormat`.

10. If `currentlyNative` is `true`, saves the count value to a local variable and then swaps the bytes for the other values in the `MyPreferences` data structure. You must save the count value before you swap because you need it for an iteration later in the function. The fact that `currentlyNative` is `true` indicates that the value does not need to be byte swapped if it is used in the currently executing code. However, the value does need to be swapped to be stored to disk.

    The values are swapped using the appropriate Core Endian macros.

11. If `currentlyNative` is `false`, flips the values in the `MyPreferences` data structure before it saves the count value to a local variable. The fact that `currentlyNative` is `false` indicates that the count value needs to have its bytes swapped before it can be used in the callback.

12. Checks to make sure the version of the data structure is supported by the application. If the version is not supported, then your callback would not swap bytes for the data and would return the result `errCoreEndianDataDoesNotMatchFormat`.

13. Calls the `MyRGBSwap` function (shown in Listing 3-8 (page 41)) to swap the bytes of the `fHighlightColor` field of the data structure.

14. Calls the `MyRGBSwap` function to swap the bytes of the `fUnderlineColor` field of the data structure.

15. Checks the data size to make sure that it is less than it should be. If not, the routine returns the error `errCoreEndianDataTooLongForFormat`.

16. Iterates through the elements in the `fPalette` array, calling the `MyRGBSwap` function to swap the bytes of the data in the array.

17. Returns `noErr` to indicate that the data is flipped without error.

Although the sample performs some error checking, it does not include all the error-handling code that it could. When you write a flipper you may want to include such code.

> **Note:** The callback does not flip any of the Boolean values in the `MyPreferences` data structure because these are single character values. The callback also ignores the C string.

You register a byte-swapping callback routine by calling the function `CoreEndianInstallFlipper`. You should register the callback when your application calls its initialization routine or when you open your resources. For example, you would register the flipper callback shown in using the following code:

```
OSStatus status = noErr;
status = CoreEndianInstallFlipper (kCoreEndianResourceManagerDomain,
                        kMyPreferencesType,
                        MyFlipPreferences,
                        NULL);
```

The system invokes the callback for the specified resource type and data domain when `currentlyNative` is `false` at the time a resource is loaded and `true` at the time the resource is set to be written. For example, the sample byte-swapping callback gets invoked any time the following line of code is executed in your application:

```
MyPreferences** hPrefs = (MyPreferences**) GetResource ('PREF',  128);
```

After swapping the bytes of the data, you can modify it as much as you'd like.

When the Resource Manager reads a resource from disk, it looks up the resource type (for example, `'PREF'`) in a table of byte-swapping routines. If a callback is installed for that resource type, the Resource Manager invokes the callback if it is appropriate to do so. Similar actions are taken when the Resource Manager writes a resource to disk. It finds the appropriate routine and invokes the callback to swap the bytes of the resource if it is appropriate to do so.

When you copy or drag custom data from an application that has a callback installed for pasteboard data, the system invokes your callback at the appropriate time. If you copy or drag custom data to a native application, the data callback is not invoked. If you copy or drag custom data to a nonnative application, the system invokes your callback to swap the bytes of the custom data. If you paste or drop custom data into your application from a nonnative application, and a callback exists for that custom data, the system invokes the callback at the time of the paste or drop. If the custom data is copied or dragged from another native application, the callback is not invoked.

Note that different pasteboard APIs use different type specifiers. The Scrap Manager and Drag Manager use `OSType` data types. The Pasteboard Manager uses Uniform Type Identifiers (UTI), and the `NSPasteboard` class uses its own type mechanism. In each case, the type is converted by the system to an `OSType` data type to discover if there is a byte-swapping callback for that type.

Apple event data types are typically swapped to network byte order when sent over a network. The callback you install is called only if a custom data type that you define is sent to another machine, or if another machine sends Apple event data to your application. The byte ordering of Apple events on the network is big-endian.

For cases in which the system would not normally invoke your byte-swapping callback, you can call the function `CoreEndianFlipData` to invoke the callback function installed for the specified data type and domain.

See Also

The following resources are available in the ADC Reference Library:

■ *Byte-Order Utilities Reference* describes the Core Foundation byte order utilities API.

■ Byte Swapping, in Core Foundation Memory Management, shows how to swap integers and floating-point values using Core Foundation byte-order utilities.

■ *File-System Performance Guidelines* provides information useful for mapping Unicode files to memory.

Guidelines for Specific Scenarios

This chapter lists an assortment of scenarios that relate to a specific technology or API. Although many of these scenarios are uncommon, you will want to at least glance at the topics to determine whether anything applies to your application. The topics are organized alphabetically.

Aliases

Aliases are big-endian on all systems. Applications that add extra information to the end of an `AliasHandle` must ensure that the extra data is always endian-neutral or of a defined endian type, preferably big-endian.

The `AliasRecord` data structure is opaque when building your application with the Mac OS X v10.4(Universal) SDK. Code that formerly accessed the `userType` field of an `AliasRecord` must use the Alias Manager functions `GetAliasUserType`, `GetAliasUserTypeFromPtr`, `SetAliasUserType`, or `SetAliasUserTypeFromPtr`. Code that formerly accessed the `aliasSize` field of an `AliasRecord` must use the functions `GetAliasSize` or `GetAliasSizeFromPtr`.

These Alias Manger functions are available in Mac OS X v10.4 and later. For more information, see *Alias Manager Reference*.

Archived Bit Fields

It's best not to use the `NSArchiver` class to archive any structures that contain bit fields as integers. Individual values are stored in the archives in an architecture and compiler dependent manner. In cases where archives already contain such structures, you can read a structure correctly by changing its declaration so that the bit fields are swapped appropriately, as shown in Listing 4-1.

You might want to examine your code to make sure such changes don't affect other code. Note that this workaround is specific to GCC.

Listing 4-1   A structure that swaps bit fields

```
typedef struct {
#ifdef __BIG_ENDIAN__
    unsigned int rotatedFromBase:1;
    unsigned int aboutToResize:1;
```

```
    unsigned int stuff:30;
#else
    unsigned int stuff:30;
    unsigned int aboutToResize:1;
    unsigned int rotatedFromBase:1;
#endif
}   _flags;
```

Automator Scripts

AppleScript actions are platform-independent and do not need any changes to run on Intel-based Macintosh computers. However, any action that contains Cocoa code, whether it is a solely Cocoa action or an action that uses both AppleScript and Cocoa code, must be built as a universal binary to run correctly on both architectures.

For more information, see *Automator Programming Guide*.

Bit Shifting

When you shift a value by the width of its type or more, the fill bits are undefined regardless of the architecture. In fact, two different compilers on the same architecture could differ on the value of y after these two statements:

```
uint32_t x = 0xDEADBEEF;

uint32_t y = x >> 32;
```

Bit Test, Set, and Clear Functions: Carbon and POSIX

Don't mix using the C bitwise operators with the Carbon functions `BitTst`, `BitSet`, and `BitClr` and the POSIX macros `setbit`, `clrbit`, `isset`, and `isclr`. If you consistently use the Carbon and POSIX functions and avoid the C bitwise operators, your code will function properly. Keep in mind, however, that you must use the Carbon and POSIX functions on the correct kind of data. The Carbon and POSIX functions perform a byte-by-byte traversal, which causes problems on an Intel-based Macintosh when they operate on data types that are larger than 1 byte. You can use these functions only on a pointer to a string of endian-neutral bytes. When you need to perform bit manipulation on integer values you should use functions such as `(int32 & (1 << 26))` instead of `BitTst(&int32, 5L)`.

You'll encounter problems when you use the function `BitTst` to test for 24-bit mode. For example, the following bit test returns `false`, which indicates that the process is running in 24-bit mode, or at least that the code is not running in 32-bit mode. The POSIX equivalents perform similarly:

```
Gestalt(gestaltAddressingModeAttr, &gestaltResult);
if (!(BitTst(&gestaltResult,31L)) )      /*If 24 bit
```

You can use any of the bit testing, setting, and clearing functions if you pass a pointer to data whose byte order is fixed. Used in this way, these functions behave the same on both architectures.

For more information, see the `ToolUtils.h` header file in the Core Services framework and *Mathematical and Logical Utilities Reference*.

CPU Subtype

Don't try to build a binary for a specific CPU subtype. Since the CPU subtype for Intel-based Macintosh computers is generic, you can't use it to check for specific functionality. If your application requires information about specific CPU functionality, use the `sysctlbyname` function, providing an appropriate selector. See *Mac OS X Man Pages* for information on using `sysctlbyname`.

Dashboard Widgets

Dashboard widgets typically contain platform-independent elements such as HTML, JavaScript, CSS, and image files. If you create a widget that contains only these elements, it should work on both PowerPC and Intel-based Macintosh computers without any modification on your part. However, if your widget contains a plug-in, you must build the plug-in as a universal binary for it to run natively on an Intel-based Macintosh computer.

For more information, see *Dashboard Programming Topics*.

Deprecated Functions

Many deprecated functions, such as those that use `PICT + PS` data, have byte swapping issues. You may want to replace deprecated functions at the same time you prepare your code to run as a universal binary. You'll not only solve byte swapping issues, but your code will use functions that ultimately benefit future development.

A function that is deprecated has an availability statement in its header file that states the version of Mac OS X in which the function is deprecated. Many API reference documents provide a list of deprecated functions. In addition, compiler warnings for deprecated functions are on by default in Xcode 2.2 and later.

Disk Partitions

The standard disk partition format on an Intel-based Macintosh computer differs from the disk partition format of a PowerPC-based Macintosh computer. If your application depends on the partitioning details of the disk, it may not behave as expected. Partitioning details can affect tools that examine the hard disk at a low level.

By default, internal hard drives on Intel-based Macintosh computers use the GUID Partition Table (GPT) scheme and external drives use the Apple Partition Map (APM) partition scheme. To create an external USB or FireWire disk that can boot an Intel-based Macintosh computer, select the GPT disk partition scheme option using Apple Disk Utility. Starting up an Intel-based Macintosh using an APM disk is not supported.

Double-Precision Values: Bit-by-Bit Sensitivity

Although both architectures are IEEE 754 compliant, there are differences in the rounding procedure used by each when operating on double-precision numbers. If your application is sensitive to bit-by-bit values in double-precision numbers, be aware that the same computation performed on each architecture may produce a different numerical result.

For more information, see Volume 1 of the Intel developer software manuals, available from the following website:

http://developer.intel.com/design/Pentium4/documentation.htm

Finder Information and Low-Level File System Operations

If your code operates on the file system at a low level and handles Finder information, keep in mind that the file system does not swap bytes for the following information:

- The `finderInfo` field in the HFSPlus data structures `HFSCatalogFolder`, `HFSPlusCatalogFolder`, `HFSCatalogFile`, `HFSPlusCatalogFile`, and `HFSPlusVolumeHeader`.

- The `FSPermissionInfo` data structure, which is used when the constant `kFSCatInfoPermissions` is passed to the HFSPlus functions `FSGetCatalogInfo` and `FSGetCatalogInfoBulk`.

The value of multibyte fields on disk always uses big-endian format. When running on a little-endian system, you must swap the bytes of any multibyte fields.

The `getattrlist` function retrieves the metadata associated with a file. The `getxattr` function, added in Mac OS X v10.4, retrieves extended attributes—those that are an extension of the basic set of attributes. When using the `getxattr` function to access the legacy attribute `"com.apple.FinderInfo"`, note that as with `getattrlist`, the information returned by this call is not byte swapped. (For more information on the `getxattr` and `getattrlist` functions see *Mac OS X Man Pages*.)

> **Note:** This issue pertains only to code that operates below CarbonCore. Calls to Carbon functions such as `FSGetCatalogInfo` are not affected.

FireWire Device Access

The FireWire bus uses big-endian format. If you are developing a universal binary version of an application that accesses a FireWire device, see "FireWire Device Access on an Intel-Based Macintosh" in *FireWire Device Interface Guide* for a discussion of the issues you can encounter.

Font-Related Resources

Font-related resource types (`FOND`, `NFNT`, `sfnt`, and so forth) are in big-endian format on both PowerPC and Intel-based Macintosh computers. If your application accesses font-related resource types directly, you must swap the fields of font-related resource types yourself.

The following functions from the ATS for Fonts API obtain font resources that are returned in big-endian format:

- `ATSFontGetTableDirectory`
- `ATSFontGetTable`
- `ATSFontGetFontFamilyResource`

The following functions from the Font Manager API obtain font resources that are returned in big-endian format. Note that Font Manager API is based on QuickDraw technology, which was deprecated in Mac OS X v10.4.

- `FMGetFontTableDirectory`
- `FMGetFontTable`
- `FMGetFontFamilyResource`

GWorlds

When the QuickDraw function `NewGWorld` allocates storage for the pixel buffer, and the depth parameter is 16 or 32 bits, the byte ordering within each pixel matters. The `pixelFormat` field of the `PixMap` data structure can have the values `k16BE555PixelFormat` or `k16LE555PixelFormat` for 2-byte pixels, and `k32ARGBPixelFormat` or `k32BGRAPixelFormat` for 4-byte pixels. (These constants are defined in the `Quickdraw.h` header file.) By default, `NewGWorld` always creates big-endian pixel formats (`k16BE555PixelFormat` or `k32ARGBPixelFormat`), regardless of the endian format of the system.

For best performance, it is generally preferable for you to use a pixel format that corresponds to the native byte ordering of the system. When you pass `kNativeEndianPixMap` in the `flags` parameter to `NewGWorld`, the byte ordering of the pixel format is big-endian on big-endian systems, and little-endian on little-endian systems.

> **Note:** QuickDraw does not support little-endian pixel formats on big-endian systems.

You can use the GWorld pixel storage as input to the Quartz function `CGBitmapContextCreate` or as a data provider for the Quartz function `CGImageCreate`. The byte ordering of the source pixel format needs to be communicated to Quartz through additional flags in the `bitmapInfo` parameter. These flags are defined in the `CGImage.h` header file. Assuming that your `bitmapInfo` parameter is already set up, you now need to combine it (by using a bitwise `OR` operator) with `kCGBitmapByteOrder16Host` or `kCGBitmapByteOrder32Host` if you created the GWorld with a `kNativeEndianPixMap` flag. Similarly, you should use `kCGBitmapByteOrder16Big` or `kCGBitmapByteOrder32Big` when you know that your pixel byte order is big-endian.

Java Applications

Pure Java applications do not require any code changes to run on Intel-based Macintosh computers. However, Java applications that interface with PowerPC-based native code will not run successfully using Rosetta on Intel-based Macintosh computers.

Specifically, the following must be built as universal binaries:

■ JNI libraries built for PowerPC-based Macintosh computers are not loaded using Rosetta because the Java Virtual Machine has already launched without using Rosetta. Java applications fail on Intel-based Macintosh computers when trying to load PowerPC-only binaries.

■ Native applications that use the VM Invocation Interface to start a Java Virtual Machine must be built as universal binaries to run on Intel-based Macintosh computers. The Java VM must run natively; attempts by an application running using Rosetta to instantiate a JVM fail.

For more information, see *Technical Q &A QA1295: Java on Intel-based Macintosh Computers* in the ADC Reference Library.

Java I/O API (NIO)

The I/O API (NIO) that was introduced in JDK 1.4 allows the use of native memory buffers. If you are a Java programmer who uses this API, you may need to revise your code. NIO byte buffers have a byte ordering that by default is big-endian. If you have Java code originally written for Mac OS X on PowerPC, when you create `java.nio.ByteBuffers` you should call the function `ByteBuffer.order(ByteOrder.nativeOrder())` to set the byte order of the buffers to the native byte order for the current architecture. If you fail to do this, you will obtain flipped data when you read multibyte data from the buffer using JNI.

Machine Location Data Structure

The Memory Management Utilities data type `MachineLocation` contains information about the geographical location of a computer. The `ReadLocation` and `WriteLocation` functions use the geographic location record to read and store the geographic location and time zone information in extended parameter RAM.

If your code uses the `MachineLocation` data structure, you need to change it to use the `MachineLocation.u.dls.Delta` field that was added to the structure in Mac OS X version 10.0.

To be endian-safe, change code that uses the old field:

```
MachineLocation.u.dlsDelta = 1;
```

to use the new field:

```
MachineLocation.u.dls.Delta = 1;
```

The `gmtDelta` field remains the same—the low 24 bits are used. The order of assignment is important. The following is incorrect because it overwrites results:

```
MachineLocation.u.dls.Delta = 0xAA;     // u = 0xAAGGGGGG; G=Garbage
MachineLocation.u.gmtDelta = 0xBBBBBB;  // u = 0x00BBBBBB;
```

This is the correct way to assign the values:

```
MachineLocation.u.gmtDelta = 0xBBBBBB;  // u = 0x00BBBBBB;
MachineLocation.u.dls.Delta = 0xAA;     // u = 0xAABBBBBB;
```

For more details see *Memory Management Utilities Reference*.

Mach Processes: The Task for PID Function

The `task_for_pid` function returns the task associated with a process ID (PID). This function can be called only if the process is owned by the `procmod` group or if the caller is `root`.

Metrowerks PowerPlant

Applications that use Metrowerks PowerPlant framework and its `PPob` resources need to make changes to the PowerPlant code. See "Using PowerPlant" (page 83) for details.

Multithreading

Multithreading is a technique used to improve performance and enhance the perceived responsiveness of applications. On computers with one processor, this technique can allow a program to execute multiple pieces of code *independently*. On computers with more than one processor, multithreading can allow a program to execute multiple pieces of code *simultaneously*. If your application is single-threaded, consider threading your application to take advantage of hardware multithreading processor capabilities. If your application is multithreaded, you'll want to ensure that the number of threads is not hard coded to a fixed number of processors.

Dual-core technology improves performance by providing two physical cores within a single physical processor package. Multiprocessor and dual-core technology all exploit thread-level parallelism to improve application and system responsiveness and to boost processor throughput.

When you prepare code to run as a universal binary, the multithreading capabilities of the microprocessor are transparent to you. This is true whether your application is threaded or not. However, you can optimize your code to take advantage of the specific way hardware multithreading is implemented for each architecture.

Objective-C: Messages to nil

In Objective-C, it is valid to send a message to a `nil` object. The Objective-C runtime assumes that the return value of a message sent to a `nil` object is `nil`, as long as the message returns an object or any integer scalar of size less than or equal to `sizeof(void*)`.

On Intel-based Macintosh computers, messages to a `nil` object always return `0.0` for methods whose return type is `float`, `double`, `long double`, or `long long`. Methods whose return value is a `struct`, as defined by the *Mac OS X ABI Function Call Guide* to be returned in registers, will return `0.0` for every field in the data structure. Other `struct` data types will not be filled with zeros. This is also true under Rosetta. On PowerPC Macintosh computers, the behavior is undefined.

Objective-C Runtime: Sending Messages

The information in this section is only for developers who use the Objective-C runtime library, which is used primarily for developing bridge layers between Objective-C and other languages, or for low-level debugging. Most developers do not need to use the Objective-C runtime library directly when programming in Objective-C.

If your application directly calls the Objective-C runtime function `objc_msgSend_stret`, you need to change your code to have it work correctly on an Intel-based Macintosh.

The x86 ABI for struct-return functions differs from the ABI for `struct-address-as-first-parameter` functions, but the two ABIs are identical on PowerPC. When you call `objc_msgSend_stret`, you must cast the function to a function pointer type that uses the expected `struct` return type. The same applies for calls to `objc_msgSendSuper_stret`.

For other details on the ABI, see "Application Binary Interface" (page 81).

If your application directly calls the Objective-C runtime function `objc_msgSend`, you should always cast to the appropriate return value. For instance, for a method that returns a `BOOL` data type, the following code executes properly on a PPC Macintosh but might not on an Intel-based Macintosh computer:

```
BOOL isEqual = objc_msgSend(string, @selector("isEqual:"), otherString);
```

To ensure that the code does executes properly on an Intel-based Macintosh computer, you would change the code to the following:

```
BOOL isEqual = ((BOOL (*)(id, SEL, id))objc_msgSend)(object,
@selector("isEqual:"), otherString);
```

Open Firmware

Macintosh computers that use an Intel microprocessor do not use Open Firmware. Although many parts of the I/O registry are present and work as expected, information that is provided by Open Firmware on a PowerPC Macintosh (such as a complete device tree) is not available in the I/O registry on a Macintosh that uses an Intel microprocessor. You can obtain some of the information from IODeviceTree by using the `sysctlbyname` or `sysctl` commands.

OpenGL

When defining an OpenGL image or texture, you need to provide a type that specifies to OpenGL which format the texture is in. Most of these functions (for example, `glTexImage2D`) take `format` and `type_` parameters that specify how the texture is laid out on disk or in memory. OpenGL supports a number of different image types; some are endian-neutral but others are not.

For example, a common image format is `GL_RGBA` with a type of `GL_UNSIGNED_BYTE`. This means that the image has a byte that specifies the red color data followed by a byte that specifies the green color data, and so forth. This format is not endian-specific; the bytes are in the same order on all architectures. Another common image format is `GL_BGRA`, often specified by the type `GL_UNSIGNED_INT_8_8_8_8_REV`. This type means that every 4 bytes of image data are interpreted as an `unsigned int`, with the most significant 8 bits representing the alpha data, the next most significant 8 bits representing the red color data, and so forth. Because this format is specific to the integer format of the host, the format is interpreted differently on little-endian systems than on big-endian systems. When using `GL_UNSIGNED_INT_8_8_8_8_REV`, the OpenGL implementation expects to find data in byte order ARGB on big-endian systems, but BGRA on little-endian systems.

Because there is no explicit way in OpenGL to specify a byte order of ARGB with 32-bit or 16-bit packed pixels (which are common image formats on Macintosh PowerPC computers), many applications specify `GL_BGRA` with `GL_UNSIGNED_INT_8_8_8_8_REV`. This practice works on a big-endian system such as PowerPC, but the format is interpreted differently on a little-endian system and causes images to be rendered with incorrect colors.

Applications that have this problem are those that use the OpenGL host-order format types, but assume that the data referred to is always big-endian. These types include, but are not limited to the following:

```
GL_SHORT
GL_UNSIGNED_SHORT
GL_INT
GL_UNSIGNED_INT
GL_FLOAT
GL_DOUBLE
GL_UNSIGNED_BYTE_3_3_2
GL_UNSIGNED_SHORT_4_4_4_4
GL_UNSIGNED_SHORT_5_5_5_1
GL_UNSIGNED_INT_8_8_8_8
GL_UNSIGNED_INT_10_10_10_2
GL_UNSIGNED_SHORT_5_6_5
GL_UNSIGNED_BYTE_2_3_3_REV
GL_UNSIGNED_SHORT_5_6_5_REV
GL_UNSIGNED_SHORT_4_4_4_4_REV
GL_UNSIGNED_SHORT_1_5_5_5_REV
GL_UNSIGNED_INT_8_8_8_8_REV
GL_UNSIGNED_INT_2_10_10_10_REV
```

If your application does not use any of these types, it is unlikely to have any problems with OpenGL. Note that an application is not necessarily incorrect to use one of these types. Many applications might already present host-order data tagged with one of these formats, especially with existing cross-platform code, because the Mac OS X implementation behaves the same way as a Windows implementation.

If an application incorrectly uses one of these types, its OpenGL textures and images are rendered with incorrect colors. For example, red might appear green, or the image might appear to be tinted purple.

You can fix this problem in one of the following ways:

1. If the images are generated or loaded algorithmically, change the code to generate the textures in host-order format that matches what OpenGL expects. For example, a JPEG decoder can be modified to store its output in 32-bit integers instead of four 8-bit bytes. The resulting data is identical on big-endian systems, but on a little-endian system, the bytes are in a different order. This matches the OpenGL expectation, and the existing OpenGL code continues to work on both architectures. This is the preferred approach.

    In many cases, rewriting the algorithms may prove a significant amount of work to implement and debug. If that's the case, an approach that asks OpenGL to interpret the texture data differently might be a better approach for you to take.

2. If the application uses `GL_UNSIGNED_INT_8_8_8_8_REV` or `GL_UNSIGNED_INT_8_8_8_8`, it can switch between them based on the architecture. Since these two types are exactly byte swapped versions of the same format, using `GL_UNSIGNED_INT_8_8_8_8_REV` on a big-endian system is equivalent to using `GL_UNSIGNED_INT_8_8_8_8` on a little-endian system and vice versa. Code might look as follows:

    ```
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_BGRA_EXT,
    #if __BIG_ENDIAN__
                    GL_UNSIGNED_INT_8_8_8_8_REV,
    #else
                    GL_UNSIGNED_INT_8_8_8_8,
    #endif
                    data);
    ```

If this is a common idiom, it might be easiest to define it as a macro that can be used multiple times:

```
#if __BIG_ENDIAN__
#define ARGB_IMAGE_TYPE GL_UNSIGNED_INT_8_8_8_8_REV
#else
#define ARGB_IMAGE_TYPE GL_UNSIGNED_INT_8_8_8_8
#endif
/* later on, use it like this */
glTexImage2D (GL_TEXTURE_2D, 0, GL_RGB,
                 width, height, 0, GL_BGRA_EXT,
                 ARGB_IMAGE_TYPE, data);
```

Note that switching between `GL_UNSIGNED_INT_8_8_8_8_REV` and `GL_UNSIGNED_INT_8_8_8_8` works only for this particular 32-bit packed-pixel data type. For 16-bit ARGB data stored using `GL_UNSIGNED_SHORT_1_5_5_5_REV`, there is no corresponding byte swapped type. Keep in mind that `GL_UNSIGNED_SHORT_5_5_5_1` is not a replacement for `GL_UNSIGNED_SHORT_1_5_5_5_REV` on an Intel-based Macintosh computer. The format is interpreted as bit-order `arrrrrbbbbbggggg` on a big-endian system, and as bit order `ggrrrrrabbbbbggg` on a little-endian system.

3. If you can't use the previous approaches, you should either generate/load your data in the native endian format of the system and use the same pixel type on both architectures or use the `GL_UNPACK_SWAP_BYTES` pixel store setting to instruct OpenGL to swap the bytes of any texture loaded on a little-endian system. This setting applies to all texture or image calls made with the current OpenGL context, so it needs to be set only once per OpenGL context, for example:

```
#if __LITTLE_ENDIAN__
           glPixelStorei(GL_UNPACK_SWAP_BYTES, 1);
#endif
```

This method causes images that use the problematic formats to be loaded as they would be on PowerPC. You should consider this option only if no other option is available. Enabling this option causes OpenGL to use a slower rendering path than normal. Performance-sensitive OpenGL applications may be significantly slower with this option enabled than with it off. Although this method can get an OpenGL-based program up and running in as little time as possible, it is highly recommended that you use one of the other two methods.

> **Note:** Using the `GL_UNSIGNED_INT_8_8_8_8` format for `GL_BGRA` data is not necessarily faster than using `GL_UNPACK_SWAP_BYTES`. In some cases, performance decreases for rendering textures that use either of those two methods compared to using a data type such as `GL_UNSIGNED_INT_8_8_8_8_REV`. It's advisable that you use Shark or other tools to analyze the performance of your OpenGL code and make sure that you are not encountering particularly bad cases.

OSAtomic Functions

The kernel extension functions `OSDequeueAtomic` and `OSEnqueueAtomic` are not available on an Intel-based Macintosh.

For more information on these functions, see *Kernel Framework Reference*.

Pixel Data

Applications that store pixel data in memory using ARGB format must take care in how they read data. If the code is not written correctly, it's possible to misread the data; the result is colors or alpha that appear wrong.

If you see colors that appear wrong when your application runs on an Intel-based Macintosh computer, the following strategy may help you identify where pixel data is being read incorrectly.

Create a test image whose pixel data is easy to identify. For example, set each pixel so that alpha is `ff`, red is `aa`, green is `bb`, and blue is `cc`. Then read that image into your application. Figure 4-1 shows such an image.

Figure 4-1   A test image that can help locate the source of color problems

It's also helpful to go through your code and cast pixel data to the `unsigned char` data type.

Start with the portion of your code that reads the image. Use the following GDB command to examine the pixel data as hexadecimal bytes:

```
x/<number_bytes>xb <address of first byte>
```

This command prints the specified number of bytes, starting with the first byte of the first pixel. You should easily be able to see whether what's displayed onscreen matches the values of the pixels in the test image. If the values you see do not match the test image, then you've identified the misreading problem. If the values match, then you need to identify other portions of your code that modify or transform pixel data, and inspect the pixel data after each transformation.

PostScript Printing

If you are using the Carbon Printing Manager, note that the PICT with PostScript (`'pictwps'`) printing path is not available on Intel-based Macintosh computers except under Rosetta. If you need only to support EPS data you can use Quartz drawing together with the function `PMCGImageCreateWithEPSDataProvider` to allow the inclusion of EPS data as part of your Quartz drawing. If you need to generate the PostScript code for your application drawing you should use the function `PMPrinterPrintWithFile`.

Quartz Bitmap Data

The Quartz constants shown in Table 4-1 specify the byte ordering of pixel formats. These constants, which are defined in the `CGImage.h` header file, are used in the `bitmapInfo` parameter. To specify byte ordering to Quartz, use a bitwise `OR` operator to combine the appropriate constant with the `bitmapInfo` parameter.

Table 4-1   Quartz constants that specify byte ordering

Constant	Specifies
kCGBitmapByteOrderMask	The byte order mask
kCGBitmapByteOrder16Big	16-bit, big-endian format
kCGBitmapByteOrder32Big	32-bit, big-endian format
kCGBitmapByteOrder16Little	16-bit, little-endian format
kCGBitmapByteOrder32Little	32-bit, little-endian format
kCGBitmapByteOrder16Host	16-bit, host-endian format
kCGBitmapByteOrder32Host	32-bit, host-endian format

QuickDraw Routines

If you have existing code that directly accesses the `picFrame` field of the QuickDraw `Picture` data structure, you should use the QuickDraw function `QDGetPictureBounds` to get the appropriately swapped bounds for a `Picture`. This function is available in Mac OS X version 10.3 and later. Its prototype is as follows:

```
Rect * QDGetPictureBounds(
        PicHandle   picH,
        Rect        *outRect)
```

If you have existing code that uses the QuickDraw `DeltaPoint` function or the HIToolbox `PinRect` function (defined in `MacWindows.h`), make sure that you do not cast the function result to a `Point` data structure. The horizontal difference is returned in the low 16 bits, and the vertical difference is returned in the high 16 bits. You can obtain the horizontal and vertical values by using code similar to the following:

```
Point pointDiff;
SInt32 difference = DeltaPoint (p1, p2);
pointDiff.h = LoWord (difference);
pointDiff.v = HiWord (difference);
```

> **Tip:** The best solution is to convert your QuickDraw code to Quartz 2D. QuickDraw was deprecated starting in Mac OS X v10.4. For help with converting to Quartz 2D, see *Quartz Programming Guide for QuickDraw Developers*.

QuickTime Components

The Component Manager recognizes which architectures are supported by a component by looking at the `'thng'` resource for the component, not the architecture of the file. You must specify the appropriate architectures in the `'thng'` resource. To accomplish this, in the `.r` file where you define the `'thng'` resource, modify your `ComponentPlatformInfo` array to look similar to the following:

```
#if defined(__ppc__)
kMyComponentFlags, kMyCodeType, kMyCodeID, platformPowerPCNativeEntryPoint,
#endif
#if defined(__i386__)
kMyComponentFlags, kMyCodeType, kMyCodeID, platformIA32NativeEntryPoint,
#endif
```

Then, rebuild your component. For details, see "Building a Universal Binary" (page 13).

QuickTime Metadata Functions

When you call the function `QTMetaDataGetItemProperty` and the type of the key whose value you are retrieving is `code`, the data returned is an `OSType`, not a buffer of four characters. (You can determine the key type by calling the function `QTMetaDataGetItemPropertyInfo`.) To ensure that your code runs properly on both PowerPC and Intel-based Macintosh computers, you must use a correctly-typed buffer so that the endian format of the data returned to you is correct. If you supply a buffer of the wrong type, for example a buffer of `UInt8` instead of a buffer of `OSType`, the endian format of the data returned in the buffer will be wrong on Intel-based Macintosh Computers.

Runtime Code Generation

If your application generates code at runtime, keep in mind that the compiler assumes that the stack must be 16-byte aligned when calling into Mac OS X libraries or frameworks. 16-byte stack alignment is enforced on Intel-based Macintosh computers, which means that you need to ensure that your code is 16-byte aligned to avoid having your application crash.

For more information, see *Mac OS X ABI Function Call Guide*.

Spotlight Importers

A Spotlight importer is a plug-in bundle that extracts information from files created by an application. The Spotlight engine uses importers to gather information about new and existing files. Spotlight importers are not compatible with Rosetta. To run an importer on an Intel-based Macintosh as well as on a PowerPC-based Macintosh, you must compile it as a universal binary.

For more information on Spotlight, see *Spotlight Overview* and *Spotlight Importer Programming Guide*.

System-Specific Predefined Macros

The C preprocessor has several predefined macros whose purpose is to indicate the type of system and machine in use. If your code uses system-specific predefined macros, evaluate whether you really need to use them. In most cases applications need to know the capabilities available on a computer and not the specific system or machine on which the application is running. For example, if your application needs to know whether it is running on a little-endian or big-endian microprocessor, you should use the `__BIG_ENDIAN__` or `__LITTLE_ENDIAN__` macros or the Core Foundation function `CFByteOrderGetCurrent`. Do not use the `__i386__` and `__ppc__` macros for this purpose.

See *GNU C 4.0 Preprocessor User Guide* for additional information.

USB Device Access

USB uses little-endian format. If you are developing a universal binary version of an application that accesses a USB device, see "USB Device Access in an Intel-Based Macintosh" in *USB Device Interface Guide* for a discussion of the issues you may encounter.

See Also

In addition to the following resources, check the ADC website periodically for updates and technical notes that might address other specific situations:

■ *Quartz Programming Guide for QuickDraw Developers* which provides information on moving code from the deprecated QuickDraw API to Quartz

■ *IA-32 Intel Architecture Optimization Reference Manual*, available from:

http://developer.intel.com/design/pentium4/manuals/index_new.htm

Preparing Vector-Based Code

This chapter is relevant only for those developers who want to start writing vector-based code or whose applications already directly use the AltiVec extension to the PowerPC instruction set. AltiVec instructions, because they are processor specific, must be replaced on Intel-based Macintosh computers. You can choose from these two options:

- Use the Accelerate framework. This is the recommended option because the framework provides a layer of abstraction that lets you perform vector-based operations without needing to use low-level vector instructions yourself. See "Accelerate Framework" (page 63).

- Port AltiVec code to the Intel instruction set architecture (ISA). This solution is available for developers who have performance needs that can't be met by using the Accelerate framework. See "Rewriting AltiVec Instructions" (page 64).

Accelerate Framework

The Accelerate framework, introduced in Mac OS X v10.3 and expanded in v10.4, is a set of high-performance vector-accelerated libraries. You don't need to be concerned with the architecture of the target machine because the routines in this framework abstract the low-level details. The system automatically invokes the appropriate instruction set for the architecture that your code runs on.

This framework contains the following libraries:

- vImage is the Apple image processing framework that includes high-level functions for image manipulation—convolutions, geometric transformations, histogram operations, morphological transformations, and alpha compositing—as well as utility functions that convert formats and perform other operations. See *Optimizing Image Processing With vImage*.

- vDSP provides mathematical functions that perform digital signal processing (DSP) for applications such as speech, sound, audio, and video processing, diagnostic medical imaging, radar signal processing, seismic analysis, and scientific data processing. The vDSP functions operate on real and complex data types and include data type conversions, fast Fourier transforms (FFTs), and vector-to-vector and vector-to-scalar operations.

- vMathLib contains vector-accelerated versions of all routines in the standard math library. See *vecLib Framework Reference*.

- LAPACK is a linear algebra package that solves simultaneous sets of linear equations, tackles eigenvalue and singular solution problems, and determines least-squares solutions for linear systems.

■ BLAS (Basic Linear Algebra Subroutines) performs basic vector and matrix computations.

■ vForce contains routines that take matrices as input and output arguments, rather than single variables.

Rewriting AltiVec Instructions

Most of the tasks required to vectorize for AltiVec—restructuring data structures, designing parallel algorithms, eliminating branches, and so forth— are the same as those you'd need to perform for the Intel architecture. If you already have AltiVec code, you've already completed the fundamental vectorization work needed to rewrite your application for the Intel architecture. In many cases the translation process will be smooth, involving direct or nearly direct substitution of AltiVec intrinsics with Intel equivalents.

The MMX, SSE, SSE2, and SSE3 extensions provide analogous functionality to AltiVec. Like the AltiVec unit, these extensions are fixed-sized SIMD (Single Instruction Multiple Data) vector units, capable of a high degree of parallelism. Just as for AltiVec, code that is written to use the Intel ISA typically performs many times faster than scalar code.

Before you start rewriting AltiVec instructions for the Intel instruction set architecture, read *AltiVec/SSE Migration Guide*. It outlines the key differences between architectures in terms of vector-based programming, gives an overview of the SIMD extensions on x86, lists what you need to do to build your code, and provides an in-depth discussion on alignment and other relevant issues.

See Also

The following resources are relevant for rewriting AltiVec instructions for the Intel architecture:

■ "Architecture-Independent Vector-Based Code" (page 75) shows how to write a fast matrix-multiplication function with a minimum of architecture-specific coding.

■ Intel software manuals describe the x86 vector extensions:

http://developer.intel.com/design/Pentium4/documentation.htm

■ Perf-Optimization-dev is a list for discussions on analyzing and optimizing performance in Mac OS X. You can subscribe at:

http://lists.apple.com/mailman/listinfo/perfoptimization-devlists.apple.com

■ The SIMD website is aimed at developers who use SIMD microprocessor instructions:

http://www.simdtech.org/home

Rosetta

Rosetta is a translation process that runs a PowerPC binary on an Intel-based Macintosh computer—it allows applications to run as nonnative binaries. Many, but not all, applications can run translated. Applications that run translated will never run as fast as they run as a native binary because the translation process itself incurs a processing cost.

How compatible your application is with Rosetta depends on the type of application it is. An application such as a word processor that has a lot of user interaction and low computational needs is quite compatible. An application that requires a moderate amount of user interaction and has some high computational needs or that uses OpenGL is most likely also quite compatible. One that has intense computing needs isn't compatible. This includes applications that need to repeatedly compute fast Fourier transforms (FFTs), that compute complex models for 3-D modeling, or that compute ray tracing.

To the user, Rosetta is transparent. Unlike Classic, when the user launches an application, there aren't any visual cues to indicate that the application is translated. The user may perceive that the application is slow to start up or that the performance is slower than it is on a PowerPC-based Macintosh. The user can discover whether an application has only a PowerPC binary by looking at the Finder information for the application. (See "Determining Whether a Binary Is Universal" (page 18).)

This appendix discusses the sorts of applications that can run translated, describes how Rosetta works, points out special considerations for translated applications, shows how to force an application to run translated using Rosetta, describes how to programmatically detect whether an application is running nonnatively, and provides troubleshooting information if your application won't run translated but you think that it should.

What Can Be Translated?

Rosetta is designed to translate currently shipping applications that run on a PowerPC with a G3 or G4 processor and that are built for Mac OS X. That includes CFM as well as Mach-O PowerPC applications.

Rosetta does *not* run the following:

- Applications built for any version of the Mac OS earlier than Mac OS X —that means Mac OS 9, Mac OS 8, Mac OS 7, and so forth

- The Classic environment

- Screen savers written for the PowerPC architecture

- Code that inserts preferences in the System Preferences pane

- Applications that require a G5 processor

- Applications that depend on one or more PowerPC-only kernel extensions

- Kernel extensions

- Java applications with JNI libraries

- Java applets in applications that Rosetta can translate; that means a web browser that Rosetta can run translated will not be able to load Java applets.

Rosetta does not support precise exceptions. Any application that relies on register states being accurate in exception handlers or signal handlers will not function properly running with Rosetta.

For more information on the limitations of Java applications using Rosetta, see "Java Applications" (page 52) and *Technical Q &A QA1295, Java on Intel-based Macintosh Computers*, which is in the ADC Reference Library.

How It Works

When an application launches on an Intel-based Macintosh computer, the kernel detects whether the application has a native binary. If the binary is not native, the kernel launches the binary using Rosetta. If the application is one of those that can be translated, it launches and runs, although not as fast as it would as a native binary. Behind the scenes, Rosetta translates and executes the PowerPC binary code.

Rosetta runs in the same thread of control as the application. When Rosetta starts an application, it translates a block of application code and executes that block. As Rosetta encounters a call to a routine that it has not yet translated, it translates the needed routine and continues the execution. The result is a smooth and continual transitioning between translation and execution. In essence, Rosetta and your application work together in a kind of symbiotic relationship.

Rosetta optimizes translated code to deliver the best possible performance on the nonnative architecture. It uses a large translation buffer, and it caches code for reuse. Code that gets reused repeatedly in your application benefits the most because it needs to be translated only once. The system uses the cached translation, which is faster than translating the code again.

Special Considerations

Rosetta must run the entire process when it translates. This has implications for applications that use third-party plug-ins or any other component that must be loaded at the time your application launches. All parts (application, plug-ins, or other components needed at launch time) must run either nonnatively or natively. For example, if your application is built as a universal binary, but it uses a plug-in that has only a PowerPC binary, then your application needs to run nonnatively on an Intel-based Macintosh computer to use the nonnative plug in.

Rosetta takes endian issues into account when it translates your application. Multibyte data that moves between your application and any system process is automatically handled for you—you don't need to concern yourself with the endian format of the data.

The following kinds of multibyte data can have endian issues if the data moves between:

■ Your translated application and a native process that's not a system process

■ A custom pasteboard provided by your translated application and a custom pasteboard provided by a native application

■ Data files or caches provided by your translated application and a native application

You might encounter this scenario while developing a universal binary. For example, if you've created a universal binary for a server process that your application relies on, and then test that process by running your application as a PowerPC binary, the endian format of the data passed from the server to your application would be wrong. You encounter the same problem if you create a universal binary for your application, but have not yet done so for a server process needed by the application.

Structures that the system defines and that are written using system routines will work correctly. But consider the code in Listing A-1.

Listing A-1   A structure whose endian format depends on the architecture

```
typedef struct
{
    int x;
    int y;
} data_t

void savefile(data_t data, int filehandle)
{
  write(filehandle, &data, sizeof(data));
}
```

When run using Rosetta, the application will write a big-endian structure; *x* and *y* are both written as big-endian integers. When the application runs natively on an Intel-based Macintosh, it will write out a little-endian structure; *x* and *y* are written as little-endian integers. It is up to you to define data formats on disk to be of a canonical endian format. Endian-specific data formats are fine as long as any application that reads or write the data understands what the endian format of the data is and treats the data appropriately.

Keep in mind that private frameworks and plug-ins can also encounter these sorts of endian issues. If a private framework creates a cache or data file, and the framework is a universal binary, then it will try to access the cache from both native and PPC processes. The framework either needs to account for the endian format of the cache when reading or writing data or needs to have two separate caches.

Forcing an Application to Run Translated

Assuming that the application meets the criteria described in "What Can Be Translated?" (page 65), applications that have only a PowerPC binary automatically run as translated on an Intel-based Macintosh. For testing purposes, there are several ways that you can force applications that have a universal binary to launch as a PowerPC binary on an Intel-based Macintosh:

- For applications, "Make a Setting in the Info Window" (page 68)

- For command-line tools "Use Terminal" (page 68)

- For an application that you are writing, "Modify the Property List" (page 69)

- Programmatically, "Use the sysctlbyname Function" (page 69)

Each of these methods is described in this section.

Make a Setting in the Info Window

You can manually set which binary to execute on an Intel-based Macintosh computer by selecting the "Open using Rosetta" option in the Info window of the application. To set the option, click the application icon, then press Command-I to open the Info window. Make the setting, as shown in Figure A-1.

Figure A-1   The Info window for the Calculator application
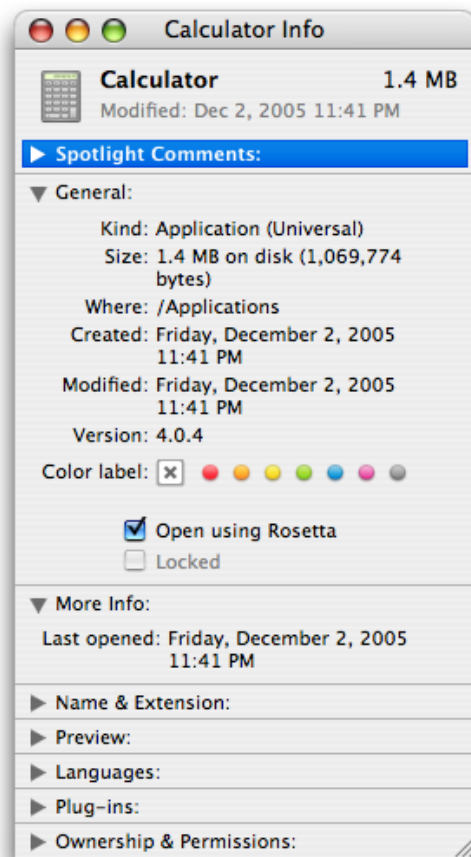
Use Terminal

You can force a command-line tool to run translated by entering the following in Terminal:

```
ditto -arch ppc <toolname> /tmp/toolname
/tmp/tool
```

Modify the Property List

You can set the default setting for the "Open using Rosetta" option by adding the following key to the `Info.plist` of your application bundle:

```
<key>LSPrefersPPC</key>
<true/>
```

This key informs the system that the application should launch as a PowerPC binary and causes the "Open using Rosetta" checkbox to be selected. You might find this useful if you ship an application that has plug-ins that are not native at the time of shipping.

Use the sysctlbyname Function

The `exec_affinity` routine in Listing A-2 controls the preferred CPU type for sublaunched processes. You might find this routine useful if you are using `fork` and `exec` to launch applications from your application.

The routine calls the `sysctlbyname` function with the `"sysctl.proc_exec_affinity"` string, passing a constant that specifies the CPU type. Pass `CPU_TYPE_POWERPC` to launch the PPC executable in a universal binary. (For information on `sysctlbyname` see *Mac OS X Man Pages*.)

Listing A-2   A routine that controls the preferred CPU type for sublaunched processes

```
cpu_type_t exec_affinity (cpu_type_t new_cputype)
{
    cpu_type_t ret;
    cpu_type_t *newp = NULL;
    size_t sz = sizeof (cpu_type_t);

    if (new_cputype != 0)
        newp = &new_cputype;

     if (sysctlbyname("sysctl.proc_exec_affinity",
            &ret, &sz, newp, newp ? sizeof(cpu_type_t) :  0) == -1) {
        fprintf(stderr, "exec_affinity: sysctlbyname failed:  %s\n",
                strerror(errno));
        return -1;
    }
    return ret;
}
```

Preventing an Application from Opening Using Rosetta

To prevent an application from opening using Rosetta, add the following key to the `Info.plist`:

```
<key>LSRequiresNativeExecution</key>
```

```
<true/>
```

Programmatically Detecting a Translated Application

Some developers may want to determine programmatically whether an application is running using Rosetta. For example, a developer writing device interface code may need to determine whether the user client is using the same endian format as the kernel.

Listing A-3 is a utility routine that can call the `sysctlbyname` function on a process ID (`pid`). If you pass a process ID of `0` to the routine, it performs the call on the current process. Otherwise it performs the call on the process specified by the `pid` value that you pass. (For information on `sysctlbyname` see *Mac OS X Man Pages*.) The `is_pid_native` routine shown in calls this routine, passing the string `"sysctl.proc_native"`.

Listing A-3   A utility routine for calling the `sysctlbyname` function

```
static int sysctlbyname_with_pid (const char *name, pid_t pid,
                                  void *oldp, size_t *oldlenp,
                                  void *newp, size_t newlen)
{
    if (pid == 0) {
        if (sysctlbyname(name, oldp, oldlenp, newp, newlen) == -1)  {
            fprintf(stderr, "sysctlbyname_with_pid(0): sysctlbyname  failed:"
                    "%s\n", strerror(errno));
            return -1;
        }
    } else {
        int mib[CTL_MAXNAME];
        size_t len = CTL_MAXNAME;
        if (sysctlnametomib(name, mib, &len) == -1) {
            fprintf(stderr, "sysctlbyname_with_pid: sysctlnametomib  failed:"
                    "%s\n", strerror(errno));
            return -1;
        }
        mib[len] = pid;
        len++;
        if (sysctl(mib, len, oldp, oldlenp, newp, newlen) == -1)  {
            fprintf(stderr, "sysctlbyname_with_pid: sysctl  failed:"
                    "%s\n", strerror(errno));
            return -1;
        }
    }
    return 0;
}
```

Listing A-4 shows a routine that determines whether the specified process is running natively or translated. The routine returns:

■   `0` if the process is running natively on an Intel-based Macintosh computer

■   `-1` if the process is not native but running translated using Rosetta

■   `1` if version of Mac OS X predates Rosetta, which means the process is native and running on a PowerPC Macintosh computer

Listing A-4   A routine that determines whether a process is running natively or translated

```
int is_pid_native (pid_t pid)
{
    int ret = 0;
    size_t sz = sizeof(ret);

    if (sysctlbyname_with_pid("sysctl.proc_native", pid,
                &ret, &sz, NULL, 0) == -1) {
        if (errno == ENOENT) {
            // sysctl doesn't exist, which means that this version of Mac OS
            // pre-dates Rosetta, so the application must be native.
            return 1;
        }
        fprintf(stderr, "is_pid_native: sysctlbyname_with_pid  failed:"
                "%s\n", strerror(errno));
        return -1;
    }
    return ret;
}
```

Troubleshooting

If you are convinced that your application falls into the category of those that should be able to run using Rosetta but it doesn't run or it has unexpected behavior, you can follow the procedure in this section to debug your application. This procedure works only for PowerPC binaries—not for a universal binary—and is the only way you can debug a PowerPC binary on an Intel-based Macintosh. Xcode debugging does not work for translated applications.

To debug a PowerPC binary on an Intel-based Macintosh, follow these steps:

1. Open Terminal.

2. Enter the following two lines:

    For `tcsh`:

    ```
    setenv OAH_GDB YES
    /<path>/<your_application>.app/Contents/MacOS/<your_application>
    ```
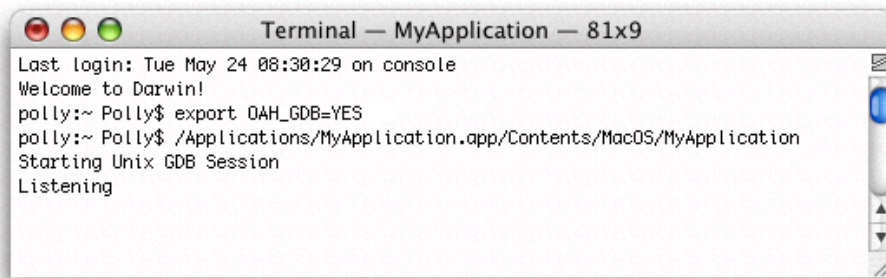
    For `bash`:

    ```
    export OAH_GDB=YES
    /<path>/<your_application>.app/Contents/MacOS/<your_application>
    ```

You'll see the Rosetta process launch and wait for a port connection (Figure A-2).

Figure A-2   Rosetta listens for a port connection


```
Last login: Tue May 24 08:30:29 on console
Welcome to Darwin!
polly:~ Polly$ export OAH_GDB=YES
polly:~ Polly$ /Applications/MyApplication.app/Contents/MacOS/MyApplication
Starting Unix GDB Session
Listening
```

3. Open a second terminal window and start up GDB with the following command:

    ```
    gdb --oah
    ```

    Using GDB on an Intel-based Macintosh computer is just like using GDB on a PowerPC Macintosh.

4. Attach your application.

    ```
    attach <your_application>
    ```

5. Press Tab.

    GDB automatically appends the process ID (`pid`) to your application name.

6. Press Return.
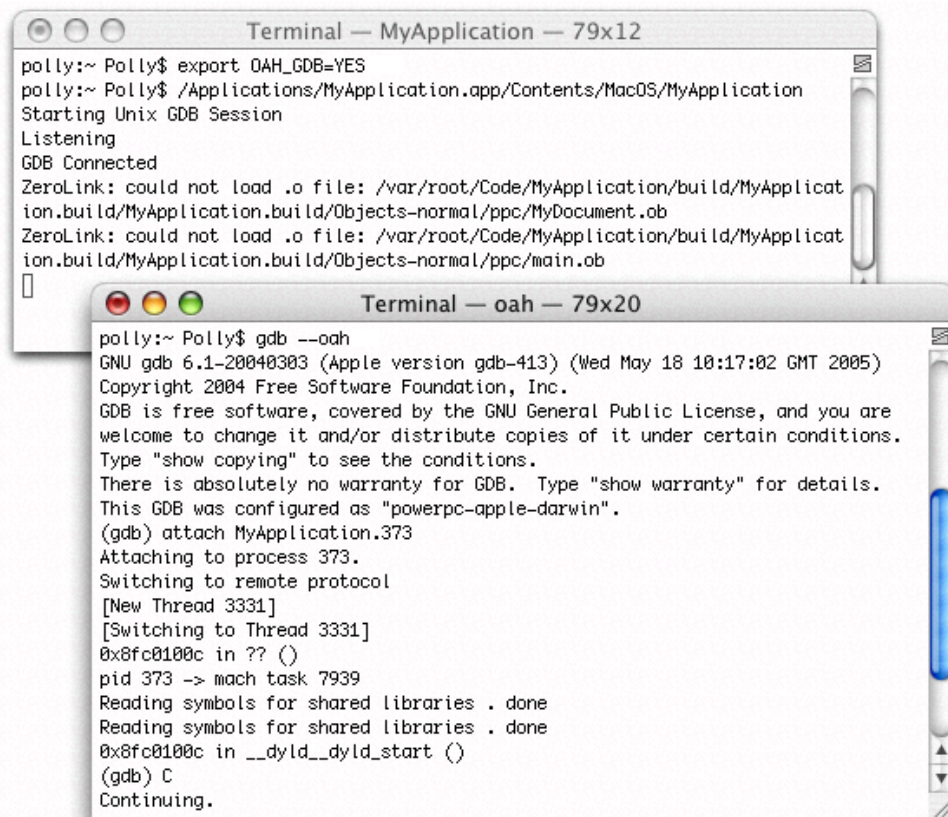
7. Type `c` to execute your application.

    > **Important:** Do not type `run`. Typing `run` will not execute your code. It will leave your application in a state that requires you to start over from the first step.

Figure A-3 shows the commands for initiating a debugging session for a PowerPC binary. After you start the session, you can debug in much the same way as you would debug a native process except that you can't call functions—either explicitly or implicitly—from within GDB. For example, you can't inspect CF objects by calling `CFShow`.

Keep in mind that symbol files aren't loaded at the start of the debugging session. They are loaded after your application is up and running. This means that any breakpoints you set are "pending breakpoints" until the executable and libraries are loaded.

Figure A-3   Terminal windows with the commands for debugging a PowerPC binary on an Intel-based
Macintosh computer

> **Note:** Debugging Rosetta applications from within either CodeWarrior or Xcode is not supported.

Architecture-Independent Vector-Based Code

The intention of this appendix is to show how to factor a mathematical calculation into architecture-independent and architecture-specific parts. Using matrix multiplication as an example, you'll see how to write a function that works for both the PowerPC and the x86 architectures with a minimum of architecture-specific coding. You can then apply this approach to other, more complex mathematical calculations.

The following basic operations are available on both architectures:

- Vector loads and stores
- Multiplication
- Addition
- An instruction to splat a float across a vector

For other types of calculations, you may need to write separate versions of code. Because of the differences in the number of registers and the pipeline depths between the two architectures, it is often advantageous to provide separate versions.

> **Note:** There is a function for 4x4 matrix multiplication in the Accelerate framework (`vecLib`) that is tuned for both architectures. You can also call `sgemm` from Basic Linear Algebra Subprograms (BLAS) (also available in the Accelerate framework) to operate on larger matrices.

Architecture-Specific Code

Listing B-1 (page 76) shows the architecture-specific code you need to support matrix multiplication. The code calls the architecture-independent function `MyMatrixMultiply`, which is shown in Listing B-2 (page 79). The code shown in Listing B-1 works properly for both instruction set architectures only if you build the code as a universal binary. For more information, see "Building a Universal Binary" (page 13).

> **Note:** The sample code makes use of a GCC extension to return a result from a code block ({}). The code may not compile correctly on other compilers. The extension is necessary because you cannot pass immediate values to an inline function, meaning that you must use a macro.

Listing B-1   Architecture-specific code needed to support matrix multiplication

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

// For each vector architecture...
#if defined( __VEC__ )
// AltiVec
    // Set up a vector type for a float[4] array for each vector  type
    typedef vector float vFloat;

    // Define some macros to map a virtual SIMD language to
    // each actual SIMD language. For matrix multiplication, the  tasks
    // you need to perform are essentially the same between the  two
    // instruction set architectures (ISA).
    #define vSplat( v, i )  ({ vFloat z = vec_splat( v, i );
                                    /* return */ z; })
    #define vMADD                   vec_madd
    #define vLoad( ptr )        vec_ld( 0, ptr )
    #define vStore( v, ptr )    vec_st( v, 0, ptr )
    #define vZero() (vector float) vec_splat_u32(0)

#elif defined( __SSE__ )
// SSE
    // The header file xmmintrin.h defines C functions for using
    // SSE and SSE2 according to the Intel C programming interface
    #include <xmmintrin.h>

    // Set up a vector type for a float[4] array for each vector  type
    typedef __m128  vFloat;

    // Also define some macros to map a virtual SIMD language to
    // each actual SIMD language.

    // Note that because i MUST be an immediate, it is incorrect  here
    // to alias i to a stack based copy and replicate that 4 times.
    #define vSplat( v, i )({ __m128 a = v; a = _mm_shuffle_ps( a,  a, \
                            _MM_SHUFFLE(i,i,i,i) ); /* return */  a; })
    inline __m128 vMADD( __m128 a, __m128 b, __m128 c )
    {
        return _mm_add_ps( c, _mm_mul_ps( a, b ) );
    }
    #define vLoad( ptr )    _mm_load_ps( (float*) (ptr) )
    #define vStore( v, ptr )    _mm_store_ps( (float*) (ptr),  v )

    #define vZero()             _mm_setzero_ps()

#else
// Scalar
    #warning To compile vector code, you must specify -faltivec,
                -msse, or both- faltivec and -msse
    #warning Compiling for scalar code.
```

```
    // Some scalar equivalents to show what the above vector
    // versions accomplish

    // A vector, declared as a struct with 4 scalars
    typedef struct
    {
        float               a;
        float               b;
        float               c;
        float               d;
    }vFloat;

    // Splat element i across the whole vector and return it
    #define vSplat( v, i )  ({ vFloat z; z.a = z.b = z.c = z.d =  ((float*)
                                &v)[i]; /* return */ z; })

 // Perform a fused-multiply-add operation on architectures that  support it
    // result = X * Y + Z
    inline vFloat vMADD( vFloat X, vFloat Y, vFloat Z )
    {
        vFloat result;

        result.a = X.a * Y.a + Z.a;
        result.b = X.b * Y.b + Z.b;
        result.c = X.c * Y.c + Z.c;
        result.d = X.d * Y.d + Z.d;

        return result;
    }

    // Return a vector that starts at the given address
    #define vLoad( ptr )  ( (vFloat*) ptr )[0]

    // Write a vector to the given address
    #define vStore( v, ptr )    ( (vFloat*) ptr )[0] = v

    // Return a vector full of zeros
    #define vZero()     ({ vFloat z; z.a = z.b = z.c = z.
                            d = 0.0f; /* return */ z; })

#endif

// Prototype for a vector matrix multiply function
void MyMatrixMultiply( vFloat A[4], vFloat B[4], vFloat C[4] );

int main( void )
{
    // The vFloat type (defined previously) is a vector or scalar  array
    // that contains 4 floats
    // Thus each one of these is a 4x4 matrix, stored in the C storage  order.
    vFloat          A[4];
    vFloat          B[4];
    vFloat          C1[4];
    vFloat          C2[4];
    int i, j, k;

    // Pointers to the elements in A, B, C1 and C2
```

```
    float *a = (float*) &A;
    float *b = (float*) &B;
    float *c1 = (float*) &C1;
    float *c2 = (float*) &C2;

    // Initialize the data
    for( i = 0; i < 16; i++ )
    {
        a[i] = (double) (rand() - RAND_MAX/2) / (double) (RAND_MAX  );
        b[i] = (double) (rand() - RAND_MAX/2) / (double) (RAND_MAX  );
        c1[i] = c2[i] = 0.0;
    }

    // Perform the brute-force version of matrix multiplication
    // and use this later to check for correctness
    printf( "Doing simple matrix multiply...\n" );
    for( i = 0; i < 4; i++ )
        for( j = 0; j < 4; j++ )
        {
            float result = 0.0f;

            for( k = 0; k < 4; k++ )
                result += a[ i * 4 + k] * b[ k * 4 + j ];

            c1[ i * 4 + j ] = result;
        }

    // The vector version
    printf( "Doing vector matrix multiply...\n" );
    MyMatrixMultiply( A, B, C2 );

    // Make sure that the results are correct
    // Allow for some rounding error here
    printf( "Verifying results..." );
    for( i = 0 ; i < 16; i++ )
        if( fabs( c1[i] - c2[i] ) > 1e-6 )
            printf( "failed at %i,%i: %8.17g %8.17g\n",  i/4,
                        i&3, c1[i], c2[i] );

    printf( "done.\n" );

    return 0;
}
```

The 4x4 matrix multiplication algorithm shown in is a simple matrix multiplication algorithm performed with four columns in parallel. The basic calculation is as follows:

```
C[i][j] = sum( A[i][k] * B[k][j], k = 0... width of A )
```

It can be rewritten in mathematical vector notation for rows of C as the following:

```
C[i][] = sum( A[i][k] * B[k][], k = 0... width of A )
```

Where:

> `C[i][]` is the *i*th row of C
>
> `A[i][k]` is the element of *A* at row *i* and column *k*

`B[k][]` is the *k*th row of *B*

An example calculation for `C[0][]` is as follows:

```
C[0][] = A[0][0] * B[0][] + A[0][1] * B[1][] + A[0][2] * B[2][] + A[0][3] * B[3][]
```

This calculation is simply a multiplication of a scalar times a vector, followed by addition of similar elements between two vectors, repeated four times, to get a vector that contains four sums of products. Performing the calculation in this way saves you from transposing B to obtain the B columns, and also saves you from adding across vectors, which is inefficient. All operations occur between similar elements of two different vectors.

Architecture-Independent Matrix Multiplication

Listing B-2 (page 79) shows architecture-independent vector code that performs matrix multiplication. This code compiles as scalar if you do not set up the appropriate compiler flags for PowerPC (`-faltivec`) or x86 (`-msse`), or if AltiVec is unavailable on the PowerPC. The matrices used in the `MyMatrixMultply` function assume the C storage order for 2D arrays, not the FORTRAN storage order.

Listing B-2   Architecture-independent code that performs matrix multiplication

```
void MyMatrixMultiply( vFloat A[4], vFloat B[4], vFloat C[4] )
{
    vFloat A1 = vLoad( A );        //Row 1 of A
    vFloat A2 = vLoad( A + 1 );    //Row 2 of A
    vFloat A3 = vLoad( A + 2 );    //Row 3 of A
    vFloat A4 = vLoad( A + 3);     //Row 4 of A
    vFloat C1 = vZero();          //Row 1 of C, initialized to zero

    vFloat C2 = vZero();          //Row 2 of C, initialized to zero

    vFloat C3 = vZero();          //Row 3 of C, initialized to zero

    vFloat C4 = vZero();          //Row 4 of C, initialized to zero


    vFloat B1 = vLoad( B );        //Row 1 of B
    vFloat B2 = vLoad( B + 1 );    //Row 2 of B
    vFloat B3 = vLoad( B + 2 );    //Row 3 of B
    vFloat B4 = vLoad( B + 3);     //Row 4 of B

    //Multiply the first row of B by the first column of A (do not  sum across)
    C1 = vMADD( vSplat( A1, 0 ), B1, C1 );
    C2 = vMADD( vSplat( A2, 0 ), B1, C2 );
    C3 = vMADD( vSplat( A3, 0 ), B1, C3 );
    C4 = vMADD( vSplat( A4, 0 ), B1, C4 );

    // Multiply the second row of B by the second column of A and
    // add to the previous result (do not sum across)
    C1 = vMADD( vSplat( A1, 1 ), B2, C1 );
    C2 = vMADD( vSplat( A2, 1 ), B2, C2 );
    C3 = vMADD( vSplat( A3, 1 ), B2, C3 );
    C4 = vMADD( vSplat( A4, 1 ), B2, C4 );
```

```
    // Multiply the third row of B by the third column of A and
    // add to the previous result  (do not sum across)
    C1 = vMADD( vSplat( A1, 2 ), B3, C1 );
    C2 = vMADD( vSplat( A2, 2 ), B3, C2 );
    C3 = vMADD( vSplat( A3, 2 ), B3, C3 );
    C4 = vMADD( vSplat( A4, 2 ), B3, C4 );

    // Multiply the fourth row of B by the fourth column of A and
    // add to the previous result (do not sum across)
    C1 = vMADD( vSplat( A1, 3 ), B4, C1 );
    C2 = vMADD( vSplat( A2, 3 ), B4, C2 );
    C3 = vMADD( vSplat( A3, 3 ), B4, C3 );
    C4 = vMADD( vSplat( A4, 3 ), B4, C4 );

    // Write out the result to the destination
    vStore( C1, C );
    vStore( C2, C + 1 );
    vStore( C3, C + 2 );
    vStore( C4, C + 3 );
}
```

Application Binary Interface

Mac OS X ABI Function Call Guide describes the function-calling conventions used in all the architectures supported by Mac OS X. For detailed information about the IA-32 ABI, read the section "IA-32 Function Calling Conventions," which:

- Lists data types, sizes, and natural alignment

- Describes stack structure

- Discusses prologs and epilogs

- Provides details on how arguments are passed and results are returned

- Tells which registers preserve their value after a procedure call and which ones are volatile

Using PowerPlant

You can use PowerPlant on an Intel-based Macintosh computer only after you make the changes detailed in this section. Most of the substantial changes that you need to make are to the PowerPlant `LStream` class. PowerPlant uses `LStream` for a number of different I/O tasks, of which the most important is reading to and writing from `'PPob'` resources. "Changing LStream Code" (page 83) provides a detailed description of all the `LStream`-related changes that are required.

Applications using PowerPlant should also set the `CSResourcesFileMapped` flag to `false` in the `Info.plist` file. This is because the PowerPlant `UTextTraits.cp` file contains a method that writes to a dereferenced handle, as shown in the following code line. This handle is obtained from the `GetResource` function.

```
void UTextTraits::LoadTextTraits (TextTraitsPtr ioTextTraits)
```

If you use the `LDataBrowser` PowerPlant class and the `'DBC#'` resource, you might also need to write a resource flipper. See "Flipping the 'DBC#' Resource Type" (page 91).

Changing LStream Code

The changes discussed in this section cause `LStream` to always read and write big-endian data. You have the following outcomes after you make these changes:

■ `PPob` resources will work correctly on both PowerPC-based and Intel-based Macintosh computers, with the possible exception of custom types used by your application. Custom types will also work correctly if you ensure that the `LStream` constructors used by your custom types employ the streaming methods of `LStream` rather than the byte-manipulation methods.

■ Data read from and written to disk or to the network and that is written with `LStream` will be portable between PowerPC-based and Intel-based Macintosh computers, with the exception of calls to the methods shown in Listing D-1. For calls to those methods, your code should either swap bytes to big-endian format or switch to using the streaming methods of `LStream`.

Make sure that you evaluate all calls to the methods in Listing D-1 (including calls to any subclassed versions of these methods) to see if they require byte swapping.

Listing D-1   Calls that may require swapping bytes

```
LStream::PutBytes
LStream::WriteData
```

```
LStream::WriteBlock
LStream::operator << (Handle inHandle)
LStream::GetBytes
LStream::ReadData
LStream::ReadBlock
LStream::PeekData
LStream::operator >> (Handle &outHandle)
LStream::WritePtr
LStream::ReadPtr
LStream::WriteHandle
LStream::ReadHandle
```

If you have custom classes in a `PPob` resource, you should change their `LStream` constructors to avoid calling `LStream::ReadData`. Specifically, you need to change the code in the same way as the `LStream` constructors for `LControl`, `LListBox`, and other PowerPlant classes are changed in the following sections. The sections, organized by the files you need to change, describe the required code changes:

LStream.h

Change the `operator << (const Rect &inRect)` method from a single `WriteBlock` call to the following:

```
Rect rect;
rect.top = CFSwapInt16HostToBig(inRect.top);
rect.left = CFSwapInt16HostToBig(inRect.left);
rect.right= CFSwapInt16HostToBig(inRect.right);
rect.bottom = CFSwapInt16HostToBig(inRect.bottom);
WriteBlock(&rect, sizeof(rect));
```

Change the `operator << (const Point &inPoint)` method from a single `WriteBlock` call to the following:

```
Point pt;
pt.v = CFSwapInt16HostToBig(inPoint.v);
pt.h = CFSwapInt16HostToBig(inPoint.h);
WriteBlock(&pt, sizeof(pt));
```

Change the `operator << (SInt16 inNum)` method from a single `WriteBlock` call to the following:

```
SInt16 n;
n = CFSwapInt16HostToBig(inNum);
WriteBlock(&n, sizeof(n));
```

Change the `operator << (UInt16 inNum)` method from a single `WriteBlock` call to the following:

```
UInt16 n;
n = CFSwapInt16HostToBig(inNum);
WriteBlock(&n, sizeof(n));
```

Change the `operator << (SInt32 inNum)` method from a single `WriteBlock` call to the following:

```
SInt32 n;
n = CFSwapInt32HostToBig(inNum);
WriteBlock(&n, sizeof(n));
```

Change the `operator << (UInt32 inNum)` method from a single `WriteBlock` call to the following:

```
UInt32 n;
n = CFSwapInt32HostToBig(inNum);
WriteBlock(&n, sizeof(n));
```

Change the `operator << (float inNum)` method from a single `WriteBlock` call to the following:

```
CFSwappedFloat32 swappedFloat;
swappedFloat = CFConvertFloat32HostToSwapped(inNum);
WriteBlock(&swappedFloat, sizeof(swappedFloat));
```

Change the `operator << (bool inBool)` method from a single `WriteBlock` call to the following:

Note: This code writes 4-byte `bool` values. If you need compatibility with existing resources, you may need to modify the code to write 1-byte `bool` values.

```
UInt32 boolValue;
boolValue = CFSwapInt32HostToBig(inBool);
WriteBlock(&boolValue, sizeof(boolValue));
```

In the `operator >> (Rect &outRect)` method, add this after the `ReadBlock` call:

```
outRect.top = CFSwapInt16BigToHost(outRect.top);
outRect.left = CFSwapInt16BigToHost(outRect.left);
outRect.right= CFSwapInt16BigToHost(outRect.right);
outRect.bottom = CFSwapInt16BigToHost(outRect.bottom);
```

In the `operator >> (Point &outPoint)` method, add the following after the `ReadBlock` call:

```
outPoint.v = CFSwapInt16BigToHost(outPoint.v);
outPoint.h = CFSwapInt16BigToHost(outPoint.h);
```

In the `operator >> (SInt16 &outNum)` method, add the following after the `ReadBlock` call:

```
outNum = CFSwapInt16BigToHost(outNum);
```

In the `operator >> (UInt16 &outNum)` method, add the following after the `ReadBlock` call:

```
outNum = CFSwapInt16BigToHost(outNum);
```

In the `operator >> (SInt32 &outNum)` method, add the following after the `ReadBlock` call:

```
outNum = CFSwapInt32BigToHost(outNum);
```

In the `operator >> (UInt32 &outNum)` method, add the following after the `ReadBlock` call:

```
outNum = CFSwapInt32BigToHost(outNum);
```

In the `operator >> (float &outNum)` method, replace the `ReadBlock` call with the following:

```
CFSwappedFloat32 swappedFloat;
ReadBlock(&swappedFloat, sizeof(swappedFloat));
outNum = CFConvertFloat32SwappedToHost(swappedFloat);
```

In the `operator >> (bool &outBool)` method, replace the `ReadBlock` call with the following:

Note: This code reads 4-byte `bool` values. If you need compatibility with existing resources, you may need to modify the code to read 1-byte `bool` values.

```
UInt32 boolValue;
ReadBlock(&boolValue, sizeof(boolValue));
outBool = CFSwapInt32BigToHost(boolValue);
```

LStream.cp

In the `operator << (double inNum)` method, change the `#if TARGET_CPU_PPC` block to the following:

```
#if TARGET_CPU_PPC || TARGET_CPU_X86
// PowerPC and Intel doubles -- they're 8 bytes already, so just  swap
// if necessary and write.

Assert_(sizeof(inNum) == 8);
CFSwappedFloat64 swappedDouble = CFConvertDoubleHostToSwapped(inNum);
WriteBlock(&swappedDouble, sizeof(swappedDouble));
```

In the `operator >> (double& outNum)` method, change the `#if TARGET_CPU_PPC` block to the following:

```
#if TARGET_CPU_PPC || TARGET_CPU_X86
// PowerPC and Intel doubles -- they're 8 bytes already, so just  read
// and swap if necessary.

Assert_(sizeof(outNum) == 8);
CFSwappedFloat64 swappedDouble;
ReadBlock(&swappedDouble, sizeof(swappedDouble));
outNum = CFConvertDoubleSwappedToHost(swappedDouble);
```

LControl.cp

In the `LStream` constructor, replace this line:

```
inStream->ReadData(&controlInfo, sizeof(SControlInfo));
```

with the following lines:

```
*inStream >> controlInfo.valueMessage;
*inStream >> controlInfo.value;
*inStream >> controlInfo.minValue;
*inStream >> controlInfo.maxValue;
```

LListBox.cp

In the `LStream` constructor, replace this line:

```
inStream->ReadData(&listInfo, sizeof(SListBoxInfo));
```

with the following lines:

```
*inStream >> listInfo.hasHorizScroll;
*inStream >> listInfo.hasVertScroll;
*inStream >> listInfo.hasGrow;
*inStream >> listInfo.hasFocusBox;
*inStream >> listInfo.doubleClickMessage;
*inStream >> listInfo.textTraitsID;
*inStream >> listInfo.LDEFid;
*inStream >> listInfo.numberOfItems;
```

In the `RestorePlace(LStream *inPlace)` method, replace the following line:

```
 inPlace->ReadData(&theRect, sizeof(Rect));
```

with this line:

```
*inPlace >> theRect;
```

Further down in the method, just under the `if (vScroll != nil)` line, replace this line:

```
inPlace->ReadData(&theRect, sizeof(Rect));
```

with this line:

```
*inPlace >> theRect;
```

And once more in the same method, just under the `if (hScroll != nil)` line, replace this line:

```
inPlace->ReadData(&theRect, sizeof(Rect));
```

with this line:

```
*inPlace >> theRect;
```

LPane.cp

In the `LStream` constructor, replace this line:

```
inStream->ReadData(&thePaneInfo, sizeof(SPaneInfo));
```

with the following lines:

```
SInt32 viewPtr;
*inStream >> thePaneInfo.paneID;
*inStream >> thePaneInfo.width;
*inStream >> thePaneInfo.height;
*inStream >> thePaneInfo.visible;
*inStream >> thePaneInfo.enabled;
*inStream >> thePaneInfo.bindings.left;
*inStream >> thePaneInfo.bindings.top;
*inStream >> thePaneInfo.bindings.right;
*inStream >> thePaneInfo.bindings.bottom;
*inStream >> thePaneInfo.left;
*inStream >> thePaneInfo.top;
*inStream >> thePaneInfo.userCon;
*inStream >> viewPtr;
thePaneInfo.superView = reinterpret_cast<LView *>(viewPtr);
```

LPrintout.cp

In the `LStream` constructor, replace this line:

```
inStream->ReadData(&thePrintoutInfo, sizeof(SPrintoutInfo));
```

with the following lines:

```
*inStream >> thePrintoutInfo.width;
*inStream >> thePrintoutInfo.height;
*inStream >> thePrintoutInfo.active;
*inStream >> thePrintoutInfo.enabled;
*inStream >> thePrintoutInfo.userCon;
*inStream >> thePrintoutInfo.attributes;
```

LScroller.cp

In the `LStream` constructor, replace this line:

```
inStream->ReadData(&scrollerInfo, sizeof(SScrollerInfo));
```

with the following lines:

```
*inStream >> scrollerInfo.horizBarLeftIndent;
*inStream >> scrollerInfo.horizBarRightIndent;
*inStream >> scrollerInfo.vertBarTopIndent;
*inStream >> scrollerInfo.vertBarBottomIndent;
*inStream >> scrollerInfo.scrollingViewID;
```

LTable.cp

In the `LStream` constructor, replace this line:

```
inStream->ReadData(&tableInfo, sizeof(STableInfo));
```

with the following lines:

```
*inStream >> tableInfo.numberOfRows;
*inStream >> tableInfo.numberOfCols;
*inStream >> tableInfo.rowHeight;
*inStream >> tableInfo.colWidth;
*inStream >> tableInfo.cellDataSize;
```

LView.cp

In the `LStream` constructor, replace this line:

```
inStream->ReadData(&viewInfo, sizeof(SViewInfo));
```

with the following lines:

```
*inStream >> viewInfo.imageSize.width;
*inStream >> viewInfo.imageSize.height;
*inStream >> viewInfo.scrollPos.h;
*inStream >> viewInfo.scrollPos.v;
*inStream >> viewInfo.scrollUnit.h;
*inStream >> viewInfo.scrollUnit.v;
*inStream >> viewInfo.reconcileOverhang;
```

LWindow.cp

In the `LStream` constructor, replace this line:

```
inStream->ReadData(&windowInfo, sizeof(SWindowInfo));
```

with the following lines:

```
*inStream >> windowInfo.WINDid;
*inStream >> windowInfo.layer;
*inStream >> windowInfo.attributes;
*inStream >> windowInfo.minimumWidth;
*inStream >> windowInfo.minimumHeight;
*inStream >> windowInfo.maximumWidth;
*inStream >> windowInfo.maximumHeight;
*inStream >> windowInfo.standardSize.width;
*inStream >> windowInfo.standardSize.height;
*inStream >> windowInfo.userCon;
```

LPopupGroupBox.cp

In the `LStream` constructor, replace this line:

```
inStream->ReadData(&cInfo, sizeof(SControlInfo));
```

with the following lines:

```
*inStream >> cInfo.valueMessage;
*inStream >> cInfo.value;
*inStream >> cInfo.minValue;
*inStream >> cInfo.maxValue;
```

LControlView.cp

In the `LStream` constructor, replace this line:

```
inStream->ReadData(&cInfo, sizeof(SControlInfo));
```

with the following lines:

```
*inStream >> cInfo.valueMessage;
*inStream >> cInfo.value;
*inStream >> cInfo.minValue;
*inStream >> cInfo.maxValue;
```

LScrollerView.cp

In the `LStream` constructor, replace this line:

```
inStream->ReadData(&scrollerInfo, sizeof(SScrollerViewInfo));
```

with the following lines:

```
*inStream >> scrollerInfo.horizBarLeftIndent;
*inStream >> scrollerInfo.horizBarRightIndent;
*inStream >> scrollerInfo.vertBarTopIndent;
*inStream >> scrollerInfo.vertBarBottomIndent;
*inStream >> scrollerInfo.scrollingViewID;
```

LPageController.cp

In the `LStream` constructor, replace these lines:

```
inStream->ReadData( &mBackColor, sizeof(RGBColor));
inStream->ReadData( &mFaceColor, sizeof(RGBColor));
inStream->ReadData( &mPushedTextColor, sizeof(RGBColor));
```

with the following lines:

```
*inStream >> mBackColor.red;
```

```
*inStream >> mBackColor.green;
*inStream >> mBackColor.blue;
*inStream >> mFaceColor.red;
*inStream >> mFaceColor.green;
*inStream >> mFaceColor.blue;
*inStream >> mPushedTextColor.red;
*inStream >> mPushedTextColor.green;
*inStream >> mPushedTextColor.blue;
```

Flipping the 'DBC#' Resource Type

If you use the `LDataBrowser` PowerPlant class, you might be using a `'DBC#'` resource. If so, you will need to write a resource flipper similar to that shown in Listing D-2.

Listing D-2   Code that flips the `'DBC#'` resource type

```
struct DataBrowserColumnSetup {
    DataBrowserTableViewColumnID    propertyID;        // UInt32
    DataBrowserPropertyType         propertyType;      // unsigned  long
    SInt32                          nameStrIndex;
    DataBrowserPropertyFlags        propertyFlags;     // UInt32
    UInt16                          minimumWidth;
    UInt16                          maximumWidth;
    UInt16                          initialWidth;
    ControlContentType              btnContentType;    // SInt16
    ControlButtonGraphicAlignment   btnContentAlign;   // SInt16
    SInt16                          btnContentDataID;
    ControlButtonGraphicAlignment   titleAlignment;    // SInt16
    ControlButtonTextPlacement      titlePlacement;    // SInt16
    SInt16                          titleFontTypeID;
    SInt16                          titleFontStyle;
    SInt16                          titleFontSize;
    UInt16                          titleOffset;
};
typedef struct DataBrowserColumnSetup   DataBrowserColumnSetup;

OSStatus FlipDBC_(OSType dataDomain, OSType dataType,
                short id, void* dataPtr, UInt32 dataSize,
                Boolean currentlyNative, void* refcon)
{
    DataBrowserColumnSetup *dbc;
    UInt16 count;
    int i;

    UInt16* countPtr = (UInt16*) dataPtr;

    if (currentlyNative) {
        count = * countPtr;
        *countPtr = EndianU16_NtoB(count);
        dataPtr = (void*)&countPtr[1];
    } else {
        *countPtr = EndianU16_BtoN(*countPtr);
        count = *countPtr;
        dataPtr = (void*) &countPtr[1];
```

```
    }

    dbc = (DataBrowserColumnSetup *) dataPtr;
    for(i = 0; i < count; i++, dbc++) {
        dbc->propertyID = Endian32_Swap(dbc->propertyID);
        dbc->propertyType = Endian32_Swap(dbc->propertyType);
        dbc->nameStrIndex = Endian32_Swap(dbc->nameStrIndex);
        dbc->propertyFlags = Endian32_Swap(dbc->propertyFlags);
        dbc->minimumWidth = Endian16_Swap(dbc->minimumWidth);
        dbc->maximumWidth = Endian16_Swap(dbc->maximumWidth);
        dbc->initialWidth = Endian16_Swap(dbc->initialWidth);
        dbc->btnContentType = Endian16_Swap(dbc->btnContentType);
        dbc->titleAlignment = Endian16_Swap(dbc->titleAlignment);
        dbc->titlePlacement = Endian16_Swap(dbc->titlePlacement);
        dbc->titleFontTypeID = Endian16_Swap(dbc->titleFontTypeID);
        dbc->titleFontStyle = Endian16_Swap(dbc->titleFontStyle);
        dbc->titleFontSize = Endian16_Swap(dbc->titleFontSize);
        dbc->titleOffset = Endian16_Swap(dbc->titleOffset);
    }
    return noErr;
}
```

Document Revision History

This table describes the changes to *Universal Binary Programming Guidelines, Second Edition*.

Date	Notes
2006-07-24	Made a few minor technical corrections.
	Revised "Network-Related Data" (page 36).
	Clarified how Listing A-4 (page 71) works.
2006-06-28	Fixed link.
	Added "PostScript Printing" (page 58).
	Redirected link from Kernel Extensions Reference to *Kernel Framework Reference*.
2006-05-23	Removed outdated links and made a few other minor changes.
	Revised code in "Flipping the 'DBC#' Resource Type" (page 91) to use an explicit `UInt16` pointer and to assign back to `dataptr` the advanced `countPtr`.
	Updated instructions in "Troubleshooting" (page 71).
	Added information about the `CCSResourcesFileMapped` flag to "Using PowerPlant" (page 83).
	Removed links to documentation that is no longer relevant.
	Added a note to "LStream.h" (page 84) concerning reading and writing `bool` values.
2006-04-04	Corrected two function names.
	Revised information in "Application Binary Interface" (page 81) so that it now only provides a link to the primary ABI reference.
2006-03-08	Improved wording and added information on Spotlight importers.

93

Date	Notes
	Added information to "Objective-C Runtime: Sending Messages" and "Objective-C: Messages to nil".
2006-02-07	Improved the wording in several sections.
	Revised wording in "Bit Shifting" (page 48), "Bit Test, Set, and Clear Functions: Carbon and POSIX" (page 48), "Troubleshooting" (page 71), and "Guidelines for Swapping Bytes" (page 31).
	Revised code in Listing A-4 (page 71) by adding a statement to handle versions of Mac OS that pre-date Rosetta.
2006-01-10	Updated content for Mac OS X v10.4.4.
	Removed the note about preliminary documentation from "Introduction to Universal Binary Programming Guidelines" (page 9).
	Changed Xcode 2.1 to Xcode 2.2 in various places throughout the document because this is the recommended version for building a universal binary.
	Updated screenshots.
	Updated information in "Disk Partitions" (page 49), "Finder Information and Low-Level File System Operations" (page 50), "Multithreading" (page 54), "Objective-C: Messages to nil" (page 54), "QuickTime Components" (page 60), "Runtime Code Generation" (page 60), and "Values in an Array" (page 39).
	Added the sections "Code on the Stack: Disabling Execution" (page 24), "Extensible Firmware Interface (EFI)" (page 26), and "Mach Processes: The Task for PID Function" (page 53).
	In "Rosetta" (page 65), updated the sections "What Can Be Translated?" (page 65) and "Forcing an Application to Run Translated" (page 67).
	In "Rosetta" (page 65), added the section "Programmatically Detecting a Translated Application " (page 70).
2005-12-06	Made refinements to existing content.
	Added code that shows how to swap bytes for values in an array. See "Values in an Array" (page 39).
	Added "Automator Scripts" (page 48), "Dashboard Widgets" (page 49), and "QuickTime Metadata Functions" (page 60).
2005-11-09	Updated for Xcode 2.2; includes pointers to newly revised tools documentation as well as improved guidelines and tips.
	Revised "Building Your Code" (page 14).
	Added "Debugging" (page 17).

Date	Notes
	Added information to "Pixel Data " (page 58) on how to track down color problems.
	Added the section "Quartz Bitmap Data" (page 59).
	Added information about IP addresses and other "false" numerical values.
	In several places throughout the book, added cross references to newly revised, relevant documentation.
	Added clarification on the `long double` data type. See "Data Types" (page 25).
	Added information about using the `PinRect` function. See "QuickDraw Routines" (page 59).
	Added information about the need for Xcode targets to be native. See "Build Assumptions" (page 13) and "Building Your Code" (page 14).
	Corrected information about how ATS for Fonts handles font resources. See "Font-Related Resources" (page 51).
	Changed extended markup language to extensible markup language.
	Improved the grammar in "Objective-C: Messages to nil" (page 54).
	Fixed a link to information on Hyper-Threading Technology. See the "See Also" (page 61) section in "Guidelines for Specific Scenarios" (page 47).
	Made numerous editorial changes throughout.
2005-10-04	Made technical improvements and minor editorial changes throughout.
	Added a few resources to See Also in "Building a Universal Binary" (page 13).
	Changed the title of the Appendix Fast Matrix Multiplication to "Architecture-Independent Vector-Based Code" (page 75).
	Added new sections to the chapter "Guidelines for Specific Scenarios" (page 47). See "FireWire Device Access" (page 51) and "USB Device Access" (page 61).
	Added information about a relevant technical note to "QuickTime Components" (page 60).
	Added an example of a color issue to "Troubleshooting Your Built Application" (page 17).
	Revised the section "Objective-C: Messages to nil" (page 54).
	Revised the code for swapping floating-point values. See "Floating-Point Values" (page 34).

95

Date	Notes
	Add a reference to *Cross-Development Programming Guide* in the chapter "Building a Universal Binary" (page 13).
	Made corrections to the section "OpenGL" (page 55).
2005-09-08	Updated a substantial amount of task and conceptual information.
	Completely replaced information related to PowerPlant. See "Using PowerPlant" (page 83).
	Removed most of the content from "Preparing Vector-Based Code" (page 63) because the document *AltiVec/SSE Migration Guide* provides a more complete discussion of porting AltiVec code to SSE.
	Removed most of the content from the appendix titled *Application Binary Interface* because the document *Mac OS X ABI Function Call Guide* provides a more complete description of the IA-32 ABI for Intel-based Macintosh computers.
	Added a section—"Java Applications" (page 52)—that provides information about Java on Intel-based Macintosh computers, including what happens under Rosetta. Added cross-references to a technical note on this topic to "Rosetta" (page 65).
2005-08-11	Numerous minor technical and editorial changes throughout.
	Removed the appendix titled *x86 Equivalent Instructions for AltiVec Instructions.*"
2005-07-07	Made numerous minor technical refinements and fixed a few typographical errors.
2005-06-17	Fixed typographical and linking errors. Made several improvements to technical content.
2005-06-07	New document that describes the architectural differences between PowerPC and Intel and provides tips for writing code that can run on both.