64-Bit Transition Guide

General

2010-01-15

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

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

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

Simultaneously published in the United States and Canada.

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

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

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

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

Contents

Tables and Listings

Introduction to 64-Bit Transition Guide

This document describes the 64-bit features that are available in Mac OS X v10.4 and v10.5. You should read it to help you determine which of these features to use and how to use them.

What Is 64-Bit Computing?

For the purposes of this document, 64-bit computing is defined as support for a 64-bit address space—that is, support for concurrent use of more than 4 GB of memory by a single executable program—no more, no less.

Beginning with version 10.4, Mac OS X supports command-line 64-bit executables on G5-based Macintosh computers and 64-bit-capable Intel Macintosh computers.

Beginning with version 10.5, Mac OS X supports full-featured 64-bit applications on G5-based and 64-bit-capable Intel Macintosh computers.

Beginning with Snow Leopard, Mac OS X uses a 64-bit kernel on some Intel computers.

Who Should Read This Document?

Mac OS X application developers should, at a minimum, read the chapter "Should You Recompile Your Software as a 64-Bit Executable?" (page 11). That chapter will help you determine whether it makes sense for your application to take advantage of 64-bit application support in Mac OS X v10.5 and later.

If you are *not* planning to update your application to include a 64-bit executable, the 64-bit changes in Mac OS X v10.5 and later will probably have no impact on you. The advantages and disadvantages of compiling your code as a 64-bit executable are discussed in more detail in the chapter "Should You Recompile Your Software as a 64-Bit Executable?" (page 11).

Developers of device drivers and kernel extensions should also read this document. Beginning with Snow Leopard, device drivers and kernel extensions must be compiled with a 64-bit slice to be loadable into a 64-bit kernel.

Organization of This Document

This document is organized into the following chapters:

- "Should You Recompile Your Software as a 64-Bit Executable?" (page 11)—provides helpful guidance about whether you should recompile your application as a 64-bit executable.

- "Major 64-Bit Changes" (page 19)—describes the high-level architectural changes between a 32-bit and 64-bit environment.
- "Making Code 64-Bit Clean" (page 23)—explains the general changes needed to make an application 64-bit clean.
- "Compiling 64-Bit Code" (page 37)—explains how to compile your application as a 64-bit executable.
- "High-Level 64-Bit API Support" (page 41)—summarizes changes to higher level APIs such as Carbon, Cocoa, and QuickTime and includes pointers to more detailed documentation on these changes.
- "Cross-Architecture Plug-in Support" (page 45)—describes ways to support legacy plug-ins across architecture boundaries using helper hosts.
- "Performance Optimization" (page 57)—gives tips for spotting common performance regressions caused by transitioning your code to 64-bit.
- "Kernel Extensions and Drivers" (page 61)—tells how to transition your drivers and other kernel extensions to 64-bit executables.
- "Simulating a 64-Bit Address Space with mmap and munmap " (page 69)—describes a technique for emulating a 64-bit address space in a 32-bit application.

See Also

For additional information, see the following documents:

- *Getting Started with Tools & Languages* includes pointers to documentation that may help you solve 64-bit-related tools issues.
- *64-Bit Transition Guide for Cocoa* and *64-Bit Guide for Carbon Developers* provide information about Apple's 64-bit application APIs.
- *Universal Binary Programming Guidelines, Second Edition* provides information about the Intel transition. You should read this document and add native Intel support to your application first, since many of the Intel changes also apply to a 64-bit port.
- *Mac OS X ABI Mach-O File Format Reference* provides 64-bit ABI information that is useful if you are writing assembly language code.
- *Xcode Quick Start* provides information about using Xcode. You should be familiar with Xcode before you port your application or driver to 64-bit.

The `gcc`, `ld`, and `lipo` man pages may also be relevant to you.

Should You Recompile Your Software as a 64-Bit Executable?

Although 64-bit executables make it easier for you to manage large data sets (compared to memory mapping of large files in a 32-bit application), the use of 64-bit executables may raise other issues. Therefore you should transition your software to a 64-bit executable format only when the 64-bit environment offers a compelling advantage for your specific purposes.

This chapter explores some of the reasons you might or might not want to transition your software to a 64-bit executable format. Before you read this entire guide, read this chapter to decide whether your software will benefit from having a 64-bit executable format. When you have finished, if you are convinced that your software will benefit from a 64-bit executable format, you should read the remaining chapters in this document.

If some of the capabilities of a 64-bit environment would be helpful to you but you do not want to transition your software to a 64-bit executable, read the section "Alternatives to 64-Bit Computing" (page 15) to learn techniques that offer many of the same benefits but let you remain in a 32-bit environment.

Common Misconceptions

Before going further, it is important to dispel a few common misconceptions.

- Myth #1

  Myth: My software has to be 64-bit (or run on a 64-bit–capable computer) to use 64-bit data or do 64-bit math.

  Fact: In 32-bit software, you can already use signed and unsigned 64-bit data types such as `long long`. Internally, operations on these 64-bit values use a pair of 32-bit registers by default. If your code needs to run only on 64-bit Macintosh computers, you can get better performance by enabling true 64-bit math in leaf functions. See "Alternatives to 64-Bit Computing" (page 15) for more information.

- Myth #2

  Myth: The kernel needs to be 64-bit in order to be fully optimized for 64-bit processors.

  Fact: The kernel does not generally need to directly address more than 4 GB of RAM at once. The kernel is able to make larger amounts of memory available to 64-bit applications by using `long long` data types to keep track of mappings internally. See "Kernel Extensions and Drivers" (page 61) for more information about how 64-bit architectures affect device drivers and kernel extensions. There is a caveat, however, when dealing with configurations with large amounts of RAM. As the physical RAM grows, the size of data structures used to manage memory mappings also grows. Beyond a certain limit, it becomes impractical to keep them in a 32-bit address space. Thus, beginning in Snow Leopard, the kernel is moving to a 64-bit executable on hardware that supports such large memory configurations.

- Myth #3

  Myth: All of the system calls and corresponding C library functions have to change (or new ones have to be added) for 64-bit compatibility.

Fact: Most of the system call arguments changed to 64-bit many years ago. Some operating systems have separate 64-bit versions of these functions, such as `llseek64`. In Mac OS X, these variants are unnecessary because those functions are already 64-bit capable.
The notable exceptions are those functions related to memory management, such as `mmap`, `malloc`, and so on. Those functions have changed in terms of the size of data passed (because the size of `size_t` changed), but this change should be largely transparent to you as a programmer.

- Myth #4

  Myth: Every application needs the ability to work with more than 4 GB of RAM.

  Fact: Most applications have relatively modest memory requirements (a gigabyte or less). Other applications need more, but many of these larger applications can support larger data sets without moving to a 64-bit address space. Some common examples are applications that work with large media files in a sequential manner, such as music and video playback applications.

- Myth #5

  Myth: My application will run much faster if it is a “native” 64-bit application.

  Fact: Some 64-bit executables may run more slowly on 64-bit Intel and PowerPC architectures because of increased cache pressure.

  On Intel-based Macintosh computers, you may see some performance improvement. The number of registers and the width of registers increases in 64-bit mode. Because of the increased number of registers, function call parameters can be passed in registers instead of on the stack. The increased register width makes certain performance optimizations possible in 64-bit mode that are not possible in 32-bit mode. These improvements will often (but not always) offset the performance impact caused by increased cache pressure.

  The 32-bit PowerPC architecture is a 32-bit subset of a 64-bit architecture. The PowerPC architecture supports 64-bit arithmetic instructions in 32-bit mode (with some limitations). Since there are ample registers on 32-bit PowerPC, function call parameters on PowerPC have always been passed in registers. For these reasons, on PowerPC architectures, software does not generally become significantly faster (and may actually slow down) when compiled as a 64-bit executable.

Factors to Consider

A 64-bit executable can provide many benefits to users and to programmers, depending on the nature of your program. As a general rule, although a 32-bit application can provide the same functionality as a 64-bit application, a 64-bit application requires less effort to support large data sets.

Some applications can benefit significantly from 64-bit computing on both PowerPC and Intel. These include data mining, web caches and search engines, CAD/CAE/CAM software, large-scale 3D rendering (such as a movie studio might use, *not* a computer game), scientific computing, large database systems (for custom caching), and specialized image and data processing systems.

On Intel-based Macintosh computers, most applications will be somewhat faster when recompiled as a 64-bit executable. Whether this benefit justifies needed porting effort depends largely on how important performance is to your particular application and whether your application would benefit from a larger address space.

Note: With very few exceptions, you should *not* compile your applications as 64-bit-only executables. Instead, you should compile them as three-way or four-way universal binaries containing both 32-bit Intel and PowerPC executable code as well as executable code for one or both 64-bit architectures.

There are a number of factors to consider when deciding whether to make your application run in 64-bit mode. These considerations are described in the sections that follow:

- "Operating System Version" (page 13)
- "Kernel Extensions" (page 13)
- "Performance-Critical Applications" (page 13)
- ""Huge" Data Objects" (page 14)
- "64-Bit Math Performance" (page 14)
- "Plug-in Compatibility" (page 15)
- "Memory Requirements" (page 15)

Operating System Version

Prior to Mac OS X v10.6, all applications that shipped with the operating system were 32-bit applications. Beginning in v10.6, applications that ship with the operating system are generally 64-bit applications.

This means that in v10.5 and previous, the first third-party 64-bit application that a user runs causes the entire 64-bit framework stack to be brought into memory, resulting in a launch performance penalty and significant memory overhead.

Similarly, in v10.6 and later, the first non-64-bit-capable application pays a performance and memory footprint penalty because Mac OS X must bring in the entire 32-bit framework stack.

Kernel Extensions

If you are writing a kernel extension, you *must* make it 64-bit-capable. Beginning in Snow Leopard, some hardware configurations use a 64-bit kernel by default. The 64-bit kernel cannot load 32-bit kernel extensions.

Performance-Critical Applications

If your application is performance critical, you might want to recompile your application as a 64-bit executable, particularly on Intel-based Macintosh computers.

Here's why. The 64-bit Intel architecture contains additional CPU registers that are not available when compiling a 32-bit Intel executable. For example, the 64-bit architecture has 16 general-purpose integer registers instead of 8. Because of the extra register space, the first few arguments are passed in registers instead of on the stack. Thus, by compiling some applications as 64-bit, you may improve performance because the code generates fewer memory accesses on function calls. As a general rule, 64-bit Intel executables run somewhat more quickly unless the increased code and data size interact badly (performance-wise) with the CPU cache.

By contrast, executables compiled for the 64-bit PowerPC architecture can access the same number of registers (32) as 32-bit PowerPC executables. As a general rule, 64-bit PowerPC executables will execute slightly more slowly unless they make significant use of 64-bit math. Thus, if your application does not require a 64-bit address space, you may want to ship your application as a 32-bit executable on PowerPC by default.

As with any complicated software system, it is difficult to predict the relative performance of recompiling a piece of software as a 64-bit executable. The only way to know for certain (on either architecture) is to compile for 64-bit and benchmark both versions of the application.

Here are some of the potential performance pitfalls:

- Larger code and data size can result in increased cache and translation lookaside buffer (TLB) misses.
- Larger code and data (both pointers and `long` integers) can require more memory to avoid paging.
- The instruction sequence to get an address or constant into a register is longer for 64-bit code on PowerPC.
- Multiply and divide operations are slower when performed on 64-bit quantities than 32-bit quantities. Other operations take roughly the same amount of time as their 32-bit counterparts. Thus, if your code frequently multiplies values of type `long`, you will see a performance impact. (The reverse is true for type `long long` because 64-bit applications do not have to break 64-bit operations up into multiple 32-bit operations.)
- When you use a 32-bit signed integer as an array index, if that number is not stored in a register, the CPU will spend extra time on each access to sign-extend the value.

For the most part, these potential performance impacts should be small, but if your application is performance critical, you should be aware of them.

“Huge” Data Objects

If your application may need random access to exceptionally large (>2GB) data sets, it is easier to support these data sets in a 64-bit environment. You can support large data sets in a 32-bit application using memory mapping, but doing so requires additional code. Thus, for new applications, you should carefully evaluate whether supporting such large data sets is required in the 32-bit version of your application.

Note: It is not generally necessary to use 64-bit programming when working with files larger than 2 GB in a streaming fashion, such as when writing an audio or video application. These sorts of applications work with only a small section of a file at any given time and thus do not generally benefit significantly from the large address space of 64-bit computing. That said, these applications often do benefit from the additional registers afforded by 64-bit computing on the Intel architecture.

64-Bit Math Performance

Applications that use 64-bit integer math extensively may see performance gains on both PowerPC- and Intel-based Macintosh computers. In 32-bit applications, 64-bit integer math is performed by breaking the 64-bit integer into a pair of 32-bit quantities. It is possible to perform 64-bit computation in leaf functions in 32-bit applications, but this functionality generally offers only limited performance improvement.

Note: You do not need to transition your application to a 64-bit executable format merely because your application performs 64-bit math. You can perform 64-bit math transparently in a 32-bit application, albeit with slightly diminished performance.

Plug-in Compatibility

If you are writing an application, any plug-ins used by your application must be compiled for the same processor architecture and address width as the running application. For this reason, if your application depends heavily upon plug-ins (audio applications, for example), you may want to ship it as 32-bit for now.

Alternatively, you might add a user-selectable install option for the 64-bit version and then glue the two binaries together using the `lipo` command in a postinstall script. Doing so will encourage plug-in developers to update their code for 64-bit execution and at the same time will minimize user complaints.

If you are writing a plug-in, you should begin transitioning your plug-in to 64-bit so that when 64-bit versions of the supporting application become available, your plug-in will not get left behind.

Beginning in Snow Leopard, Apple-developed applications (including key components of the OS) are transitioning to 64-bit executables. This means that users with 64-bit-capable computers will be running the 64-bit slice of these key system components. Any plug-ins (screen savers, printer dialog extensions, and so on) that need to load in these applications must be recompiled as 64-bit plug-ins.

As a special exception, the System Preferences application provides a 32-bit fallback mode. If the user selects a system preferences pane without a 64-bit slice, it relaunches itself as a 32-bit executable (after displaying a dialog box). To maximize your users' experience, however, you should still transition these preference panes to 64-bit plug-ins at your earliest convenience.

Memory Requirements

The memory usage of a 64-bit application may be significantly larger than for a 32-bit version of the same application. The difference in usage varies from application to application depending on what percentage of data structures contain data members that are larger in a 64-bit process. For this reason, on a computer with a small amount of memory, you may not want to run the 64-bit version of your application even if the computer can support it.

This concern is described in more detail in "Performance Optimization" (page 57), along with some tips for improving your memory usage in a 64-bit environment.

Alternatives to 64-Bit Computing

If you need your application to do 64-bit integer math, you can do so already in Mac OS X by using `long long` data types.

On PowerPC, if you compile your application using the `-mcpu=G5` flag (to use G5-specific optimizations) and the `-mpowerpc64` flag (to allow 64-bit math instructions), your 32-bit application can achieve 64-bit math performance comparable to that of a 64-bit application. This technique has some performance disadvantages, however, because nonleaf functions still work with 64-bit integer values in a pair of 32-bit registers due to the design of the 32-bit function call ABI.

Applications compiled with the `-mcpu=G5` and `-mpowerpc64` flags will not execute on non-G5 hardware. If you need to support G3 or G4 hardware, you can still do 64-bit math without these options with only a small performance penalty.

If your application accesses large files in a streaming fashion, such as an audio or video application, you can use existing Mac OS X file interfaces. Nearly all the file interfaces in Mac OS X are capable of handling 64-bit offsets even in 32-bit applications. However, Mac OS APIs that existed prior to HFS+ (such as QuickTime) may require you to use different functions for large file access. See the latest documentation for the APIs you are using for more specific information.

If you have a performance-critical application that would benefit from more than 4 GB of memory, you should read the section "Using mmap to Simulate a Large Address Space" (page 16).

Using mmap to Simulate a Large Address Space

As an alternative to using a large address space, you can simulate one in your application by creating your own pseudo-virtual-memory engine using the `mmap` system call. Instead of referring to data using pointers, use a data structure that contains a reference to a file and an offset into that file.

At first glance, this technique may seem incredibly inefficient, because you would expect the operating system to constantly move data into and out of memory. In practice, however, the Mac OS X VM system caches open files heavily. Thus, even though your application has only 4 GB of address space for use at any given time, your application can actually use far more than 4 GB of physical memory concurrently in the form of disk caches.

For this reason, if you do not close the file descriptor after you call `mmap` on the file, and if your computer's RAM is large enough to hold your application's entire data set, most of the memory mapping and unmapping operations should require little or no I/O. If the physical RAM is not large enough, your data ends up being paged to disk anyway; thus your performance is only marginally affected. Upon closing the file descriptor, these pages are released (after flushing dirty pages to disk).

Note: For optimal performance, you should generally limit the amount of data that you map at any given time to some reasonable percentage of total physical memory. To obtain the size of physical memory, you can use the following code:

```
#include <inttypes.h>
#include <stdio.h>

main()
{
    uint64_t mem_size;
    size_t len = sizeof(mem_size);
    int fail;
    if (sysctlbyname("hw.memsize", &mem_size, &len, NULL, 0) != 0) {
        perror("sctest");
    } else {
        printf("RAM size in bytes is %" PRIu64 ".\n", mem_size);
    }
}
```

When you need to access a piece of data, your in-application virtual memory code checks to see whether that information has already been mapped into memory. If not, it should map the data using `mmap`. If the `mmap` operation fails, your application has probably run out of usable virtual address space and must therefore choose a "victim" memory region and unmap it.

For optimal performance, a user-space VM system must use proper mapping granularity for the data. If the data divides neatly into fixed-size objects, these provide good units for mapping. Because the length of the mapped region always rounds up to the nearest page size boundary, you will usually find that performance improves if you map in groups of objects.

Note: You can find out the page size of the computer hardware you are using with the following code:

```
#include <inttypes.h>
#include <stdio.h>

main()
{
    uint64_t page_size;
    size_t len = sizeof(page_size);
    int fail;
    if (sysctlbyname("hw.pagesize", &page_size, &len, NULL, 0) != 0) {
        perror("sctest");
    } else {
        printf("RAM size in bytes is %" PRIu64 ".\n", page_size);
    }
}
```

If your data doesn't have convenient fixed-size objects, you may choose an arbitrary page size (no less than the underlying physical page size) and divide the data into pages of that size. (A power-of-2 boundary is particularly convenient because you can then calculate the page number and the offset into the page by using bit masks and shift operations.)

No matter how you map the data, unless you do a lot of access pattern profiling, you may find it difficult to guess a good mapping granularity for most applications. For this reason, you should design your code with proper abstraction so that you can more easily adjust the mapping granularity in the future.

The code sample in "Simulating a 64-Bit Address Space with mmap and munmap " (page 69) demonstrates the use of `mmap` to map and unmap pieces of a large file.

Major 64-Bit Changes

There are many differences between 32-bit and 64-bit environments in Mac OS X, including tool usage changes, changes to the size and alignment of data types, alignment pragmas, and I/O Kit drivers. This chapter describes the main changes developers should be aware of when porting code to 64-bit. You should read this chapter if you've decided to port your code to 64-bit or if you are writing a new code from scratch.

Tools Changes

You'll find a number of issues when porting code to a 64-bit executable. You can address most of these issues with subtle tweaks to your code. However, before you touch the first line of code, there are a few broad issues you should be aware of:

- Compiling an application as a 64-bit executable requires you to use GCC 4.0 or later. See "Compiling 64-Bit Code Using GCC" (page 37) for information about related compiler flags.
- Xcode has additional options related to 64-bit compilation. For information about compiling 64-bit applications with Xcode, see "Compiling 64-Bit Code Using Xcode" (page 39).
- Any tools that understand the Mach-O ABI (stack frames, calling convention, and so on) must change. These changes affect mainly third-party compilers and linkers. For more information, see *Mac OS X ABI Mach-O File Format Reference*.

Data Type Changes

This section describes the changes to data type sizes and alignment in 64-bit executables, and explains how these changes will impact your code.

Data Type Size and Alignment

Mac OS X uses two data models: ILP32 (in which integers, `long` integers, and pointers are 32-bit quantities) and LP64 (in which integers are 32-bit quantities, and `long` integers and pointers are 64-bit quantities). Other types are equivalent to their 32-bit counterparts (except for `size_t` and a few others that are defined based on the size of `long` integers or pointers).

While almost all UNIX and Linux implementations use LP64, other operating systems use various data models. Windows, for example, uses LLP64, in which `long long` variables and pointers are 64-bit quantities, while `long` integers are 32-bit quantities. Cray, by contrast, uses ILP64, in which `int` variables are also 64-bit quantities.

In Mac OS X, the default alignment used for data structure layout is natural alignment (with a few exceptions noted below). Natural alignment means that data elements within a structure are aligned at intervals corresponding to the width of the underlying data type. For example, an `int` variable, which is 4 bytes wide, would be aligned on a 4-byte boundary.

Note: The data types `fpos_t`, `off_t`, and `long long` were historically exceptions to the natural alignment rules in the 32-bit PowerPC standard. Though they were all 8-byte (64-bit) types, they were aligned on 4-byte (32-bit) boundaries. In a 64-bit environment, these data types are naturally aligned on 8-byte (64-bit) boundaries.

Table 2-1 shows some common data types used in Mac OS X, along with their size and alignment. LP64 differences are highlighted in bold.

Table 2-1 Size and alignment of base data types in Mac OS X

Data type	ILP32 size	ILP32 alignment	LP64 size	LP64 alignment
char	1 byte	1 byte	1 byte	1 byte
short	2 bytes	2 bytes	2 bytes	2 bytes
int	4 bytes	4 bytes	4 bytes	4 bytes
long	4 bytes	4 bytes	**8 bytes**	**8 bytes**
pointer	4 bytes	4 bytes	**8 bytes**	**8 bytes**
size_t	4 bytes	4 bytes	**8 bytes**	**8 bytes**
long long	8 bytes	4 bytes	8 bytes	**8 bytes**
fpos_t	8 bytes	4 bytes	8 bytes	**8 bytes**
off_t	8 bytes	4 bytes	8 bytes	**8 bytes**

Note: Floating-point data type sizes are the same whether you are generating a 32-bit or 64-bit executable. However, the size of `long double` is 128 bits wide in GCC 4.0 and later (required for 64-bit compilation). Previous versions of the compiler (3.3 and earlier) used a 64-bit-wide `long double` type.

Because changes in size and alignment can significantly affect the data size produced by your code, you should generally pack structures so that the largest data types appear first, followed by progressively smaller data types. In this way, you maximize the use of space.

If, for compatibility, you need to support on-disk or network data structures containing 64-bit values aligned on 4-byte boundaries, you can override the default alignment using pragmas. See "Making Code 64-Bit Clean" (page 23) for more information.

In addition to these changes to the base data types, various layers of Mac OS X have other data types that change size or underlying type in a 64-bit environment. These changes are described in *64-Bit Guide for Carbon Developers*, *64-Bit Transition Guide for Cocoa*, and "Kernel Extensions and Drivers" (page 61).

Data Type Impact on Code

Data type and alignment changes impact developers in several broad areas.

- Interprocess communication, networking, shared memory, and user-kernel boundary crossings

  If you need your 64-bit software to communicate with 32-bit software (whether over a network, through local IPC mechanisms, through shared memory, or through crossing the user-kernel boundary in any way), choose data types carefully. A good practice is to always use explicitly sized data types, such as `uint32_t`, rather than generic data types (such as `long`).

  You may find it hard to use some mechanisms of interprocess communication, such as shared memory, when sharing data between 32-bit and 64-bit code. In particular, you should avoid passing pointers into shared memory regions and instead use offsets into the shared buffer.

- Files stored on disk

  If you need your application to write binary data in a file format that is shared between 64-bit and 32-bit versions, make sure that the size and alignment of data structures are the same in both versions. Specifically, these programs should avoid storing data of type `long` to disk.

  Alternatively, you can create a separate file format that is specific to the 64-bit version of your application. For some applications, creating a new format may be easier than maintaining a shared file format. This should be considered the exception rather than the rule, however.

  Finally, never underestimate the convenience of a generic exchange format such as XML.

- Libraries

  All libraries used by 64-bit applications or kernel extensions must be recompiled with a 64-bit compiler. If these libraries are also needed for 32-bit applications or kernel extensions, you must use a dual-architecture library (or have multiple copies of the library).

- Plug-ins

  Applications you compile as a 64-bit executable cannot load 32-bit plug-ins directly. Similarly, applications you compile as a 32-bit executable cannot load 64-bit plug-ins.

  If your application must support plug-ins compiled for multiple architectures, you should use a helper application and communicate with that helper using an interprocess communication mechanism. This is described further in "Cross-Architecture Plug-in Support" (page 45).

- Graphical user interfaces

  Higher-level frameworks used for graphical user interfaces are available as 64-bit frameworks *only* in Mac OS X v10.5 and later. Previous versions of Mac OS X will automatically use the 32-bit version of your executable.

- Data alignment differences

  When you are compiling code with a 64-bit target, keep in mind that the default alignment for 64-bit data types is 64-bit rather than the 32-bit alignment you may be used to. If you require interoperability with 32-bit software (file formats, network protocols, user-kernel boundary crossings, and so on), you must change the code.

  If you do not have to maintain format compatibility with existing data, to avoid wasting memory and storage, you should reorder the members of the structure so that the 64-bit data types fall on a 64-bit offset from the start of the structure. If such a change is not possible for compatibility reasons, you can override the alignment rules using a pragma. See "Making Code 64-Bit Clean" (page 23) for details.

Note: The `mac68k` structure packing pragma is not available in 64-bit applications. See "Alignment Pragmas" (page 30) for more information.

Security Changes

In 64-bit executables, executing code in data segments is not allowed. To achieve this, the NX (no execute) bit is set on the page table entries for the stack, heap, and initialized and uninitialized data segments.

In general, this should have no impact on developers unless you are doing something nonstandard such as writing self-modifying code. If you are writing code that requires execution on the stack or in the heap, you must explicitly mark the pages executable using `mprotect` or other similar calls.

Making Code 64-Bit Clean

Before you begin to update your code, you should familiarize yourself with the document *Mac OS X Technology Overview*. After reading that document, the first thing you should do is compile your code with the `-Wall` compiler flag and fix any warnings that occur. In particular, make sure that all function prototypes are in scope, because out-of-scope prototypes can hide many subtle portability problems.

At a high level, to make your code 64-bit clean, you must do the following:

- Avoid assigning 64-bit `long` integers to 32-bit integers.
- Avoid assigning 64-bit pointers to 32-bit integers.
- Fix alignment issues caused by changes in data type sizes.
- Avoid pointer and `long` integer truncation during arithmetic operations.

General Programming Tips

This section contains some general tips for making your code 64-bit clean.

Update architecture-specific code. If your software contains any architecture-specific code, you must either add extra code for each additional architecture or modify your preprocessor directives so that the same code is included for multiple supported architectures.

Unless you are including inline assembly language code, you should generally test for the presence of architecture-neutral macros such as `__LITTLE_ENDIAN__` or `__LP64__` rather than testing for a specific processor architecture.

The macro `__LP64__` can be used to test for LP64 compilation in an architecture-independent way. For example:

```
#ifdef __LP64__
// 64-bit code
#else
// 32-bit code
#endif
```

For code that is truly architecture-specific (such as assembly language code), you should continue to use architecture-specific tests. Be aware, however, that when compiling for a 64-bit architecture, code wrapped in a test for a 32-bit architecture is *not* compiled into the executable.

For example, code wrapped with the `#ifdef __ppc__` directive will not be included when compiling for the `ppc64` architecture. The following listing (Listing 3-1) gives examples of how to write tests for various architectures.

Listing 3-1 Architecture definition changes

```
#ifdef __ppc__
// 32-bit PowerPC code
#else
#ifdef __ppc64__
// 64-bit PowerPC code
#else
#if defined(__i386__) || defined(__x86_64__)
// 32-bit or 64-bit Intel code
#else
#error UNKNOWN ARCHITECTURE
#endif
#endif
#endif
```

Code that looks for only the __ppc__ or __i386__ definition will break if you compile for the related 64-bit architecture.

For code specific to Mac OS X (non-cross-platform), the TargetConditionals.h header also provides macro support for architecture-specific code. For example:

```
#include <TargetConditionals.h>

#if TARGET_RT_LITTLE_ENDIAN
    ...
#elif TARGET_RT_BIG_ENDIAN
    ...
#else
    #error Something is very wrong here.
#endif
```

Avoid casting pointers to nonpointers. You should generally avoid casting a pointer to a nonpointer type for any reason (particularly when performing address arithmetic). Alternatives are described in "Avoiding Pointer-to-Integer Conversion" (page 28).

Update assembly code. Any PowerPC assembly code needs to be checked because some instructions behave differently in 64-bit mode. Any Intel assembly code needs to be rewritten because 64-bit Intel assembly language is significantly different from its 32-bit counterpart. For more information, see "Porting Assembly Language Code" (page 32).

Any assembly code that directly deals with the structure of the stack (as opposed to simply using pointers to variables on the stack) must be modified to work in a 64-bit environment. For more information, see *Mac OS X ABI Mach-O File Format Reference*.

Fix format strings. Print functions such as printf can be tricky when writing code to support 32-bit and 64-bit platforms because of the change in the sizes of pointers. To solve this problem for pointer-sized integers (uintptr_t) and other standard types, various macros exist in the inttypes.h header file.

The format strings for various data types are described in Table 3-1. These additional types, listed in the inttypes.h header file, are described in Table 3-2.

Table 3-1 Standard format strings

Type	Format string
int	%d
long	%ld
long long	%lld
size_t	%zu
ptrdiff_t	%td
any pointer	%p

Table 3-2 Additional inttypes.h format strings (where *N* is some number)

Type	Format string
int*N*_t (such as int32_t)	PRId*N*
uint*N*_t	PRIu*N*
int_least*N*_t	PRIdLEAST*N*
uint_least*N*_t	PRIuLEAST*N*
int_fast*N*_t	PRIdFAST*N*
uint_fast*N*_t	PRIuFAST*N*
intptr_t	PRIdPTR
uintptr_t	PRIuPTR
intmax_t	PRIdMAX
uintmax_t	PRIuMAX

For example, to print an intptr_t variable (a pointer-sized integer) and a pointer, you write code similar to that in Listing 3-2.

Listing 3-2 Architecture-independent printing

```
#include <inttypes.h>
void *foo;
intptr_t k = (intptr_t) foo;
void *ptr = &k;

printf("The value of k is %" PRIdPTR "\n", k);
printf("The value of ptr is %p\n", ptr);
```

Data Type and Alignment Tips

Here are a few tips to help you avoid problems stemming from changes to data type size and alignment.

Be careful when mixing integers and long integers. The size and alignment of `long` integers and pointers have changed from 32-bit to 64-bit.

For the most part, if you always use the `sizeof` function when allocating data structures and avoid assigning pointers to nonpointer types, the size and alignment of pointers should not affect your code, because structures containing pointer members are generally not written to disk or sent across networks between 32-bit and 64-bit applications. This is something to consider when writing kernel extensions that read structures passed in from applications, however.

If you frequently move data between variables of type `int` and `long`, the change in the size of `long` can cause problems. You will see various related problems throughout this section.

Be sure to control alignment of shared data. The alignment of `long long` (64-bit) integers has changed from 32-bit to 64-bit. This alignment change can pose a problem when you are exchanging data between 32-bit and 64-bit code.

In Listing 3-3, the alignment changes even though the data types are the same size.

Listing 3-3 Alignment of `long long` integers in structures

```
struct bar {
    int foo0;
    int foo1;
    int foo2;
    long long bar;
};
```

When this code is compiled with a 32-bit compiler, the variable `bar` begins 12 bytes from the start of the structure. When the same code is compiled with a 64-bit compiler, the variable `bar` begins 16 bytes from the start of the structure, and a 4-byte pad is added after `foo2`.

If you must maintain data structure compatibility, to allow a single data structure to be shared, you can use a pragma to force power alignment mode (PowerPC only) or packed alignment mode for each structure, as needed. Then add appropriate pad bytes (if necessary) to obtain the desired alignment. An example is shown in Listing 3-4.

If backwards compatibility with existing structures is not important, you should reorder the data structure so that the largest fields are at the beginning of the structure. That way, the 8-byte fields begin at offset `0` and thus are aligned on 8-byte boundaries without the need to add an alignment pragma.

Listing 3-4 Using pragmas to control alignment

```
#pragma pack(4)
struct bar {
    int foo0;
    int foo1;
    int foo2;
    long long bar;
};
#pragma options align=reset
```

You should use this option only when absolutely necessary, because there is a performance penalty for misaligned accesses.

Note: If your code already explicitly uses `#pragma options align=power`, you should generally use `#pragma pack(4)` instead.

Use sizeof with malloc. Since pointers and long integers are no longer 4 bytes long, never call `malloc` with an explicit size (for example, `malloc(4)`) to allocate space for them. Always use `sizeof` to obtain the correct size.

Never assume you know the size of any structure (containing a pointer or otherwise); always use `sizeof` to find out for sure. To avoid future portability problems, search your code for any instance of `malloc` that isn't followed by `sizeof`. The `grep` command and regular expressions are your friend, though using Find in the Xcode Edit menu can do the job.

64-bit sizeof returns size_t. Note that `sizeof` returns an integer of type `size_t`. Because the size of `size_t` has changed to 64 bits, do not pass the value to a function in a parameter of size `int` (unless you are certain that the size cannot be that large). If you do, truncation will occur.

Use explicit (fixed-width) C99 types. You should use explicit types where possible. For example, types with names like `int32_t` and `uint32_t` will always be a 32-bit quantity, regardless of future architectural changes.

32-bit type	Suggested C99 type
`char` or `unsigned char` (only when used as a one-byte integer)	`int8_t` or `uint8_t`
`short` or `unsigned short`	`int16_t` or `uint16_t`
`int` or `unsigned int`	`int32_t` or `uint132_t`
`long` or `unsigned long`	`int32_t` or `uint32_t`
`long long` or `unsigned long long`	`int64_t` or `uint64_t`

Watch for conversion errors. Conversion of shorter types to 64-bit longs may yield unexpected results in certain cases. Be sure to read "Sign Extension Rules for C and C-derived Languages" (page 30) if you are seeing unexpected values from math that mixes `int` and `long` variables.

Use 64-bit types for pointer arithmetic results. Because the size of pointers is a 64-bit value, the result of pointer arithmetic is also a 64-bit value. You should always store these values in a variable of type `ptrdiff_t` to ensure that the variable is sized appropriately.

Avoid truncating file positions and offsets. Although file operations have always used 64-bit positions and offsets, you should still check for errors in their use. Errors will become more and more important as common file sizes grow. Use `fpos_t` for file position and `off_t` for file offset.

Be careful with variable argument lists. Variable argument lists (`varargs`) do not provide type information for the arguments, and the arguments are not promoted to larger types automatically. If you need to distinguish between different incoming data types, you are expected to use a format string or other similar mechanism to provide that information to the `varargs` function. If the calling function does not correctly provide that information (or if the `varargs` function does not interpret it correctly), you will get incorrect results.

In particular, if your varargs function expects a long type and you pass in a 32-bit value, the varargs function will contain 32 bits of data and 32 bits of garbage from the next argument (which you will lose as a result). Likewise, if your varargs function is expecting an int type and you pass in a long, you will get only half of the data, and the rest will incorrectly appear in the argument that follows.

For example, if you use incorrect printf format strings, you will get incorrect behavior. Some examples of these format string mistakes are shown in "General Programming Tips" (page 23).

Avoiding Pointer-to-Integer Conversion

You should generally avoid casting a pointer to a nonpointer type for any reason. If possible, rewrite any code that uses these casts, either by changing the data types or by replacing address arithmetic with pointer arithmetic. For example, the following code:

```
int *c = something passed in as an argument....
int *d = (int *)((int)c + 4); // This code is WRONG!
```

results in pointer truncation. Because the resulting value would be correct for sufficiently small pointers, these bugs can be difficult to find. Instead, this code can be replaced with:

```
int *c = something passed in as an argument....
int *d = c + 1;
```

(Of course, this example is somewhat contrived, and such use of pointers is relatively uncommon.)

A more common problem is storing a pointer temporarily in a variable of type int. In most cases, the compiler will warn you that a pointer is being assigned to an integer of a different size. However, in a few cases, code containing such an assignment will compile without warning. For example, if the code stores the values in a variable of type long and then later copies it to an integer, the pointer itself is not *directly* truncated, so the compiler may not generate a warning. These problems are particularly hard to spot.

Finally, a common problem is the need to offset a pointer by a specific number of bytes. Instead of casting to an integer and using integer math, you should cast the pointer to a byte-width pointer type such as char * or uint8_t *. After you do this, the pointer will behave like an integer for arithmetic purposes. For example:

```
int *myptr = getPointerFromSomewhere();
int *shiftbytwobytes = (int *)(((int)myptr) + 2);
```

can be rewritten as:

```
int *myptr = getPointerFromSomewhere();
int *shiftbytwobytes = (int *)(((char *)myptr) + 2);
```

By avoiding assignment of pointers to any nonpointer type, you avoid almost all pointer-related problems, because pointers are rarely stored or exchanged between 32-bit and 64-bit processes. In a few situations, however, there may be no easy way to avoid address-to-integer conversions. The uintptr_t type exists for these edge cases.

Note: Casting 64-bit pointers to integer types is also required when passing a pointer via a Mach port. In these cases, use the `caddr_t` type.

Working with Bits and Bitmasks

When working with bits and masks with 64-bit values, you must be careful to avoid getting 32-bit values inadvertently. Here are some tips to help you:

Shift carefully. If you are shifting through the bits stored in a variable of type `long`, don't assume that the variable is of a particular length. Instead, use the value `LONG_BIT` to determine the number of bits in a `long`. The result of a shift that exceeds the length of a variable is architecture-dependent.

Use inverted masks if needed. Be careful when using bit masks with variables of type `long`, because the width differs between 32-bit and 64-bit architectures. There are two ways to create a mask, depending on whether you want the mask to be zero-extended or one-extended:

- If you want the mask value to contain zeros in the upper 32 bits on a 64-bit architecture, the usual fixed-width mask will work as expected, because it will be extended in an unsigned fashion to a 64-bit quantity.
- If you want the mask value to contain ones in the upper bits, however, you should write the mask as the bitwise inverse of its inverse, as shown in Listing 3-5.

Listing 3-5 Using an inverted mask for sign extension

```
function_name(long value)
{
    long mask = ~0x3; // 0xfffffffc or 0xfffffffffffffffc
    return (value & mask);
}
```

In the code, note that the upper bits in the mask are filled with ones in the 64-bit case.

Tools Tips

Here are some tips to help you use the compiler more effectively in transitioning your code to 64-bit:

- If data is being inadvertently truncated, to help you find the source, try turning on additional compiler warnings.
- In 64-bit-capable versions of GCC (4.0 and later), the size of a `long double` will be 128 bits instead of 64 bits. This change is not limited to code compiled as a 64-bit executable, but it is a toolchain change you should be aware of.

You can find detailed tips and information about 64-bit tools changes in "Compiling 64-Bit Code" (page 37).

Alignment Pragmas

Occasionally, developers use alignment pragmas to change the way that data structures are laid out in memory. They usually do this for backward compatibility. In many cases, Apple added pragmas to maintain data structure compatibility between 68K-based and PowerPC-based code running on the same machine under Mac OS 9 and earlier. Mac OS X retained these alignment overrides to maintain binary compatibility with existing Carbon data structures between Mac OS 9 and Mac OS X.

There is a performance cost associated with pragmas, however; memory accesses to unaligned data fields result in a performance penalty. Because there are no existing 64-bit Mac OS X GUI applications with which to be compatible, it is not necessary to preserve binary compatibility for these data structures in 64-bit applications. Thus, to improve overall performance, when compiling 64-bit executables, the Mac OS X version of GCC ignores requests for `mac68k` alignment.

If you are using this pragma only to access Apple data structures, you should not need to make any code changes to your code. When compiling 64-bit code, the compiler ignores the pragmas and your code works correctly. If, however, you currently use the `mac68k` alignment pragma in your own data structures that will be shared between 32-bit and 64-bit versions of your application (or if you use the `mac68k` pragma for a data structure that corresponds with the register layout of a physical device), you must rewrite the data structure to use a packed alignment and pad the structure appropriately.

With the exception of Altivec data types, the following code is equivalent to `mac68k` alignment:

```
#pragma pack(2)
...structure declaration goes here...
#pragma options align=reset
```

Similarly, with the exception of Altivec data types, the following code is equivalent to the standard 32-bit PowerPC and Intel alignment (known on PowerPC as embedded power alignment):

```
#pragma pack(4)
...structure declaration goes here...
#pragma options align=reset
```

Sign Extension Rules for C and C-derived Languages

C and similar languages use a set of sign extension rules to determine whether to treat the top bit in an integer as a sign bit when the value is assigned to a variable of larger width. The sign extension rules are as follows:

1. The sum of a signed value and an unsigned value of the same size is an unsigned value.
2. Any promotion always results in a signed type unless a signed type cannot hold all values of the original type (that is, unless the resulting type is the same size as the original type).
3. Unsigned values are zero extended (not sign extended) when promoted to a larger type.
4. Signed values are always sign extended when promoted to a larger type, even if the resulting type is unsigned.

5. Constants (unless modified by a suffix, such as `0x8L`) are treated as the smallest size that will hold the value. Numbers written in hexadecimal may be treated by the compiler as `signed` and `unsigned int`, `long`, and `long long` types. Decimal numbers will always be treated as `signed` types.

Listing 3-6 shows an example of unexpected behavior resulting from these rules along with an accompanying explanation.

Listing 3-6 Sign extension example 1

```
int a=-2;
unsigned int b=1;
long c = a + b;
long long d=c; // to get a consistent size for printing.

printf("%lld\n", d);
```

Problem: When this code is executed on a 32-bit architecture, the result is `-1` (`0xffffffff`). When the code is run on a 64-bit architecture, the result is `4294967295` (`0x00000000ffffffff`), which is probably not what you were expecting.

Cause: Why does this happen? First, the two numbers are added. A signed value plus an unsigned value results in an unsigned value **(rule 1)**. Next, that value is promoted to a larger type. This promotion does not cause sign extension **(rule 2)**.

Solution: To fix this problem in a 32-bit-compatible way, cast `b` to `long`. This cast forces the non-sign-extended promotion of `b` to a 64-bit type prior to the addition, thus forcing the signed integer to be promoted (in a signed fashion) to match. With that change, the result is the expected `-1`.

Listing 3-7 shows a related example with an accompanying explanation.

Listing 3-7 Sign extension example 2

```
unsigned short a=1;
unsigned long b = (a << 31);
unsigned long long c=b;

printf("%llx\n", c);
```

Problem: The expected result (and the result from a 32-bit executable) is `0x80000000`. The result generated by a 64-bit executable, however, is `0xffffffff80000000`.

Cause: Why is this sign extended? First, when the shift operator is invoked, the variable `a` is promoted to a variable of type `int`. Because all values of a `short` can fit into a signed `int`, the result of this promotion is signed **(rule 3)**.

Second when the shift completed, the result was stored into a `long`. Thus, the 32-bit signed value represented by `(a << 31)` was sign extended **(rule 4)** when it was promoted to a 64-bit value (even though the resulting type is unsigned).

Solution: To fix this problem, you should cast the initial value to a `long` prior to the shift. Thus, the short will be promoted only once—this time, to a 64-bit type (at least when compiled as a 64-bit executable).

Velocity Engine and SSE Alignment Tips

Although the SSE and Velocity Engine C and assembly language interfaces have not changed for 64-bit, if you are using these technologies, you should review any code that attempts to align pointers to 16-byte addresses for processing.

For example, the following code contains two errors:

```
TYPE *aligned = (TYPE *) ((int) misalignedPtr & 0xFFFFFFF0); // BAD!
```

First, the pointer is cast to an `int` value, which results in truncation. Even after this problem is fixed, however, the pointer will still be truncated because the constant value `0xFFFFFFF0` is not a 64-bit value.

Instead, this code should be written as:

```
#include <stdint.h>
TYPE *aligned = (TYPE *) ((intptr_t) misalignedPtr & ~(intptr_t)0xF);
```

Porting Assembly Language Code

This section describes some of the issues involved in porting assembly language code to a 64-bit application. On the Intel architecture, in addition to the issues described in this section, you must considerably modify any assembly language code that deals with the stack directly, because the 64-bit ABI differs significantly from the 32-bit ABI. The subject of stack frames is beyond the scope of this section. For more information, see *Mac OS X ABI Mach-O File Format Reference*.

Porting PowerPC Assembly Language Code to 64-Bit

The include file `<architecture/ppc/mode_independent_asm.h>` contains a number of useful pseudo-mnemonics and macros to make it easier for you to write assembly language code that can be built in 32-bit or 64-bit mode.

Record form ("."), overflow form ("o"), and extended form ("e") instructions all behave differently in 64-bit mode. Specifically, they test and set condition bits on the 64-bit result, not the 32-bit result. For example:

```
lis     r5,0x4000
add.    r5,r5,r5
blt     xxx
```

In 32-bit mode, the add instruction sets `cr0_lt`, and so the branch is taken. In 64-bit mode, the branch is not taken.

All word-compare, shift, load, and store instructions must be carefully considered. Many need to be changed to use the corresponding 64-bit opcode. For example, this 32-bit code:

```
lwz     r5,0(r3)    // load an address
rlwinm  r5,r5,0,0,29// word align
cmplwi  r5,0        // null?
```

should be rewritten as:

```
ld      r5,0(r3)    // load an address
rldicr  r5,r5,0,0,60// doubleword align
cmpldi  r5,0        // null?
```

Porting Intel Assembly Language Code to 64-Bit

On Intel-based Macintosh computers, 64-bit code uses the Intel 64 (formerly EM64T) extensions to the Intel assembly language ISA. This section summarizes the differences between Intel 64 code and IA32 code in terms of their impact on registers and instruction sets.

Note: The Intel 64 architecture, otherwise known as x86-64, should not be confused with the IA64 architecture. The IA64 architecture, short for Intel Architecture 64, is the architecture used by Itanium processors. Despite their similar names, the Intel 64 Architecture is not related to the Itanium (IA64) architecture.

Register Changes

The 64-bit registers on Intel have different names than their 32-bit counterparts do. In addition, there are more of them. These register names are listed in Table 3-3.

Table 3-3 Register naming on 32-bit and 64-bit Intel architectures

IA32 32-bit register	Intel 64 Architecture 64-bit variant	Description
EIP	RIP	Instruction Pointer
EAX	RAX	General Purpose Register A
EBX	RBX	General Purpose Register B
ECX	RCX	General Purpose Register C
EDX	RDX	General Purpose Register D
ESP	RSP	Stack Pointer
EBP	RBP	Frame Pointer
ESI	RSI	Source Index Register
EDI	RDI	Destination Index Register
----	R8 *	Register 8 (new)
----	R9 *	Register 9 (new)
----	R10 *	Register 10 (new)
----	R11 *	Register 11 (new)
----	R12 *	Register 12 (new)
----	R13 *	Register 13 (new)

IA32 32-bit register	Intel 64 Architecture 64-bit variant	Description
----	R14 *	Register 14 (new)
----	R15 *	Register 15 (new)

All of the new registers (R8 through R15) added in the Intel 64 architecture instruction set can also be accessed as 32-bit, 16-bit, and 8-bit registers. For example, register R8 can be addressed in the following ways:

Register name	Description
R8	A 64-bit register.
R8d	A 32-bit register containing the bottom half of R8.
R8w	A 16-bit register containing the bottom half of R8d.
R8l (Lowercase "l")	An 8-bit register containing the bottom half of R8w.

Note: When you assign a value to the 32-bit version of these registers, the value is automatically zero extended to fill the corresponding 64-bit register. To sign extend during assignment, use the movsx instruction instead.

In addition to adding general-purpose registers, the Intel 64 Architecture instruction set has eight additional vector registers. In the IA32 instruction set, the vector registers are numbered XMM0 through XMM7. The Intel 64 Architecture instruction set extends this by adding XMM8 through XMM15.

Instruction Changes

Most IA32 instructions can take 64-bit arguments. All IA32 instruction set extensions up through SSE3 are included as part of the Intel 64 Architecture. In addition, a number of new instructions have been added.

A complete list of these changes is beyond the scope of this document. For information on these changes, see the links in "For More Information" (page 34).

For More Information

For more information on porting and optimizing Intel assembly language code for 64-bit, you should also read:

- *Mac OS X ABI Mach-O File Format Reference*—ABI documentation for Mac OS X.
- http://developer.intel.com/technology/intel64/index.htm—Intel 64 Architecture technology page (Intel).
- http://software.intel.com/en-us/parallel/—Intel multicore programming documentation site (Intel).
- http://software.intel.com/en-us/articles/porting-to-64-bit-intel-architecture/—Porting to 64-bit Intel architecture (Intel).
- http://software.intel.com/en-us/articles/porting-code-to-intel-em64t-based-platforms/—Information about 64-bit optimization. Note that the ABI information at this location is Windows oriented, so those portions do not apply.

- http://www.x86-64.org/documentation/assembly.html—General information on 64-bit Intel assembly (x86-64.org).

Compiling 64-Bit Code

The first part of this document describes issues you should consider when bringing code to a 64-bit architecture. You should read through those chapters before you compile your code for the first time, to help you determine whether the compiler warnings are useful or relevant (and possibly do an initial code scrub to look for common errors).

After you have read those chapters, it's time to compile your code with a 64-bit target architecture. You can either compile your code directly (using GCC) or through Xcode. This chapter takes you through the process of setting up your build environment for 64-bit compilation.

Compiling 64-Bit Code Using GCC

For the most part, compiling 64-bit code using GCC works the same way as compiling 32-bit code; there are a few exceptions, however:

- You *must* use GCC 4.0 or later. To choose a GCC version to be used when typing `gcc` on the command line, type `gcc_select 4.0`. To change the GCC version to be used in Xcode, see "Compiling 64-Bit Code Using Xcode" (page 39).
- You should turn on the `-Wall` flag (and possibly the `-Wconversion` flag if you are debugging conversion problems) in order to get additional warnings about potential pointer truncation and other issues.
- You must specify a 64-bit architecture (PPC64 or x86-64) with `-arch ppc64` or `-arch x86_64`. You can also compile binaries with multiple architectures, such as `-arch ppc -arch ppc64 -arch x86_64`.

In addition to these exceptions, there are a few new flags and features added for 64-bit architectures. Also, a few flags are not available for 64-bit architectures. The key differences are described in the next section.

New Flags and Features for 64-Bit Architectures

Several flags related to 64-bit architectures have been added or modified in GCC:

`-arch ppc64`
: The 64-bit PowerPC architecture option.

`-arch x86_64`
: The 64-bit x86 architecture option.

`-mmacosx-version-min=10.5`
: In addition to its usual function, this flag edits PowerPC linker behavior to enforce a 4 GB "page zero" size. (By default, all 64-bit Intel executables have a 4 GB page zero.) Thus, if you compile your application

with this flag, the lower 4 GB of the address space of your process will be empty and will be protected against reading or writing.

As a special exception, any code generated by compiling with `-mdynamic-no-pic` on PowerPC is located within the lower 2 GB. This exception applies only to the code itself, not data used by the code.

By using this flag, any code that is not 64-bit clean becomes immediately obvious. As soon as the upper portion of a pointer is zeroed, any attempt to dereference that pointer fails because the access will fall within the now-protected page zero.

In addition, this flag changes the CPU subtype of the resulting executable to prevent execution of the 64-bit portion on a Mac OS X v10.4 system.

⚠ **Warning:** In some versions of the developer tools, the `-mmacosx-version-min` flag is set to version 10.4 by default. You must explicitly set a 10.5 minimum deployment target in Xcode when compiling a 64-bit application. If you do not, the application will crash when run on a 64-bit-capable Mac OS X v10.4 system.

To use a different value for the 64-bit slice of your application, read "Using Architecture-Specific Flags" (page 39).

`-Wconversion`

Although not technically new for 64-bit architectures, this option is mostly useful when transitioning 32-bit code to 64-bit. This flag causes additional warnings to be printed when certain conversions occur between data types of different sizes. Most of these warnings will not be useful, so you should not necessarily fix everything that generates a warning with this flag. However, you may sometimes find this flag useful for tracking down hard-to-find edge cases.

In particular, this flag can also help track down edge cases in which a series of legal conversions result in an illegal conversion. For example, with this flag, the compiler will issue a warning if you assign a pointer to a 64-bit integer, pass that pointer into a 32-bit function argument, and subsequently convert the 64-bit function result back into a pointer.

Of course, this flag also produces numerous unexpected warnings that are harmless. For example, in ANSI C, a character literal (for example, `'c'`) is of type `int`, so passing it to a function that takes type `char` gives a truncation warning. Although the warning is pedantically correct, it is largely irrelevant. You'll have to sort through all these warnings by hand and determine which ones are legitimate and which ones are fiction.

`-Wformat`

While not a 64-bit-specific flag, this flag will help you catch mistakes in format arguments to `printf`, `sprintf`, and similar functions. This flag is turned on by default if you use the `-Wall` flag.

`-Wshorten-64-to-32`

This flag is like `-Wconversion`, but is specific to 64-bit data types. This flag causes GCC to issue a warning whenever a value is implicitly converted (truncated) from a 64-bit type to a 32-bit type. You should fix any warnings generated by this flag, as they are likely to be bugs.

In addition, the following flags are highly recommended:

`-Wall`

Turns on a lot of generally useful warnings.

`-Wimplicit-function-declaration`

Warns if a function is used prior to its declaration. This can help catch many mistakes caused by differing sizes of function arguments and return types.

Compiling 64-Bit Code Using Xcode

This section explains how to get started compiling 64-bit code using Xcode. These instructions assume that you have already installed the necessary command-line components—that is, a 64-bit–aware version of the compiler, linker, assembler, and other low-level tools.

With Xcode 1.0 and later, you can build multiarchitecture binaries (MABs). Because each target can define the set of architectures for the target being built, you can disallow architectures on a per-target basis. You might, for example, choose to build a target with a reduced list of architectures if the target contains assembler code for a particular processor or is not 64-bit-clean.

Each time you run the command-line tool `xcodebuild`, you can specify which target architectures to build. You can also configure a "build style" to build a particular set of architectures from within Xcode.

Xcode then builds the target for each of the architectures specified, skipping any architectures that the target does not support. If the target doesn't support any of the specified architectures, that target is skipped entirely.

The build setting `VALID_ARCHS` defines the architectures for which a given target can be built. This setting should contain a list of architectures separated by spaces. For example, to specify that your target can be built for `ppc`, `ppc64`, `i386`, and `x86_64`, set `VALID_ARCHS` to `"ppc ppc64 i386 x86_64"` (without the quotes) in the Xcode inspector for your target.

The build setting `ARCHS` defines the architectures for which the entire project should be built. This setting should also contain a space-delimited list of architectures. This build setting can be defined either on the command-line to `xcodebuild`, or in a build style in Xcode.

For example, to build your project for both 32-bit and 64-bit architectures, type:

```
xcodebuild ARCHS="ppc ppc64 i386 x86_64"
```

You can also set `ARCHS="ppc ppc64 i386 x86_64"` in a build style in your project. Similarly, if you want to build only a 64-bit version, specify `ARCHS="ppc64 x86_64"`.

If your source code includes special 64-bit versions of framework headers or library headers, you may need to add search paths to the Header Search Paths and Framework Search Paths build settings in the target inspector.

If you are building a target for more than one architecture at the same time, you will see each source file being compiled more than once. This is normal behavior. Xcode compiles each source file once for each architecture so that you can pass different compiler flags for each architecture. The files are glued together at the end of compilation using `lipo`. For more information, see `lipo`.

Using Architecture-Specific Flags

Normally, any build settings you specify in the target inspector are used for all architectures for which the target is built. In many cases, however, you need to specify different flags for 64-bit architectures.

If you want to specify additional per-architecture compiler flags, you can use the `PER_ARCH_CFLAGS_`*<arch>* family of build settings, where *<arch>* is the name of the architecture. For example, to specify compiler flags that apply only to 64-bit PowerPC compilation, add them to the `PER_ARCH_CFLAGS_ppc64` build setting.

For example, if you want to make your 64-bit slice run only on Mac OS X v10.6 instead of v10.5, you could set a per-architecture value for “Mac OS X Deployment Target”:

- Click the target and choose “Get Info” from the File menu.
- Click “Mac OS X Deployment Target”.
- Click the gear at the lower left corner of the info window and choose “Add Build Condition Setting” from the resulting pop-up menu.
- Change the architecture to “Intel 64-bit” and specify the x86-64 setting for this option.
- Add additional conditions as needed for additional architectures.
- Change the main setting (above the constrained settings) to whatever you want the default value to be.

Historical Footnote: Compiling 64-Bit Command-Line Code for Mac OS X v10.4

Mac OS X v10.4 supports some 64-bit executables. However, Mac OS X v10.4 does not include a full 64-bit stack; Mac OS X v10.4 contains only `libSystem` and the Accelerate framework in 64-bit versions. In addition, Mac OS X 10.4 includes neither the Objective-C runtime nor a 64-bit Objective-C-capable version of `dyld`. Because of these differences, if you try to execute a 64-bit executable in 10.4 that depends on these 10.5-specific features, it would crash.

To prevent new 64-bit executables from running as 64-bit on version 10.4, Apple changed the CPU subtype for 64-bit executables that depend on high-level frameworks.

If you need to compile an executable as 64-bit for Mac OS X v10.4, you must select the 10.4 deployment target when building 64-bit executables and separate your code into a 32-bit front-end (GUI) portion and a 64-bit back-end (processing) portion.

High-Level 64-Bit API Support

Beginning in Mac OS X v10.5, most Mac OS X APIs are available to 64-bit applications. Going forward, Apple plans to make new APIs 64-bit-compatible unless otherwise noted. However, not all existing 32-bit APIs are supported in 64-bit applications.

This chapter includes a brief summary of these API changes and limitations and includes pointers to other documentation that provides more detailed coverage for specific technology areas.

High-Level API Changes at a Glance

The high-level API changes related to 64-bit support generally fall into one of the following categories:

- "Changes to Data Types" (page 41)
- "New/Replaced/Deprecated APIs" (page 42)

These are described in detail in the sections that follow.

Changes to Data Types

The most significant change is to data types and type usage. A number of Apple-specific data types are defined differently in the 64-bit world. In some cases, 32-bit values have been replaced with 64-bit values for future expansion. In other cases, data types that become 64-bit in a 64-bit environment have been replaced with data types that remain 32-bit—to preserve file format compatibility, for example.

The result of these changes is that a number of derived data types have different sizes (and thus, different declarations) depending on whether they are being used in a 32-bit or 64-bit context.

To support 64-bit applications, Mac OS X has changed its data types and type usage in these broad areas:

- Size (and alignment) of base data types (for example, `long`)
- Choice of underlying base data types used in derived data types (such as `SInt32`)
- Base data types used in structure fields (such as those in `ScriptLanguageRecord`) and arguments to functions and methods
- API replacement and deprecation

These data types go by many names in various technology areas, but in terms of their underlying representation, the affected data types are one of those shown in "Data Type Changes" (page 19). In addition, a number of functions that use these base data types directly have been changed to use derived data types so that their underlying type can vary between 32-bit and 64-bit environments.

There are four common situations in which data types differ in the 64-bit world:

- 32-bit int data types that need to hold pointers. Because a pointer is 64 bits in length, these uses of int data types were changed to long data types.
- 32-bit int data types that could reasonably hold a larger data set in a 64-bit application. Because the viable number of objects in a data set can be much larger in a 64-bit application, these have been changed to long data types when it makes sense for such a large collection to exist. This determination varies on an API-by-API basis.
- 32-bit long data types that represent part of a data structure whose size and structure must not change. Because long is 64-bit on 64-bit architectures, these were changed to int to preserve compatibility.
- 32-bit long data types that represent counts, constants, or flags that cannot practically exceed the limits of a 32-bit integer (for example, the window identifier). Because long is 64-bit on 64-bit architectures, many such occurrences of long were changed to int where it does not make sense for a larger value to ever occur. This determination varies between APIs.

For example, the data type URefCon is defined in 32-bit applications as:

```
typedef unsigned long URefCon;
```

and in 64-bit applications as:

```
typedef void *URefCon;
```

These changes, which are sprinkled throughout all of the functions and data types in nearly every technology area, represent the vast majority of changes you will encounter.

New/Replaced/Deprecated APIs

In addition to API changes resulting from changes to the data types used in parameters and return values, other technology areas are changing significantly in the 64-bit world. Most of these changes are specific to C language APIs.

In certain technology areas (Carbon particularly), a few APIs have been deprecated for 32-bit use beginning in Mac OS X v10.5. Most of these APIs are not available in 64-bit applications. For example, any functions using FSSpec are gone, so you must use FSRef-based functions. This change affects a number of other related technology areas. There are also a number of other small, isolated changes in various APIs. You can find out more about these changes using the Research Assistant in Xcode.

In addition to these function-level deletions, some entire Carbon and QuickTime technologies will not be supported in 64-bit applications.

Technology Area Changes at a Glance

Changes to technology areas fall into these broad categories: Carbon, Cocoa/Objective C, QuickTime, and other C APIs. The sections that follow explain these changes in more detail.

Cocoa and Objective-C Application APIs

Most Objective-C APIs will not change substantially for 64-bit because most of the actual data types are sufficiently abstracted that the actual representation doesn't matter to the application. For example, a `CFNumber` or `NSNumber` object could have arbitrary representation under the hood. However, if you extract this information into standard C types, you must be careful about the sizes of those types in a 64-bit environment.

There are exceptions, however. A number of `typedef` declarations in AppKit and Foundation are changing for 64-bit. Specifically, data types whose base types were originally defined as an `enum` now have base types that specify the desired integer representation, such as `int` or `long`.

In addition, the types `NSInteger` and `NSUInteger` have been added. These are used to replace the use of `int` and `unsigned int` in a number of function declarations. Because these types have the same underlying base type in a 32-bit environment, developers should not need to change their code for type compatibility in 32-bit applications. In 64-bit applications, however, the base types for `NSInteger` and `NSUInteger` are `long` and `unsigned long`, respectively. Thus in 64-bit applications, you will need to replace these uses of `int` and `unsigned int` with `NSInteger` and `NSUInteger`.

Finally, some Objective-C method declarations may change, particularly those that use the underlying C data types `int` and `long` or types derived from them. These APIs will have the same issues as standard C APIs, though to a lesser degree.

For more detailed information, see *64-Bit Transition Guide for Cocoa*.

Kernel and I/O Kit APIs

Drivers and other kernel extensions should be transitioned to 64-bit executables beginning in Snow Leopard.

In addition, the I/O Kit has been extended somewhat in Mac OS X v10.5 to include support for 64-bit applications running on a 32-bit kernel. These changes are primarily in the form of additional methods in the `IOMemoryDescriptor` class.

Note: Some user-space frameworks attempt to include header files from the I/O Kit and kernel frameworks. The kernel definitions of some of these data types are not 64-bit–safe prior to Snow Leopard. For example, the kernel framework headers define `UInt32` as a `long`.

When working with the I/O Kit, you should be very careful to only include headers that were intended for user-space applications, and you should be careful to not use pre-10.5 SDK versions of the header when compiling the 64-bit side of your application.

All supported BSD kernel interfaces (KPIs) and system calls should be 64-bit clean beginning in Mac OS X v10.4. Software that uses these calls from user space should not require any modifications specific to the use of these interfaces assuming they use the specified types for arguments and return values.

QuickTime

The QuickTime Kit classes will be the primary interface into QuickTime. Methods that take or return native QuickTime C identifiers (in particular, Movie, MovieController, Track, and Media) are not supported in 64-bit applications.

Although the QuickTime C APIs are not deprecated for 32-bit use, you cannot directly use them in 64-bit applications.

The QuickTime Music Architecture (QTMA) API is not available in 64-bit applications. As an alternative, you should use the Core Audio API.

For more information on QuickTime API support in 64-bit applications, see *64-Bit Guide for Carbon Developers*.

Carbon

Some Carbon Managers and technologies are significantly reduced or unavailable in 64-bit applications. For detailed information, see *64-Bit Guide for Carbon Developers*.

Other C Application APIs

In general, most C API changes are in the use of `int` and `long` within function prototypes. Similarly, many data types based on `int` or `long` have changed sizes.

In Core Graphics a number of functions that previously returned `int` now return `bool` to more accurately reflect the information returned. Also, a number of Core Graphics functions now use `CGFloat` instead of `float`. The size of a `CGFloat` is larger in 64-bit applications than in 32-bit applications to allow for greater precision and range.

Finally, in Core Foundation and other technology areas, a number of uses of `int` and `long` have been replaced by aliases to these types in the form of named types such as `CFIndex`. Some of these are new types created because the underlying type changes from `int` to `long` (or vice versa) between 32-bit and 64-bit declarations. In other cases, these are preexisting types that simply do a better job at explaining the usage of a given parameter (for example, using a `CFIndex` to hold an index value).

For more detailed information, see *64-Bit Guide for Carbon Developers*.

Cross-Architecture Plug-in Support

In some cases, you may find it useful to support plug-ins written for an architecture other than the one your application is running on at the time. You may need this simply for debugging purposes, but this approach may also be useful if you want your application to support existing plug-ins on newer architectures. For example, audio software manufacturers may find it easier to drive adoption of 64-bit versions of their application if they also support existing 32-bit audio unit plug-ins.

Designing a host application to load a given plug-in is a highly specialized task. This chapter provides an overview of common approaches to doing this. This chapter assumes that you have already written a dummy plug-in loader that loads the plug-in into memory (even if it doesn't actually do anything with the plug-in yet).

In addition, this chapter describes several common interprocess communication APIs, explains how to pass large amounts of data between the host application and the plug-in helper host, and tells how to launch that host for a particular architecture.

Choosing a Helper Host Architecture Model

Before you can build a helper host, you must first choose an architecture model that accomplishes your needs. There are many possible design models for helper hosts, each with varying levels of functionality and difficulty. This section describes three such models and explains the problems you may encounter with each model.

Of these models, remote hosting is generally recommended because it is the easiest and most reliable. Limited function-call marshaling works when the scope of the API is limited. Full programmatic function-call marshaling, although described here, should usually be avoided because the exceptions and edge cases can make it impractical.

Programmatic Function-Call Marshaling

The first thing most developers consider doing when they design a helper host is trying to make every function call from the plug-in result in the same function being called in the host. With programmatic function-call marshaling, your application extracts the symbols from the plug-in, then generates custom library code to marshall arguments across address space boundaries (or even from one machine to another).

In general, the sheer number of exceptions and edge cases involved makes programmatic function-call marshaling highly impractical, and thus it should generally be avoided. However, this design may be reasonable if the plug-ins call only C APIs.

Such a design, although powerful, is tricky to get right, particularly when used across byte-order boundaries, because this design requires intimate understanding of every data structure involved to know whether or not a field should be byte swapped. For example, swapping various BSD-level networking data structures would be disastrous.

Fortunately, these data types are by far the exception rather than the rule, and can generally be ignored. However, the prevalent use of structure hiding (for example, using `void *` pointers and opaque types) essentially makes programmatic function-call marshaling nearly impossible, because there is no way to programmatically determine the underlying structure of a piece of data passed in this manner (and in many cases, the value may be meaningless in the context of a different process).

Opaque data structures are particularly an issue if a plug-in executes some code in the local process and passes the resulting data to closely related functions in the remote process. For example, file system references (FSRef) would not make sense when passed via IPC to a process running on a different architecture because of byte order differences in the underlying (opaque) structure. Similarly, file descriptors (POSIX) function differently depending on the application, and thus cannot be usefully passed via IPC.

If the plug-in must call arbitrary C++ or Objective-C class or instance methods on classes outside the plug-in itself, it becomes even more difficult to remotely execute function calls because of the need to maintain synchronization of class instances between the helper host and the main host. Since you cannot recompile the plug-in, replacing variables with accessor methods is impossible. This means that each function that potentially manipulates state must copy all of the state from the main host's notion of the class instance. Further, there is still some possibility that the host could update public class members without your knowledge, leading to potentially significant changes in your host application.

Finally, this method will not work transparently if the plug-in calls any functions that involve Mach ports, because port rights are not shared between the two processes unless they are explicitly passed from one process to the other or are inherited from the parent. Similarly, you should not try to marshal system calls in this way, because byte swapping at the lower levels of the operating system can be particularly complex.

Limited Function-Call Marshaling

Limited function-call marshaling is a far more realistic approach than fully programmatic marshaling. First, identify a set of (generally C) routines that call back into the host application. Then, replace those in the helper host with libraries that call across address space boundaries.

Because the scope of the supported API is limited, it is much more practical for you to support them through function-call marshaling, because you can hand-code routines for each function or class that that you intend to call across an address-space boundary instead of relying on programmatically generated functions and classes.

As with programmatic function-call marshaling, you must be particularly careful when working with pointers. If pointer arguments are of a known type and size, it is relatively easy to work with them. However, you may encounter problems if you do not know the size of the referenced object and if you need to byte swap or otherwise manipulate the pointer contents during the boundary crossing.

C++ and Objective-C classes are a bit harder. You can't simply pass pointers to classes, because they won't be valid on the other side of the communications channel. However, if the number of classes is limited, you can emulate class pointers by using stub classes in the helper host that contain an extra member variable that stores the address of the real class instance on the host side.

Similarly, you must emulate any callback pointers passed as arguments, because the callback pointer is meaningless in the context of the main host application. You can emulate these pointers either through message passing in the reverse direction or through RPC from the primary host into the helper host.

When you use limited function-call marshaling, your helper host can be very compact and completely transparent. However, your stub libraries must contain every function that you intend to override. For large plug-in APIs, this approach can be daunting, particularly if you are not in control of the API itself.

Remote Hosting

Remote hosting is strongly recommended for most helper host implementations because it is relatively easy to implement reliably. With remote hosting, instead of relying on knowledge of the plug-in architecture, you rely on your knowledge of the plug-in host itself. Because you are in control of the code in question, you will be aware of any changes to the API. Also, because the interface between a host and its built-in engine rarely involves callback pointers, you can use a simpler communication mechanism.

With remote hosting, you create a stripped-down version of your application that displays no user interface itself (except possibly a mirror of the menu bars with appropriate message passing to the main application). This miniature application should include a full set of data processing functionality. In this model, the host application passes a chunk of data to the helper host, then relies on the helper host to process the data just as the host application's built-in plug-in engine would.

You may choose to add a command-line flag (using `argc` and `argv`) to your application and, upon seeing that flag, call a separate initialization routine in which only the back-end functionality is configured. If you do, your helper host can simply be another running instance of your main application binary.

The biggest change you must make to support remote hosting is to maintain the state of your plug-in support engine through function calls instead of variable assignment (if you don't already do so). After you make that change throughout your host, the problem becomes a relatively simple set of changes to these functions:

- State changes to the plug-in layer of the host application must be reflected in the helper host.
- In the helper host, whenever a plug-in calls a callback that changes the state information stored in the helper host, your application must notify the main host so that the two remain in sync.
- Additional code must be added to handle passing of any data on which the plug-in will operate.

This state synchronization can be achieved through a relatively straightforward use of interprocess communication (discussed in "Using Interprocess Communication" (page 47)). For transport of large data, you should generally transfer the data using memory mapping. This technique is described in "Memory Mapping for Bulk Data Transport" (page 52).

Using Interprocess Communication

For interprocess communication, you can design a helper host using three broad models:

- **Remote procedure call (RPC) model**—You can use remote procedure calls to handle synchronization of your main host and helper host for you. There are many different RPC APIs available, each with different features and limitations, but at a high level they all behave similarly.

  A helper host designed using the RPC model contains a stub library. The functions in this library are created by RPC support tools based on an interface description. When your helper host (or a plug-in) calls these functions, the stub library sends a message to the main application, which causes the corresponding function in the main application to be called. When that function has completed, the result (if any) is returned in the same manner.

  An RPC-based helper host can be easier to write than one based on a pure client/server model, but it may suffer from limitations in the RPC API itself. For example, Mach RPC has no notion of complex data structures. As a result, you still need to write a fair amount of code to marshal arguments across address space boundaries. Mach RPC is discouraged, and is not officially supported.

You will learn about several RPC APIs in "Remote Procedure Call APIs" (page 48).

- **Client/server message passing model**—Open a socket-based or pipe-based connection to the main host application and use it to pass messages back and forth.

  A client/server helper host design consists of three main parts:

  - ❏ Create stub versions of the functions you want your helper host to call in the main host. When a plug-in or some other part of the helper host calls one of these stub functions, the function adds a message to a queue and waits on a condition variable. The plug-in thread then sleeps until a response is received and is posted to the buffer by the communication thread.
  - ❏ Create a communication thread in the helper host to transmit message entries from the buffer and store the responses in some other part of the same message entry.
  - ❏ Create a listener thread in the primary host to manage this communication on the other end.

  You will learn about several client/server messaging APIs in "Client/Server Messaging APIs" (page 49).

- **Memory mapping**—Create a region of memory that is simultaneously accessible to two (or more) processes and use it to share data between them.

  If you want to use memory mapping for passing messages back and forth, you would have to write a lot of additional code. For this reason, for most messaging needs, you should use a different IPC mechanism. However, when you need to move large quantities of information in bulk, client/server and RPC communication can get bogged down.

  For this reason, memory mapping is a common technique for passing large amounts of information in an out-of-band fashion (that is, in lieu of sending it through the primary IPC channel). For example, it would make sense for an audio plug-in helper host to communicate changes to parameters using a traditional IPC API and use memory mapping for a locking-free buffer between the main host and the helper host.

  You can find out more about memory mapping in "Memory Mapping for Bulk Data Transport" (page 52).

Remote Procedure Call APIs

There are three remote procedure calls that are commonly used in Mac OS X: distributed objects, Mach RPC, and Sun RPC. Of these, only distributed objects is a public API recommended for general use.

Distributed Objects

If you are calling Objective-C APIs, distributed objects is the recommended RPC API because it does a lot of the work for you.

If you are calling C or C++ APIs, you must wrap them in Objective-C classes before you can use this API. Before you consider doing so, you should read "Limited Function-Call Marshaling" (page 46).

For more information on distributed objects, see *Distributed Objects Programming Topics*.

Mach RPC

Mach RPC is not considered a public interface, and its direct use is not generally recommended. However, if you decide to use it, you can find information about it at the following URLs:

http://www.cs.cmu.edu/afs/cs/project/mach/public/www/doc/osf.html—OSF's Mach Documentation (from CMU)

http://www.cs.cmu.edu/afs/cs/project/mach/public/www/doc/tutorials.html—Mach Tutorials and Examples (from CMU)

Sun RPC

Sun RPC is beyond the scope of this document. You can find more information in the following places:

`rpc` man page

`rpcgen` man page

`xdr` man page

`rpcinfo` man page

`portmap` man page

Sun RPC is generally not recommended for new designs.

Client/Server Messaging APIs

Mac OS X supports several client/server messaging APIs, including Apple events, BSD sockets, and pipes (standard input and output, for example). These APIs are described in the sections that follow.

- "Apple Events" (page 49)
- "Socket Programming" (page 50)
- "Standard Input and Output" (page 50)
- "Message Queues" (page 52)

These APIs can be used by themselves in some cases. Many APIs lend themselves to sending messages back and forth easily and do not depend on receiving a response from the other end. However, when you need to get a response from a function or method call, you may want to implement a higher level construct on top of these messaging APIs to make it easier to handle these asynchronous responses. One such method is described in "Message Queues" (page 52).

Apple Events

A common API for interprocess communication in Mac OS X is Apple events. The Apple Events API is a fairly straightforward API for low-bandwidth IPC, and you are probably already using it in your application. If so, you can add additional message types for communication between your application and the plug-in host.

For more information, see *Apple Events Programming Guide*.

Socket Programming

The most common API for simple interprocess communication is an old standby, sockets. There are a number of different Mac OS X technologies for working with sockets, including:

- `CFNetwork` API (described in *CFNetwork Programming Guide*)
- `NSSocketPort` API (described in *NSSocketPort Class Reference*)
- BSD socket API (described in `socket`)

Each of these APIs implements the same underlying message, a bidirectional stream of bytes between both ends. Stream-based messaging presents a problem if your helper host needs to concurrently support multiple plug-ins, however, because you will need to multiplex data from multiple sources. You can solve this problem by using message queues, as described in "Message Queues" (page 52).

As an added bonus, communication via sockets is not limited to a single machine. If you are writing software that can benefit from distributed computing, such remote communication can be a significant benefit.

If you are writing an audio helper host, most of the work is done for you beginning in Mac OS X v10.4. The AUNetSend and AUNetReceive audio units can make helper hosting relatively painless to implement, whether on a local machine or remotely. However, with these plug-ins, *all* information passes through the TCP/IP stack even if the destination is on the local machine.

Keep in mind two caveats if you use TCP/IP for passing the actual data back and forth instead of just passing control information. First, the latency of remote communications is *not* insignificant. If this matters in your application (for example, an audio application), you *must* compensate for this latency or quality will suffer greatly. Second, the amount of information being sent is substantial, and thus, for performance reasons, socket programming may not be ideal for hosting a large number of individual plug-ins on a helper host. If you expect a large number of non-native plug-ins, you should generally use memory mapping to pass data from the main host to the helper host, as described in "Memory Mapping for Bulk Data Transport" (page 52).

With those caveats in mind, sockets also open up the possibility of alternative software usage models. For example, you might design an audio application so that the front end can run on a small, low-power, fanless computer in a studio control room, with all of the heavy lifting performed by a separate computer in another room. You could implement the user interface by temporarily hosting a local copy of plug-ins when a user wants to show their user interface, then sending control change messages across the wire to the actual host (where the plug-ins are all running with no UI displayed). Then, use TCP/IP for sending *only* the raw audio from the audio interface. For audio play-through while recording, you should mix the incoming audio into the output on the front-end computer as the very last step in processing.

The details of creating and using sockets are beyond the scope of this document. For additional information, consult the documents listed above. You may also find useful information in the UNIX Socket FAQ, which can be found at http://www.developerweb.net/forum/. This FAQ includes code examples that illustrate how to use TCP/IP and UNIX domain sockets at the BSD API level.

Standard Input and Output

Another common API for interprocess communication is standard input and output. This API provides a pair of unidirectional streams. Much like socket programming, the stream-based nature of standard input and output requires you to keep additional state information if you need to associate responses to messages with the original message. A good way to solve that problem is through the use of message queues, as described in "Message Queues" (page 52).

One thing that makes standard input and output convenient is that they are largely set up for you. Every process in a UNIX-based system has standard input and output automatically. You can take advantage of these to communicate between a parent process (your main application) and its children (your helper host).

To communicate with child processes in Cocoa, you should use the `NSTask` API, described in NSTask Class Reference. For more information on this method, read "Creating and Launching an NSTask" and "Ending an NSTask".

Alternatively, in BSD tools, you can accomplish the same thing at a file descriptor level using a few low-level APIs as shown in this example:

```
#include <stdlib.h>
comm_channel *startchild(char *path)
{
        comm_channel *channel = malloc(sizeof(*channel));
        pid_t childpid;
        int in_descriptors[2];
        int out_descriptors[2];

        /* Create a pair of file descriptors to use for communication. */
        if (pipe(in_descriptors) == -1) {
                fprintf(stderr, "pipe creation failed.\n");
                goto error_exit;
        }
        if (pipe(out_descriptors) == -1) {
                fprintf(stderr, "pipe creation failed.\n");
                goto error_exit;
        }
        /* Create a new child process. */
        if ((childpid = fork()) == -1) {
                fprintf(stderr, "fork failed.\n");
                goto error_exit;
        }
        if (childpid) {
                /* Parent process */
                channel->in_fd = in_descriptors[0];
                close(in_descriptors[1]);
                channel->out_fd = out_descriptors[1];
                close(out_descriptors[0]);
                return channel;
        } else {
                /* Child process */
                if (dup2(in_descriptors[1], STDOUT_FILENO) == -1) {
                        fprintf(stderr, "Call to dup2 failed.\n");
                        goto error_exit;
                }
                close(in_descriptors[0]);

                if (dup2(out_descriptors[0], STDIN_FILENO) == -1) {
                        fprintf(stderr, "Call to dup2 failed.\n");
                        goto error_exit;
                }
                close(out_descriptors[1]);

                execl(path, path, NULL);

                /* If we get here, something went wrong. */
                fprintf(stderr, "Exec failed.\n");
                goto error_exit;
        }
        return channel;
    error_exit:
        free(channel);
        perror("msg_send");
        return NULL;
}
```

> **Note:** If you use a function like this one, you should always specify an absolute path to your helper application.

Message Queues

Message queues provide a way for one process to communicate with another process in a flexible fashion over a stream-based transport without requiring that the two processes behave in a lockstep fashion at all times. You can build message queues on top of either bidirectional communication channels, such as sockets, or on top of pairs of unidirectional communication channels, such as pipes or standard input and output.

A message queue at its simplest consists of a linked list of message structures. Each message structure contains an outgoing message and a location in which the response will be stored. Depending on how you write your code, it may contain a callback, to be executed upon completion, or a single handler that calls the right function based on the original message type.

On each end, you should have a thread to handle messages from the socket. You can use your run loop thread as a handler thread if you are writing a traditional application, or you can use a separate message thread if you prefer to use lower-level socket APIs.

The code for managing a message queue is relatively straightforward, locking issues notwithstanding. A complete code example is provided in the companion files associated with this document. The companion files archive can be downloaded from the sidebar when viewing this document as HTML at the ADC Reference Library (developer.apple.com).

Memory Mapping for Bulk Data Transport

For moving large quantities of data between two applications, unless you are communicating over a network, you should generally avoid most traditional message-passing algorithms because of the inherent CPU overhead and latency involved. Instead, you should consider a shared memory design using mmap.

The following example shows how to create a shared memory region between a process and its child:

```
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/dirent.h>
#include <fcntl.h>
#include <stdlib.h>

/* Create the map file and fill it with bytes. */
char *create_shm_file(char *progname, int length)
{
    int fd, i;
    char *filename=malloc(MAXNAMLEN+1);
    char *ret;
    char byte = 0;
    sprintf(filename, "/tmp/%s-XXXXXXXX", progname);
    ret = mktemp(filename);

    fd = open(filename, O_RDWR|O_CREAT, 0600);
    for (i=0; i<length; i++) {
        write(fd, &byte, 1);
    }
    return ret;
}

/* Map the file into memory in a read-write fashion */
void *map_shm_file(char *filename, int length)
{
    int fd = open(filename, O_RDWR, 0);
    void *map;
    if (fd == -1) return NULL; /* Could not open file */
    map = mmap(NULL, length, PROT_READ|PROT_WRITE,
        MAP_FILE|MAP_SHARED, fd, 0);
    return map;
}
```

Using this sample code, the two applications can rendezvous using a file as a shared memory buffer between them. As long as both applications use the same file, any changes made by one application will be seen by the other and vice versa. Your application can then assign pieces of this buffer to be used for various tasks just as though you were using anonymous memory returned by a call to `malloc`.

If you intend to work with page-sized regions, you should also take note of the functions described in the `mpool` manual page. However, for most purposes, you should write your own pool allocator if you need to regularly allocate and deallocate shared memory.

For more information on the functions used in the example above, see the man pages for `mmap`, `open`, and `mktemp`.

Note: Memory-mapped files do not grow automatically. Thus, you cannot map beyond the file's EOF marker. If you need to extend the length of a mapping file, you must do so using normal file I/O routines prior to remapping the file.

You will probably find it easier to use a separate map file each time that you require another large chunk of shared space, however. Usually, you should only need to map another large piece of shared memory when loading a new plug-in or creating a new plug-in instance. Thus, the mapping file name and descriptor can be stored as an additional piece of metadata associated with a given plug-in instance.

A good way of working with shared memory is for you to use a lock-free ring buffer design. In such a design, each communication endpoint reads from two variables but writes only to one. In this way, both sides know where in the buffer the other endpoint is working.

For example:

```
typedef struct ringbuffer {
    void *buffer;
    int buflen;
    int readpos;
    int writepos;
} *ringbuffer;
```

```
#define BYTES_TO_READ(ringbuffer) (ringbuffer->writepos - \
    ringbuffer->readpos + \
    ((ringbuffer->readpos > ringbuffer->writepos) * \
        (ringbuffer->buflen)))

/*  Use >= here because if readpos and writepos are equal,
    the buffer must be assumed to be empty.  Otherwise,
    the buffer would start out full. For this reason,
    the writepos must not be allowed to overtake the read
    position, so subtract one from the final value.
*/
#define BYTES_TO_WRITE(ringbuffer) (ringbuffer->readpos - \
    ringbuffer->writepos + \
    ((ringbuffer->writepos >= ringbuffer->readpos) * \
        ringbuffer->buflen) - 1)
```

The code reading from this buffer knows that it can always read from readpos forwards up to writepos (or if writepos is less than readpos, it can read to the end of the buffer, then read from the start of the buffer up to writepos). After reading, the read code updates readpos to reflect the location of the last byte read.

In a similar fashion, the code writing to this buffer knows that it can safely write from the writepos position until it reaches readpos, wrapping around the end of the buffer if necessary. After writing, the write code updates writepos to reflect the location of the last byte written.

Because only one process will ever modify either readpos or writepos, no synchronization between the two processes is required. Note, however, that the reading code *must* protect readpos against other threads within that process, and the writing code must do the same for writepos.

Launching the Helper Host

After you've build a helper host, the next step is to determine the architecture of the plug-in. For PEF/CFM plug-ins, it is safe for you to assume that the plug-in contains 32-bit PowerPC executable code. For Mach-O plug-ins, the method you should use varies according to the version of Mac OS X being used.

For backward compatibility with versions of Mac OS X prior to 10.5, your application should use the detection code presented in the *CheckExecutableArchitecture* sample code. This sample code is straightforward and presents a fairly easy way to determine which architecture to use for loading existing plug-ins.

In Mac OS X v10.5 and later, you should use the CFBundle API. This API is safer as a long-term solution, because it will support any binary format that is supported by that particular version of Mac OS X, thus freeing you from the need to alter the code as new binary formats are introduced. The relevant functions are:

```
CFArrayRef CFBundleCopyExecutableArchitecturesForURL(CFURLRef url);
CFArrayRef CFBundleCopyExecutableArchitectures(CFBundleRef bundle);
```

The next step is to execute the helper host, choosing the appropriate architecture in the process. In Mac OS X v10.5 and later, the recommended way to launch an executable using a particular architecture is through an extension to posix_spawn. This API is described in the manual page for posix_spawn, posix_spawnattr_init, and the related manual pages linked from those pages. The extension for choosing an architecture to launch is described in the manual page for posix_spawnattr_setbinpref_np.

To support helper hosts on Mac OS X v10.4, you can use separate copies of your helper host for each processor architecture instead of a universal binary, then launch whichever version is appropriate.

Performance Optimization

When transitioning your application, kernel extension, or other code to 64-bit, you may notice a performance decrease, depending on your code. While in some cases this is caused by simply moving more data around, in many cases, it is caused by fairly minor data structure layout issues and other easily corrected issues. This chapter briefly describes some of these issues and explains how to correct them.

Data Structure Bloat

Software compiled as a 64-bit executable is likely to use more memory than its 32-bit counterpart because of the increased size of data and pointers. However, many of these increases are exacerbated by changes in data alignment. Such increases, which can affect performance, are easily avoided.

As noted in "Data Type and Alignment Tips" (page 26), data structure alignment is different for 64-bit executables. The following snippet shows a small data structure:

```
struct mystruct {
    char member_one[4];
    void *member_two;
    int member_three;
    void *member_four;
    int member_five;
}
```

In a 32-bit executable, this data structure takes 20 bytes. The pointers are 4 bytes long, and are aligned on a 4-byte boundary. If the structure is allocated dynamically, even though you only request 20 bytes, the memory allocator will actually assign it a memory region that is 32 bytes long (the nearest power-of-2 boundary).

In a 64-bit executable, this data structure bloats to a whopping 36 bytes and is padded to 40 bytes (because it must be 8-byte-aligned), which means that if allocated dynamically, it occupies 64 bytes.

To explain why, assume that the variable starts at address zero (0). The variable member_one takes 4 bytes, so the next member would normally start at byte 4.

Because member_two is a pointer, it must be aligned on an 8-byte boundary. Thus, its address must be divisible by 8. Because 4 is not divisible by 8, the structure must be padded. The result is a 4-byte gap between member_one and member_two.

The variable member_three is not a problem—it needs to be aligned at a 4-byte boundary and would normally begin at offset 16, so it does not generate a gap.

The variable member_four, however, must be aligned at an 8-byte boundary. The compiler generates another 4-byte gap, and this variable begins at offset 24. Finally, the variable member_five begins at offset 32 and ends at offset 36.

Fortunately, by making small changes to the order of the structure members, you can reduce the size increase dramatically:

```
struct mystruct {
    char member_one[4];
    int member_three;
    void *member_two;
    void *member_four;
    int member_five;
}
```

By shifting the third member up before the second member, the variables `member_two` and `member_three` now naturally fall on 8-byte boundaries. As a result, this structure is reduced to a mere 28 bytes, and is padded to 32 bytes by the compiler to make it 8-byte aligned. It also takes the same 32-byte dynamic memory allocation as it did in the 32-bit version of the code.

The easiest way to guarantee maximum data structure efficiency is to start with the members aligned to the largest byte boundary and work your way down to the members aligned to the smallest. Place the 8-byte-aligned `long` and `pointer` members first, followed by 4-byte-aligned `int` values, followed by two-byte-aligned `short` values, and finally end with byte-aligned `char` values.

```
struct mystruct {
    void *member_two;
    void *member_four;
    int member_five;
    int member_three;
    char member_one[4];
}
```

Cache-Line Misses

Changes in data structure alignment can cause differences in cache-line hits and misses. While this usually does not have a significant impact on performance, in certain degenerate cases it can have a devastating impact. Consider the following code:

```
for (i=0; i<columns; i++) {
    for (j=i; j<rows; j++) {
        arr[j][i] = ...
    }
}
```

This code performs a series of operations on an array. However, it iterates through the array in a fairly inefficient way. C stores arrays in row-major order, but this code iterates in a fashion more appropriate for a column-major array—iterating through the first entry in each row, then the second entry in each row, and so on.

If you are particularly unlucky and your stride length is on the same order as the length of a cache line (or longer), you could easily find that every few accesses, you end up invalidating a cache line containing a previously used data row that you will use again on the next pass through the loop. As a result, the CPU cache fails to boost performance.

Where this often comes back to hurt 64-bit application performance is when the size of data structure members changes to be just slightly larger so that the entire data structure no longer fits within the number of cache lines available, and thus on that last row, the first cache line is invalidated, wiping out the cache for the first row. Then, the first row is processed a second time, invalidating the cache for the next row, and so on.

To fix this performance regression, change the code to iterate through the array in the opposite order so that you perform numerous accesses within each row. With that change, most of the accesses within a given row are serviced out of the cache, and you should see a tremendous performance benefit. (In many cases, you may also see some benefit even in the 32-bit version of your application.)

Of course, this is a somewhat contrived example. Usually, problems involving degenerate cache line behavior are much harder to spot. If you think you may be seeing a cache-related performance regression, the Shark application can provide valuable insight into the number of cache misses your code generates.

Once you have determined that this is the source of your problems, you should do whatever you can do reduce the cache collisions by reducing the size of the data structures themselves, reducing your stride length when iterating through arrays, or reorganizing your in-memory data structures in a way that increases temporal locality of reference.

Avoiding Unaligned Accesses

If you create data structures with packed alignment, you may see a performance regression caused by unaligned access penalties. For example:

```
#pragma pack(1)
typedef struct
{
        long a;
        int b;
        int b1;
        int b2;
        long double c;
} myStruct;
#pragma options align=reset
```

Ideally, the variable c should be aligned on a 16-byte boundary. In a 32-bit environment, it ends up aligned correctly because the variable a is 4 bytes long. In a 64-bit environment, this variable ends up offset by 4 bytes from its ideal position because the variable a is 8 bytes long.

While a single unaligned access penalty is relatively small, the cumulative effects can be significant. Given a loop that assigns the value zero (0) repeatedly to variables a, b, and c, you can improve performance of the loop by more than 20% by simply moving the variable b2 after the variable c. Of course, this wrecks performance in the 32-bit version because variable c is then unaligned.

The best solution is to not use packing pragmas except when working with data structures that mandate it (structures that represent hardware registers or are intended to be written to a file, sent across a network connection, shared across address space boundaries, and so on) and let the compiler use natural alignment for all data structures that you work with on a regular basis in your software.

For maximum efficiency, you should also reorder this structure to place the variable `c` first, followed by the variable `a`, followed by the remaining variables (in order from largest alignment size to smallest). This achieves optimal packing efficiency while avoiding alignment penalties regardless of CPU architecture and often obviates the need for packing pragmas.

Kernel Extensions and Drivers

The kernel (including the I/O Kit) in Mac OS X v10.5 and earlier was a 32-bit environment. Beginning in Snow Leopard, the kernel is moving to a 64-bit environment on some hardware. This chapter describes the rationale for this change and explains how it affects you as a developer of device drivers or other kernel extensions.

This chapter is divided into four sections:

- "Why a 64-bit Kernel?" (page 61)—Explains why Mac OS X is transitioning to a 64-bit kernel.
- "What You Must Do" (page 62)—Describes the basic steps involved in transitioning a kernel extension or driver to 64-bit.
- "64-Bit Kernel Data Type Changes" (page 63)—Describes additional data type changes specific to kernel-space code in a 64-bit environment.
- "Additional Tips For 64-Bit KEXTs" (page 64)—Provides helpful tips for transitioning and debugging 64-bit kernel extensions and drivers.

Why a 64-bit Kernel?

In modern operating systems, applications run in a virtual address space. Thus, an application's notion of an address is not the same as the physical hardware's notion of an address. The benefit is that applications can "see" a huge address space even if there is not sufficient RAM to support it.

Underneath this abstraction is a virtual memory subsystem within the operating system kernel. This subsystem manages the mappings between an application's view of the world and the hardware's view. The virtual address space is broken up into fixed size blocks, called pages, each of which can be individually mapped to an arbitrary hardware address. The operating system then tells the CPU to associate these virtual pages with certain physical hardware addresses using a table known as the page table. Regardless of page table organization, this table eventually grows until it contains (at minimum) an entry for every page of physical RAM in the system.

In addition to these page table entries, the operating system maintains additional data structures that keep track of which physical pages are associated with which processes, which pages are free, and so on. Of these, the most common (by volume) is the `vm_page` structure (usually seen in the form of `vm_page_t` pointers to it), which describes various properties of free pages and resident (in physical memory) logical pages, such as their physical address, their paging state, the number of wired memory maps that reference the page, and so on. The OS maintains a `vm_page` structure for every physical page of RAM.

For a computer with 64 GB of RAM, given a 4 KB page size, the OS must manage almost 17 million pages of physical RAM, each of which has a page table entry and a `vm_page` structure. In total, these data structures would potentially consume well over a gigabyte of kernel memory by themselves. In a 32-bit (4GB) address space, this would significantly limit the kernel address space available for other purposes.

The space constraints are compounded by other data structures (mbuf storage, for example) that should ideally be allowed to scale with the size of available memory. By moving to a 64-bit address space (when run on supported hardware), the Mac OS X kernel can accommodate these data structures in large memory configurations.

In the client version of Mac OS X, the kernel defaults to 32-bit operation on all computers. In Mac OS X Server, however, the kernel defaults to 64-bit mode on XServe (April 2009 and later) and Mac Pro (March 2009 and later) computers with large RAM configurations.

For a complete list of Macintosh computers that support the 64-bit kernel, see http://support.apple.com/kb/HT3770.

In both client and server versions, you can boot supported computers into 64-bit mode by holding down the 6 and 4 keys at startup time or by adding `arch=x86_64` to your `boot-args` string using the `nvram` command. In Mac OS X Server, you can force 32-bit booting by holding down the 3 and 2 keys or by adding `arch=i386` to your `boot-args` string.

What You Must Do

As a driver developer, you `must` update your drivers with 64-bit binaries. The 64-bit kernel cannot load 32-bit kernel extensions. Fortunately, because the I/O Kit is a relatively modern environment with few legacy design constraints, most kernel extensions can be adapted fairly easily to 64-bit. Many drivers "just work" after changing the compile settings. However, there are a few steps you must take along the way.

Recompile your code

In a 64-bit kernel environment, device drivers and kernel extensions must be made 64-bit clean and compiled as 64-bit executables. This process is essentially the same as for any other 64-bit code. In particular, you should be aware of the changes described in "Major 64-Bit Changes" (page 19), "Making Code 64-Bit Clean" (page 23), and "Compiling 64-Bit Code" (page 37). There is one small difference, however: you must use GCC 4.2 or later when compiling 64-bit kernel extensions. (64-bit applications can be compiled with GCC 4.0.)

Update dependency information

In the 64-bit kernel, the kernel exports *only* KPI dependencies, not the general kernel dependencies or unsupported dependencies. For example, `com.apple.kpi.iokit` is supported, but `com.apple.kernel.iokit` is not.

Note: If you need to support operating systems that predate the availability of the KPI symbol sets, you must use a separate driver bundle with a different `Info.plist` file for those older operating systems.

Stop using unsupported symbols

In addition, the exported KPI symbol lists are cleaned up for the 64-bit environment. If your code uses functions that are not exported by the 64-bit kernel, you will receive compile-time or load-time errors. You must fix these by moving off of these APIs and moving to APIs that are supported for 64-bit. You can learn more about these APIs in *Kernel Framework Reference*.

Update user-client code

Device drivers that talk directly to a user-space application without using I/O Kit families (such as user clients and the I/O Kit families themselves) may need to be changed in order to correctly communicate with applications. A 32-bit kernel extension may have to communicate with a 64-bit application and vice-versa, which can cause problems with data structure size, alignment, and so on.

For more information about problems you may encounter when passing data between applications and kernel extensions with different word sizes, see "Data Type and Alignment Tips" (page 26) and "Additional Tips For 64-Bit KEXTs" (page 64).

For more about user clients and device interfaces in general, read *I/O Kit Fundamentals*.

Complete the move to `IODMACommand`

On Intel-based Macintosh computers with 64-bit Intel processors, device drivers that support direct memory access (DMA) *must* be updated to use the `IODMACommand` class beginning with Mac OS X v10.4.7. (Device drivers on PowerPC may be updated to use this class, but doing so is not required.)

The `IODMACommand` class provides bounce buffers for devices that do not support 64-bit physical addressing, and uses direct mapping for devices that do. For more information, see the documentation for `IODMACommand`.

Check for kernel-specific data type changes

Two key data types used in the kernel have different underlying base types in a 64-bit kernel environment. This occasionally can cause problems when printing some numeric values and when subclassing other classes. These changes are described in "64-Bit Kernel Data Type Changes" (page 63).

Fix bugs

After making these overarching changes, the remaining fixes, you should look for the problem described in "Additional Tips For 64-Bit KEXTs" (page 64) and correct them if necessary.

64-Bit Kernel Data Type Changes

In addition to the general C data type changes described in "Data Type Changes" (page 19), the underlying type behind two kernel-specific data types has changed.

Type name	32-bit type	64-bit type
`SInt32`	`long`	`int`
`UInt32`	`unsigned long`	`unsigned int`

These changes pose two potential problems: format strings and C++ method overriding.

First, these changes affect format strings for `printf` and `IOLog` calls. When printing these values, you can either modify your code to use `%ld` when compiling 32-bit and `%d` when compiling 64-bit or cast both values to an `int` and use `%d` explicitly to avoid the warning.

Second, these changes can affect overridden methods, as described in "Additional Tips For 64-Bit KEXTs" (page 64).

Additional Tips For 64-Bit KEXTs

In addition to all of the common issues and changes described in "Making Code 64-Bit Clean" (page 23), here are several other common kernel-specific mistakes you should watch for when porting your device driver or other kernel extension to 64-bit:

Use the Correct SDK Version

Although user-space code may be compiled against the 10.5 SDK, you must compile your kernel-space driver code against the 10.6 SDK when compiling the 64-bit slice. To build a KEXT that supports existing 32-bit architectures, you must use per-architecture build settings, as described in "Using Architecture-Specific Flags" (page 39).

Check the Signatures of Overridden Methods

As mentioned in "64-Bit Kernel Data Type Changes" (page 63), the data types `UInt32` and `SInt32` are of type `long` in the 32-bit kernel environment, but are of type `int` in the 64-bit kernel environment. These changes present a potential problem for overridden C++ methods.

If a C++ method has an argument of type `UInt32` (for example) and a subclass overrides that method but defines the parameter as being of type `long`, in the 32-bit environment, the subclass version overrides the method in the superclass correctly because the types are equivalent.

When recompiled for 64-bit, however, the subclass version is still of type `long`, but the original class version is now of type `int`. Because the two methods no longer have the same signature, the subclass version does not override the method in the superclass, and as a result, the method's behavior will depend on which type of integer is used by the calling function. This behavior is almost certainly not what you want.

For this reason, it is imperative that you check all classes that override existing classes and make sure that any methods you write use the exact same named types as any methods they are overriding.

Avoid Truncating Virtual Addresses

Inside the kernel, references to virtual memory addresses are often handled using nonpointer types. The most common use is the value returned by the `getVirtualAddress` method of `IOMemoryMap`. Be careful to assign these addresses only to variables with 64-bit integer types such as `mach_vm_address_t` and never to variables with 32-bit integer types such as `UInt32`.

Use the Large Zero Page Flag

To help debug pointer truncation issues, pass the `-no_shared_cr3` flag as part of your boot arguments. (See "Building and Debugging Kernels" for information about setting boot arguments.) This flag causes the kernel to enforce a 128 TB page zero in the kernel and provides similar benefits to the 4 GB page zero in user-space applications.

With a 64-bit kernel, the kernel itself occupies the top 128TB of virtual address space, while the currently active user-space application occupies the bottom 128TB (or 4 GB for a 32-bit application). Because the user mappings do not overlap with the kernel mappings, they do not need to be flushed when switching into kernel space and back. (Page permissions are used to ensure that the kernel's address space cannot actually be accessed by the user-space application even though the mappings are in place.)

As a side effect, however, because this unified page table is used, pages in the currently executing user-space application remain accessible after transitioning into the kernel. If a 32-bit application (or a 64-bit application without a 4GB page zero) is running, any pointer used by the kernel that gets truncated to 32 bits may end up pointing into a valid address range that contains the application's code or data. As a result, it is unsafe to assume that a truncated pointer in the kernel will result in an illegal access panic. (Further, such a stray pointer may cause applications to crash in hard-to-diagnose ways.)

By specifying the `-no_shared_cr3` flag during debugging and testing, a separate kernel mode page table is swapped in and the TLB is flushed during these transitions, thus ensuring that accessing a truncated pointer in the kernel results in an illegal access exception, which triggers a panic.

Note: The `-no_shared_cr3` flag behaves somewhat differently with a 32-bit kernel. For most 64-bit applications, because the bottom 4GB region is usually unmapped, the kernel can be mapped into this region. Thus, when switching into the kernel, the page table remains the same and no TLB flush is needed. The `-no_shared_cr3` flag forces a page table reload and TLB flush during this transition.

Use IODMACommand for Devices with Limited Addressing Support

Some devices can only handle physical addresses that fit into 32 bits. To the extent that it is possible to use 64-bit addresses you should do so, but for these devices, you can either use `IODMACommand` or the `initWithPhysicalMask` method of `IOBufferMemoryDescriptor` to allocate a bounce buffer within the bottom 4 GB of physical memory.

Fix User Client Code

Communication between a user-space application and kernel code is basically the same whether you are in a 32-bit kernel or a 64-bit kernel. In Mac OS X v10.5, your code potentially must deal with 32-bit or 64-bit Intel or PowerPC applications, and compatibility with 32-bit PowerPC applications is still required in Mac OS X v10.6.

That said, if your user-space framework only supports 32-bit applications currently, the transition to a 64-bit kernel (and other changes in Mac OS X v10.6) may prompt you to update this code to support 64-bit applications.

Here are some tips for cross-architecture communication:

- **Maintain consistent structure sizes**—Where possible, build your data structures in such a way that they do not change in size between architectures. If your structures contain pointers, maintaining consistent structure sizes is more difficult, but not impossible. One way to make pointer-laden structures consistent is to use a union between the pointer and a larger data type. For example:

  ```
  struct my_struct {
      int a;
      char b;
      union {
          void *c;
          uint64_t pad_01;
      };
  };
  ```

- **Take advantage of `IOUserClient::initWithTask`**—If you are writing a user client, the `IOUserClient::initWithTask` method has two forms. One form takes an additional `OSDictionary` parameter that provides information about the client. To determine whether the remote process is a 32-bit PowerPC client running in Rosetta, include this code in your user client:

  ```
  if (properties && properties->getObject(kIOUserClientCrossEndianKey)) {
      // Connecting application is a 32-bit PowerPC
      // application running in Rosetta.  Byte
      // swap as needed.
  }
  ```

  For more information, see the *SimpleUserClient* sample.

- **Use magic numbers**—A magic number is a number that you place inside a data structure to allow you to determine whether the structure is valid, is in the correct byte order, and so on. It can be as simple as a version number, so long as the version number is never zero and is not the same when its bytes are reversed.

  If you are communicating in some way other than a user client, you can determine the byte order of the remote application using a magic number, and with a bit more effort, you can also determine the word length (32-bit or 64-bit) using this technique.

  For example, consider the following structure and assignment statements:

  ```
  struct mystruct {
      uint32_t magic;
      ....
  };
  struct mystruct mystruct_instance;
  mystruct_instance.magic=0x32160804;
  mystruct_instance.pad = 0xffffffff;
  ```

  If you receive such a structure as a block of data, you can trivially determine the byte order as follows:

  ```
  void *blob = ...
  uint32_t *magic = blob;
  if (*magic == 0x32160804) {
      // no swap needed
  } else {
      // byte swap needed
  }
  ```

  If your data structures change size in 64-bit applications, you should use a different magic number to identify these 64-bit structures.

  As an alternative, if you can guarantee that the four bytes following the magic number will never be zero, you can check for 64-bit applications like this:

  ```
  void *blob = ...
  uint32_t *magic = blob;
  if (*magic == 0x04081632) {
      // Application is 32-bits, byte swap needed.
  } else if (*magic == 0x00000000) {
      // Application is 64-bit PowerPC.
  } else if (*magic == 0x32160804) {
      // Application is built for the same
      // architecture as this code, but may
      // be either 32-bit or 64-bit on Intel.
      magic++
      if (*magic == 0x00000000) {
          // remote app is 64-bit Intel.
      } else {
          // remote app is 32-bit, built for the
          // same architecture as this code.
      }
  }
  ```

As a slight variation, if you cannot guarantee the four bytes will be nonzero but can guarantee that they will not be 0xffffffff, you could use a signed long value instead, then use any hexadecimal value of 0x80000000 or greater for the magic number so that it will be sign extended on 64-bit architectures, then replace 0x00000000 with 0xffffffff in both places in the above example.

- **Declare byte order and word size explicitly**—This is similar to the concept of magic numbers except that the remote end of the communication identifies its architecture explicitly. For example, you might write code like this:

```
#if defined(__LP64__)
    #ifdef __LITTLE_ENDIAN__
        #define HOST_ORDER=1
    #else
        #define HOST_ORDER=2
    #endif
#elif defined(__LITTLE_ENDIAN__)
        #define HOST_ORDER=3
#else
        #define HOST_ORDER=4
#endif

struct mystruct {
    int order;
};
struct mystruct mystruct_instance;
mystruct_instance.order = HOST_ORDER;
```

This still puts the burden of reading the field squarely on the code receiving the structure, but makes it much easier.

- **Pre-convert data structures to a consistent size and order**—As an alternative to these techniques, you can write code in user space to convert all pointers to uint64_t values in a consistent byte order and make your kernel code convert it again if needed.

- **Replace outdated** IOConnectMethod* **calls**—The following functions are not supported in a 64-bit environment:

```
IOConnectMethodScalarIScalarO
IOConnectMethodScalarIStructureO
IOConnectMethodScalarIStructureI
IOConnectMethodStructureIStructureO
IOConnectMethodScalarIScalarO
IOConnectMethodScalarIStructureO
IOConnectMethodScalarIStructureI
IOConnectMethodStructureIStructureO
```

You should instead use the IOConnectCall* functions:

```
IOConnectCallMethod
IOConnectCallAsyncMethod
IOConnectCallStructMethod
IOConnectCallAsyncStructMethod
IOConnectCallScalarMethod
IOConnectCallAsyncScalarMethod
```

Simulating a 64-Bit Address Space with mmap and munmap

This appendix contains sample code (Listing A-1) that demonstrates how to use `mmap` and `munmap` to simulate a large address space using offsets into a file.

Listing A-1 Using `mmap` and `munmap`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/mman.h>

/*
* max_mmap_size(filename)
* A not-so-great attempt to determine at run time how much
* of a given file can be mapped in.  Usually, the amount will
* be the minimum of filesize or 0x7fffffff. The math is a
* wee bit off on the binary search.
*/
size_t
max_mmap_size(const char *file) {
        int fd;
        struct stat sbuf;
        size_t retval;

        off_t min = 0, max = (size_t)(~0);

        fd = open(file, O_RDONLY);
        if (fd == -1)
                return 0;

        fstat(fd, &sbuf);
        if ((sbuf.st_mode & S_IFMT) != S_IFREG) {
                close(fd);
                return 0;
        } else if ((off_t)max > sbuf.st_size)
                max = (sbuf.st_size);

        retval = (size_t)((max + min) / 2);

        while (min != max) {
                caddr_t t;
                errno = 0;

                t = mmap(0, retval, PROT_READ, MAP_SHARED, fd, 0);
                if (t == (caddr_t)-1 && errno != 0) {
                        if (errno != EINVAL)
```

```
                                return 0;
                        /* too large */
                        max = retval - 1;
                } else {
                        min = retval + 1;
                }

                munmap(t, retval);
                retval = (size_t)((min + max) / 2);
        }

        close(fd);
        return (size_t)min - 1;
}


/*
* sum(filename, starting_offset, size)
* Sum the bytes in the file specified by the <filename> parameter,
* starting at the specified offset and continuing for <size> bytes.
* This is very slow, but this function will eventually touch
* every byte of the file segment you have asked for.
*/
long long
sum(const char *file, off_t offset, size_t count) {
        long long retval = 0;
        struct stat sbuf;
        int fd;
        caddr_t t;
        char *cp;

        fd = open(file, O_RDONLY);

        if (fd == -1) {
                return -1LL;
        }

        fstat(fd, &sbuf);
        if (offset > sbuf.st_size)
                return -1;

        sbuf.st_size -= offset;
        if (sbuf.st_size < count)
                count = sbuf.st_size;
        if (count == 0) {
                retval = -1LL;
                goto out;
        }

        errno = 0;
        fprintf(stderr, "mmap(NULL, %u, PROT_READ, MAP_FILE|MAP_SHARED, %d,
%qu)\n",
                count, fd, offset);

        t = mmap(NULL, count, PROT_READ, MAP_FILE|MAP_SHARED, fd, offset);
        if (t == (caddr_t)-1 && errno != 0) {
                fprintf(stderr, "cannot mmap %s: %s\n", file, strerror(errno));
                retval = -1LL;
                goto out;
        }

        for (cp = t; cp < &t[count]; cp++)
                retval += *cp;
        munmap(t, count);

out:
        close(fd);
        return retval;
}


main(int ac, char **av) {
        size_t max;
        long long s = 0;
        long long t, off = 0;

        if (ac != 2) {
                fprintf(stderr, "usage: %s <filename>\n", av[0]);
                exit(1);
        }

        max = max_mmap_size(av[1]);
        printf("max = %u\n", max);

        /*
         * Cycle through the given filename, in <max>-byte
         * segments.
         */
        while ((t = sum(av[1], off, max)) != -1) {
                s += t;
                off += max;
        }

        printf("sum = %qd\n", s);
        return 0;
}
```

Document Revision History

This table describes the changes to *64-Bit Transition Guide*.

Date	Notes
2010-01-15	Fixed typographical errors and other minor nits.
2009-04-17	Updated to cover 64-bit drivers in Mac OS X v10.6.
2008-04-08	Fixed minor typographical errors and links.
2007-05-10	Updated for Mac OS X v10.5.
2005-11-09	Made minor typographical fixes and added mention that a G5 processor is required to run 64-bit binaries.
2005-08-11	Changed terminology from "fat binary" to "universal binary." Added mention of the __LP64__ macro.
2005-07-07	Fixed minor typographical errors.
2005-06-04	Fixed minor typographical errors.
2005-04-29	First public release
2004-11-02	Updated version information and system call tables.
2004-06-29	Initial (developer seed) revision.