64-Bit Transition Guide for Cocoa Touch

Developer

Contents

Tables and Listings

About 64-Bit Cocoa Touch Apps

Important: This is a preliminary document for an API or technology in development. Although this document has been reviewed for technical accuracy, it is not final. This Apple confidential information is for use only by registered members of the applicable Apple Developer program. Apple is supplying this confidential information to help you plan for the adoption of the technologies and programming interfaces described herein. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future seeds of the API or technology.

When desktop operating systems transitioned from 32-bit to 64-bit addressing, 64-bit apps were critical to the OS transition. Now, iOS is getting a similar desktop-class architecture. Starting with iOS 7, you can build iOS apps that take advantage of 64-bit processors. An app that supports 64-bit processing almost always gains improved performance when compared with a 32-bit app running on the same device.

At a Glance

Among other architecture improvements, a 64-bit ARM processor includes twice as many integer and floating-point registers as earlier processors do. As a result, 64-bit apps can work with more data at once for improved performance. Apps that extensively use 64-bit integer math or custom NEON operations see even larger performance gains. In a 64-bit process, pointers are 64 bits and some integer types, once 32 bits, are now 64 bits. Many data types in system frameworks, especially UIKit and Foundation, have also changed. Generally, 64-bit apps run more quickly and efficiently than their 32-bit equivalents. However, the transition to 64-bit code brings with it increased memory usage. If not managed carefully, the increased memory consumption can be detrimental to an app's performance.

Convert Your App to a 64-Bit Binary After Updating It for iOS 7

Xcode can build your app with both 32-bit and 64-bit binaries included. This combined binary requires a minimum deployment target of iOS 6 or later. The 64-bit binary runs only on iOS 7 or later. If you have an existing app, you should first update your app for iOS 7 and then port it to run on 64-bit processors. By updating it first for iOS 7, you can remove deprecated code paths and use modern practices. If you are creating a new app, target iOS 7 and compile 32-bit and 64-bit versions of your app.

When iOS is executing on a 64-bit device, iOS includes separate 32-bit and 64-bit versions of the system frameworks. When all apps running on the device are compiled for the 64-bit runtime, iOS never loads the 32-bit versions of those libraries, which means that the system uses less memory and launches apps more quickly. Because all of the built-in apps already support the 64-bit runtime, it is to everyone's benefit that all apps running on 64-bit devices be compiled for the 64-bit runtime, especially apps that support background processing. Even apps that are not performance sensitive gain from this memory efficiency.

Chapter: "Major 64-Bit Changes" (page 8), "Optimizing Memory Performance" (page 27)

Convert and Then Test Your App

The architecture for 64-bit apps on iOS is almost identical to the architecture for OS X apps, making it easy to create a common code base that runs in both operating systems. Converting a Cocoa Touch app to 64-bit follows a similar transition process as the one for Cocoa apps on OS X. Pointers and some common C types change from 32 bits to 64 bits. Code that relies on the `NSInteger` and `CGFloat` types needs to be carefully examined.

Start by building the app for the 64-bit runtime, fixing any warnings that occur as well as searching your code for specific 64-bit issues. For example:

- Make sure all function calls have a proper prototype.
- Avoid truncating 64-bit values by accidentally assigning them to a 32-bit data type.
- Ensure that calculations are performed correctly in the 64-bit version of your app.
- Create data structures whose layouts are identical in the 32-bit and 64-bit versions of your app (such as when you write a data file to iCloud).

Test your app on a 64-bit iOS device because some changes can be detected only when running on actual hardware. When your app is running, profile its performance and memory usage, improving both as necessary.

Chapters: "Converting Your App to a 64-Bit Binary" (page 13), "Optimizing Memory Performance" (page 27)

How to Use This Document

Read "Major 64-Bit Changes" (page 8) to understand the changes in the 64-bit runtime, and then read "Converting Your App to a 64-Bit Binary" (page 13) for concrete details on how to update your app. To ensure that your app's memory footprint remains modest, read "Optimizing Memory Performance" (page 27).

Prerequisites

You need to have some familiarity with creating apps to understand the information in this guide. For more information, see *Start Developing iOS Apps Today*.

In some cases, a detailed understanding of ANSI C programming is necessary to make the most of the information in this guide.

See Also

The details of both the 32-bit and the 64-bit application binary interfaces can be found in *iOS ABI Function Call Guide*.

Major 64-Bit Changes

When different pieces of code must work together, they must follow standard agreed-upon conventions about how code should act. Conventions include the size and format of common data types, as well as the instructions used when one piece of code calls another. Compilers are implemented based on these conventions so that they can emit binary code that works together. Collectively, these conventions are referred to as an **application binary interface (ABI)**.

iOS apps rely on a low-level application binary interface and coding conventions established by the Objective-C language and the system frameworks. Starting with iOS 7, some iOS devices use 64-bit processors and offer both a 32-bit and a 64-bit runtime environment. For most apps, the 64-bit runtime environment differs from the 32-bit runtime environment in two significant ways:

- In the 64-bit runtime, many data types used by Cocoa Touch frameworks (as well as the Objective-C language itself) have increased in size or have stricter memory alignment rules. See "Changes to Data Types" (page 8).
- The 64-bit runtime requires proper function prototypes to be used when making function calls. See "Changes to Function Calling" (page 11).

Changes to Data Types

In the C and Objective-C languages, the built-in data types do not have a predefined size or alignment in memory. Instead, each platform defines the characteristics of the built-in types. This means that, within the restrictions defined in the language standard, a platform can use values that best match the underlying hardware and operating system. The 64-bit runtime on iOS changes the sizes of many built-in data types. At a higher level, many data types used by Cocoa Touch frameworks have also changed. This section describes the changes to the data types commonly used in Objective C code.

Two Conventions: ILP32 and LP64

The 32-bit runtime uses a convention called ILP32, in which integers, `long` integers, and pointers are 32-bit quantities. The 64-bit runtime uses the LP64 convention; integers are 32-bit quantities, and `long` integers and pointers are 64-bit quantities. These conventions match the ABI for apps running on OS X (and similarly, the Cocoa Touch conventions match the data types used in Cocoa), making it easy to write interoperable code between the two operating systems.

Table 1-1 describes all of the integer types commonly used in Objective-C code. Each entry includes the size of the data type and its expected alignment in memory. The highlighted table entries indicate places where the LP64 convention differs from the ILP32 convention. These size differences indicate places where your code's behavior changes when compiled for the 64-bit runtime. The compiler defines the `__LP64__` macro when compiling for the 64-bit runtime.

Table 1-1 Size and alignment of integer data types in OS X and iOS

Integer data type	ILP32 size	ILP32 alignment	LP64 size	LP64 alignment
`char`	1 byte	1 byte	1 byte	1 byte
`BOOL`, `bool`	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**
`long long`	8 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**
`NSInteger`	4 bytes	4 bytes	**8 bytes**	**8 bytes**
`CFIndex`	4 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**

To summarize the changes:

- The size of pointers increased from 4 bytes to 8 bytes.
- The size of `long` integers increased from 4 bytes to 8 bytes. The `size_t`, `CFIndex`, and `NSInteger` types also increased from 4 bytes to 8 bytes.
- The alignment of `long long` integers and the data types based on it (`fpos_t`, `off_t`) has increased from 4 bytes to 8 bytes. All other types use the natural alignment, meaning that data elements within a structure are aligned at intervals corresponding to the width of the underlying data type.

Table 1-2 shows the floating-point types commonly used in iOS and OS X. Although the built-in data types do not change in size, in the 64-bit runtime the `CGFloat` type changes from a `float` to a `double`, providing a larger range and accuracy for these values in Quartz and in other frameworks that use the Core Graphics types. (Floating-point types always use natural alignment.)

Table 1-2 Size of floating-point data types in OS X and iOS

Floating-point type	ILP32 size	LP64 size
`float`	4 bytes	4 bytes
`double`	8 bytes	8 bytes
`CGFloat`	4 bytes	**8 bytes**

The 64-bit ARM environment uses a little-endian environment; this matches the 32-bit iOS runtime used by devices with ARM processors that support the ARMv7 architecture.

Impact of Data Type Changes on Your App

When you compile your app for both the 32-bit and 64-bit runtime environments, keep in mind that your app's behavior differs according to the environment it runs in and can result in performance differences or severe compatibility problems. Here are some of the things you need to consider:

- Increased memory pressure

  Because so many fundamental types have increased in size, the 64-bit version of your app uses more memory than the 32-bit version does. For example, even something as simple as a linked list is more expensive when compiled for the 64-bit runtime. Expect to spend more time optimizing the performance of the 64-bit version of your app. You may also need to consider different strategies when designing your app's data structures, including the instance variables used in your Objective-C classes. For some examples, see "Optimizing Memory Performance" (page 27).

- Exchanging data between 64-bit and 32-bit software

  If you ship both 32-bit and 64-bit software, sometimes both versions need to interoperate on the same data. For example, a user might access a file stored in iCloud from both 32-bit and 64-bit devices. Or you might be creating a game that sends network data between devices. In both runtimes, make sure that the data being read or written uses a common memory layout, meaning that the sizes and offsets of every data element are identical.

- Calculations might produce different results

A 64-bit integer supports a vastly larger range of possible values than a 32-bit integer supports. If your app uses an integer type that changed from 32 bits to 64 bits, the results might be different in the 64-bit version of your app. Specifically, a calculation that exceeds the maximum value of a 32-bit integer overflows on a 32-bit integer but not on a 64-bit integer. Other such edge cases exist.

- Values may be truncated when data is copied from a larger data type to a smaller data type

  Some apps assume that two data types are the same rather than safely converting between them. In the 64-bit runtime, previous assumptions about data types may no longer be correct. For example, assigning an `NSInteger` value to an `int` type works in the 32-bit runtime because both types are 32-bit integers. But in the 64-bit runtime, the two are not the same type and so data may be lost.

Changes to Function Calling

When a function call is made, the compiler emits code to pass the parameters from the caller to the callee. For example, an ABI might specify that the caller places its parameters into registers, or it might specify that the caller pushes the values onto a stack in memory. Normally, if you aren't writing assembly language, the calling conventions are rarely important. But in the 64-bit runtime, sometimes you need to be aware of how functions are called. Functions that accept a variable number of parameters (variadic functions) are called using a different set of conventions than calls to functions that accept a fixed set of parameters.

Impact of Function Call Changes on Your App

When coding an app that targets the 64-bit runtime, your app must always call a function using its exact definition. As a result:

- Every function must have a prototype.
- When you cast a function, you must be careful that the cast version of the function has the same signature as the original function. In particular, avoid casting a a function to a form that takes a different set of parameters (such as casting a function pointer that takes a variadic pattern to one that takes a fixed number of parameters). Additional rules apply if your code directly calls message passing functions in the Objective-C runtime. (See "Dispatch Objective-C Messages Using the Method Function's Prototype" (page 24).)

Changes to Objective-C

If you are writing low-level code that targets the Objective-C runtime directly, you can no longer access an object's `isa` pointer directly. Instead, you need to use the runtime functions to access that information.

Other Changes to the 64-Bit Runtime

The 64-bit ARM instruction set is significantly different from the 32-bit instruction set. If your app includes any assembly language code, you need to rewrite it to use the new instruction set. You also need a more detailed description of the 64-bit calling conventions in iOS, because the conventions do not exactly match the ARM standard. For more information, see *iOS ABI Function Call Guide*.

Summary

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.
- Avoid pointer and `long` integer truncation during arithmetic operations (or other arithmetic problems caused by the change in integer types).
- Fix alignment issues caused by changes in data type sizes.
- Ensure that memory structures that are shared between the 32-bit and 64-bit runtimes share a similar layout.
- Rewrite any assembly language code so that your code uses the new 64-bit opcodes and runtime.
- Avoid casting variadic functions to functions that take a fixed number of parameters, or vice versa.

Converting Your App to a 64-Bit Binary

At a high level, here are the steps to create an app that targets both the 32-bit and the 64-bit runtime environments:

1. **Install Xcode 5.**
2. **Open your project.** Xcode prompts you to modernize your project. Modernizing the project adds new warnings and errors that are important when compiling your app for 64-bit.
3. **Update your project settings to support iOS 6 or later.** You cannot build a 64-bit project if it targets an iOS version earlier than iOS 6.
4. **Change the Architectures build setting in your project to "Standard Architectures (including 64-bit)."**
5. **Update your app to support the 64-bit runtime environment.** The new compiler warnings and errors will help guide you through this process. However, the compiler does not do all of the work for you; use the information in this document to help guide you through investigating your own code.
6. **Test your app on actual 64-bit hardware.** iOS Simulator can also be helpful during development, but some changes, such as the function calling conventions, are visible only when your app is running on a device.
7. **Use Instruments to tune your app's memory performance.**

The remainder of this chapter describes problems that frequently occur when porting a Cocoa Touch app to the 64-bit runtime environment. Use these sections to guide your own efforts to investigate your code.

Do Not Cast Pointers to Integers

There are few reasons to cast pointers to an integer type. By consistently using pointer types, you ensure that all of your variables are sized large enough to hold an address.

For example, the code in Listing 2-1 casts a pointer to an `int` type because it wants to perform arithmetic on the address. In the 32-bit runtime, this code works because an `int` type and a pointer are the same size. However, in the 64-bit runtime, a pointer is larger than an `int` type, so the assignment loses some of the pointer's data. Here, because the pointer is being advanced by its native size, you can simply increment the pointer.

Listing 2-1 Casting a pointer to `int`

```
int *c = something passed in as an argument....
int *d = (int *)((int)c + 4); // Incorrect.
int *d = c + 1;               // Correct!
```

If you must convert a pointer to an integer type, always use the `uintptr_t` type to avoid truncation. Note that modifying pointer values via integer math and then converting it back into a pointer can violate basic type aliasing rules. This can lead to unexpected behavior from the compiler as well as processor faults when a misaligned pointer is accessed.

Use Data Types Consistently

Many common programming errors result when you don't use data types consistently throughout your code. Even though the compiler warns you of many problems that result from using data type inconsistently, it's also useful to see a few variations of these patterns so that you can recognize them in your code.

When calling a function, always match the variable that receives the results to the function's return type. If the return type is a larger integer than the receiving variable, the value is truncated. Listing 2-2 shows a simple pattern that exhibits this problem. The `PerformCalculation` function returns a `long` integer. In the 32-bit runtime, both `int` and `long` are 32 bits, so the assignment to an `int` type works, even though the code is incorrect. In the 64-bit runtime, the upper 32 bits of the result are lost when the assignment is made. Instead, the result should be assigned to a `long` integer; this approach works consistently in both runtimes.

Listing 2-2 Truncation when assigning a return value to a variable

```
long PerformCalculation(void);

    int  x = PerformCalculation(); // incorrect
    long y = PerformCalculation(); // correct!
```

The same problem occurs when you pass in a value as a parameter. For example, in Listing 2-3 the input parameter is truncated when executed in the 64-bit runtime.

Listing 2-3 Truncation of an input parameter

```
int PerformAnotherCalculation(int input);
    long i = LONG_MAX;
```

```
    int x = PerformCalculation(i);
```

In Listing 2-4, the return value is also truncated in the 64-bit runtime, because the value returned exceeds the range of the function's return type.

Listing 2-4 Truncation when returning a value

```
int ReturnMax()
{
    return LONG_MAX;
}
```

All of these examples result from code that assumes that `int` and `long` are identical. The ANSI C standard does not make this assumption, and it is explicitly incorrect when working in the 64-bit runtime. By default, if you modernized your project, the `-Wshorten-64-to-32` compiler option was automatically enabled, so the compiler automatically warns you about many cases where a value is truncated. If you did not modernize your project, you should explicitly enable that compiler option. Optionally, you may want to include the `-Wconversion` option, which is more verbose, but finds more potential errors.

In a Cocoa Touch app, look for the following integer types and make sure you are using them correctly:

- `long`
- `NSInteger`
- `CFIndex`
- `size_t` (the result of calling the `sizeof` intrinsic operation)

And in both runtime environments, the `fpos_t` and `off_t` types are 64 bits in size, so never assign them to an `int` type.

Enumerations Are Also Typed

In the LLVM compiler, enumerated types can define the size of the enumeration. This means that some enumerated types may also have a size that is larger than you expect. The solution, as in all the other cases, is to make no assumptions about a data type's size. Instead, assign any enumerated values to a variable with the proper data type.

Common Type-Conversion Problems in Cocoa Touch

Cocoa Touch, notably Core Foundation and Foundation, add additional situations to look for, because they offer ways to serialize a C data type or to capture it inside an Objective-C object.

NSInteger changes size in 64-bit code. The `NSInteger` type is used throughout Cocoa Touch; it is a 32-bit integer in the 32-bit runtime and a 64-bit integer in the 64-bit runtime. So when receiving information from a framework method that takes an `NSInteger` type, use the `NSInteger` type to hold the result.

Although you should never make the assumption that an `NSInteger` type is the same size as an `int` type, here are a few critical examples to look for:

- Converting to or from an `NSNumber` object.
- Encoding and decoding data using the `NSCoder` class. In particular, if you encode an `NSInteger` on a 64-bit device and later decode it on a 32-bit device, the decode method throws an exception if the value exceeds the range of a 32-bit integer. You may want to use an explicit integer type instead (see "Use Explicit Integer Data Types" (page 19)).
- Working with constants defined in the framework as `NSInteger`. Of particular note is the `NSNotFound` constant. In the 64-bit runtime, its value is larger than the maximum range of an `int` type, so truncating its value often causes errors in your app.

CGFloat changes size in 64-bit code. The `CGFloat` type changes to a 64-bit floating point number. As with the `NSInteger` type, you cannot assume that `CGFloat` is a `float` or a `double`. So use `CGFloat` consistently. Listing 2-5 (page 16) shows an example that uses Core Foundation to create a `CFNumber`. But the code assumes that a `CGFloat` is the same size as a `float`, which is incorrect.

Listing 2-5 Use `CGFloat` types consistently

```
// Incorrect.
CGFloat value = 200.0;
CFNumberCreate(kCFAllocatorDefault, kCFNumberFloatType, &value);

// Correct!
CGFloat value = 200.0;
CFNumberCreate(kCFAllocatorDefault, kCFNumberCGFloatType, &value);
```

Be Careful When Performing Integer Computations

Although truncation is the most common problem, you may also run into other problems related to the integer change. This section includes some additional guidance you can use to update your code.

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. Unsigned values are zero extended (not sign extended) when promoted to a larger type.
2. Signed values are always sign extended when promoted to a larger type, even if the resulting type is unsigned.
3. 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 `int`, `long`, and `long long` types and may be either `signed` or `unsigned` types. Decimal numbers are always treated as `signed` types.
4. The sum of a signed value and an unsigned value of the same size is an unsigned value.

Listing 2-6 shows an example of unexpected behavior resulting from these rules along with an accompanying explanation.

Listing 2-6 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 in the 32-bit runtime, the result is `-1` (`0xffffffff`). When the code is run in the 64-bit runtime, 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 4)**. Next, that value is promoted to a larger type. This promotion does not cause sign extension.

Solution: To fix this problem in a 32-bit-compatible way, cast `b` to a `long` integer. 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 2-7 shows a related example with an accompanying explanation.

Listing 2-7 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` integer can fit into a signed `int` type, the result of this promotion is signed.

Second, when the shift completed, the result was stored into a `long` integer. Thus, the 32-bit signed value represented by `(a << 31)` was sign extended **(rule 2)** when it was promoted to a 64-bit value (even though the resulting type is unsigned).

Solution: Cast the initial value to a `long` integer prior to the shift. The short is promoted only once—this time, to a 64-bit type (at least when compiled as a 64-bit executable).

Working with Bits and Bitmasks

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

Do not assume a data type has a particular length. If you are shifting through the bits stored in a variable of type `long` integer, use the `LONG_BIT` value to determine the number of bits in a `long` integer. 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 `long` integers, because the width differs between 32-bit and 64-bit runtimes. 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 in the 64-bit runtime, the usual fixed-width mask works 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, write the mask as the bitwise inverse of its inverse, as shown in Listing 2-8.

Listing 2-8 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.

Create Data Structures with Fixed Size and Alignment

When data is shared between the 32-bit and 64-bit versions of your app, you may need to create data structures whose 32-bit and 64-bit representations are identical, mostly when the data is stored to a file or transmitted across a network to a device that may have the opposite runtime environment. But also keep in mind that a user might back up their data stored on a 32-bit device and then restore that data to a 64-bit device. So interoperability of data is a problem you must solve.

Use Explicit Integer Data Types

The C99 standard provides built-in data types that are guaranteed to be a specific size, regardless of the underlying hardware architecture. You should use these data types when your data must be a fixed size or when you know that a particular variable has a limited range of possible values. By choosing a proper data type, you get a fixed-width type you can store in memory and you also avoid wasting memory by allocating a variable whose range is much larger than you need.

Table 2-1 lists the C99 types along with the ranges of allowed values for each.

Table 2-1 C99 explicit integer types

Type	Range
int8_t	-128 to 127
int16_t	-32,768 to 32,767

Type	Range
int32_t	-2,147,483,648 to 2,147,483,647
int64_t	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
uint8_t	0 to 255
uint16_t	0 to 65,535
uint32_t	0 to 4,294,967,295
uint64_t	0 to 18,446,744,073,709,551,615

Be Careful When Aligning 64-Bit Integer types

In the 64-bit runtime, the alignment of all 64-bit integer types changes from 4 bytes to 8 bytes. Even if you specify each integer type explicitly, the two structures may still not be identical in both runtimes. In Listing 2-9, the alignment changes even though the fields are declared with explicit integer types.

Listing 2-9 Alignment of 64-bit integers in structures

```
struct bar {
    int32_t foo0;
    int32_t foo1;
    int32_t foo2;
    int64_t bar;
};
```

When this code is compiled with a 32-bit compiler, the field `bar` begins 12 bytes from the start of the structure. When the same code is compiled with a 64-bit compiler, the field `bar` begins 16 bytes from the start of the structure. Four bytes of padding are added after `foo2` so that `bar` is aligned to an 8-byte boundary.

If you are defining a new data structure, organize the elements with the largest alignment values first and the smallest elements last. This structure organization eliminates the need for most padding bytes. If you are working with an existing structure that includes misaligned 64-bit integers, you can use a pragma to force the proper alignment. Listing 2-10 shows the same data structure, but here the structure is forced to use the 32-bit alignment rules.

Listing 2-10 Using pragmas to control alignment

```
#pragma pack(4)
struct bar {
    int32_t foo0;
    int32_t foo1;
    int32_t foo2;
    int64_t bar;
};
#pragma options align=reset
```

Use this option only when necessary, because there is a performance penalty for misaligned accesses. For example, you might use this option to maintain backward compatibility with data structures already in use in a 32-bit version of your app.

Allocate Memory Using sizeof

Never call `malloc` with an explicit size (for example, `malloc(4)`) to allocate space for a variable. Always use `sizeof` to obtain the correct size for any structure or variable you allocate. Search your code for any instance of `malloc` that isn't followed by `sizeof`.

Update Format Strings to Support Both Runtimes

Print functions such as `printf` can be tricky when you are writing code to support both runtimes, because of the data type changes. To solve this problem for pointer-sized integers (`uintptr_t`) and other standard types, use the various macros defined in the `inttypes.h` header file.

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

Table 2-2 Standard format strings

Type	Format string
`int`	`%d`
`long`	`%ld`
`long long`	`%lld`

Type	Format string
`size_t`	`%zu`
`ptrdiff_t`	`%td`
any pointer	`%p`

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

Type	Format string
`int[N]_t` (such as `int32_t`)	`PRId[N]` (such as `PRId32`
`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 2-11.

Listing 2-11 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);
```

Take Care with Functions and Function Pointers

Function calls in the 64-bit runtime are handled differently than functions in the 32-bit runtime. The critical difference is that function calls with variadic prototypes use a different sequence of instructions to read their parameters than functions that take a fixed list of parameters. Listing 2-12 shows the prototypes for two functions. The first function (`fixedFunction`) always takes a pair of integers. The second function takes a variable number of parameters (but at least two). In the 32-bit runtime, both of the function calls in Listing 2-12 use a similar sequence of instructions to read the parameter data. In the 64-bit runtime, the two functions are compiled using conventions that are completely different.

Listing 2-12 64-bit calling conventions vary by function type

```
int fixedFunction(int a, int b);
int variadicFunction(int a, ...);

int main
{
    int value2 = fixedFunction(5,5);
    int value1 = variadicFunction(5,5);
}
```

Because the calling conventions are much more precise in the 64-bit runtime, you need to make sure that a function is always called correctly, so that the callee always finds the parameters that the caller provided.

Always Define Function Prototypes

When compiling using the modernized project settings, the compiler generates errors if you attempt to make a function call to a function that does not have an explicit prototype. You must provide a function prototype so that the compiler can determine whether the function is a variadic function or not.

Function Pointers Must Use the Correct Prototype

If you pass a function pointer in your code, its calling conventions must stay consistent. It should always take the same set of parameters. Never cast a variadic function to a function that takes a fixed number of parameters (or vice versa). Listing 2-13 is an example of a problematic piece function call. Because the function pointer

was cast to use a different set of calling conventions, a caller is going to place the parameters in a place that the called function is not expecting. This mismatch may cause your app to crash or to exhibit other unpredictable behaviors.

Listing 2-13 Casting between variadic and nonvariadic functions results in an error

```
int MyFunction(int a, int b, ...);

int (*action)(int, int, int) = (int (*)(int, int, int)) MyFunction;
action(1,2,3); // Error!
```

Important: If you cast a function in this way, the compiler does not generate an error or warning and its unpredictable behavior is not visible in iOS Simulator. Always test on a real device before shipping your app.

Dispatch Objective-C Messages Using the Method Function's Prototype

An exception to the casting rule described above is when you are calling the `objc_msgSend` function or any other similar functions in the Objective-C runtime that send messages. Although the prototype for the message functions has a variadic form, the method function that is called by the Objective-C runtime does not share the same prototype. The Objective-C runtime directly dispatches to the function that implements the method, so the calling conventions are mismatched, as described previously. Therefore you must cast the `objc_msgSend` function to a prototype that matches the method function being called.

Listing 2-14 shows the proper form for dispatching a message to an object using the low-level message functions. In this example, the `doSomething:` method takes a single parameter and does not have a variadic form. It casts the `objc_msgSend` function using the prototype of the method function. Note that a method function always takes an `id` variable and a selector as its first two parameters. After the `objc_msgSend` function is cast to a function pointer, the call is dispatched through that same function pointer.

Listing 2-14 Using a cast to call the Objective-C message sending functions

```
- (int) doSomething:(int) x { ... }
- (void) doSomethingElse {
    int (*action)(id, SEL, int) = (int (*)(id, SEL, int)) objc_msgSend;
    action(self, @selector(doSomething:), 0);
}
```

Be Careful When Calling Variadic Functions

Variable argument lists (`varargs`) do not provide type information for the arguments, and the arguments are not automatically promoted to larger types. 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 get incorrect results.

In particular, if your `varargs` function expects a `long` integer and you pass in a 32-bit value, the `varargs` function contains 32 bits of data and 32 bits of garbage from the next argument (which you lose as a result). Likewise, if your `varargs` function is expecting an `int` type and you pass in a `long` integer, you get only half of the data, and the rest incorrectly appears in the argument that follows. (One example of this occurs if you use an incorrect `printf` format strings.)

Use the Built-in Synchronization Primitives

Sometimes, apps implement their own synchronization primitives to improve performance. The iOS runtime environment provides a full complement of primitives that are optimized and correct for each CPU on which they are running. This runtime library is updated as new architectural features are added. Existing 32-bit apps that rely on the runtime library automatically take advantage of new CPU features introduced in the 64-bit world. If you implemented custom primitives in a previous version of your app, now you might be relying on instructions or paths that are orders of magnitude slower than the built-in primitives. Instead, always use the built-in primitives.

Never Hard-Code the Virtual Memory Page Size

Most apps do not need to know the size of a virtual memory page, but some use it for buffer allocations and some framework calls. Starting with iOS 7, the page size may vary between devices in both the 32-bit and 64-bit runtimes. Therefore, always use the `getpagesize()` function to get the size of the page.

Go Position Independent

The 64-bit runtime environment supports only Position-Independent Executables (PIE). By default most modern apps are built as position independent. If you have something that is preventing your app from building as a position independent code, such as a statically linked library or assembly code, you need to update this code when porting your app to the 64-bit runtime. Position-independence is also highly recommended for 32-bit apps.

For more information, see *Building a Position Independent Executable*.

Don't Forget your 32-bit Version

Customers are already using the 32-bit version of your app and will continue to do so for years. Converting your code to use 64-bit integer types everywhere may sound like a good solution, but it isn't. A 64-bit processor can perform 64-bit integer operations just as quickly as 32-bit integer operations, but a 32-bit processor performs 64-bit calculations much more slowly. If you use 64-bit types consistently, your 32-bit app is going to lose performance compared to your current version of your app. Instead, use integer types that match the range of values you need to compute and store. If the results fit in a 32-bit integer, use a 32-bit integer. See "Use Explicit Integer Data Types" (page 19).

More generally, when designing an app that works in both the 32-bit and 64-bit runtime environments, you should make decisions that are correct for both environments, meaning that you should make decisions that work equally well in both environments. When you can't make a single code base that runs equally well in each environment, create separate solutions that are the right ones for each runtime.

Make Your App Run Well in the 32-Bit Runtime

Currently, an app that is written for the 64-bit runtime environment must also support the 32-bit runtime. So you need to make apps that work well when running in either environment. Usually this means creating a single design that works well in both environments. Occasionally, you may need to design specific solutions for each runtime environment.

For example, you might be tempted to use 64-bit integers throughout your code; both environments support 64-bit integer types, and using only a single integer type everywhere simplifies your app's design. If you use 64-bit integers everywhere, your app will run more slowly in the 32-bit runtime. A 64-bit processor can perform 64-bit integer operations just as quickly as 32-bit integer operations, but a 32-bit processor performs 64-bit calculations much more slowly. And in both environments, the variable may use more memory than is necessary. Instead, a variable should use an integer type that matches the range of values you expect it to hold. If a calculation always fits in a 32-bit integer, use a 32-bit integer. See "Use Explicit Integer Data Types" (page 19).

Optimizing Memory Performance

Because of improvements in 64-bit processors, 64-bit apps have the potential to perform faster than 32-bit apps. At the same time, the 64-bit runtime increases the size of pointers and some scalar data, resulting in a larger memory footprint for your app. A larger memory footprint results in increased pressure on processor caches and virtual memory and can adversely affect performance. When developing a 64-bit app, it is critical to profile and optimize your app's memory usage.

For a comprehensive discussion on optimizing memory usage, see *Memory Usage Performance Guidelines*.

Profile Your App

Before attempting to optimize your app's memory usage, you should first create standard tests that you can run against both the 32-bit and 64-bit versions of your app. Standardized tests can measure the penalty for compiling a 64-bit version of your app when compared with the 32-bit version. It also provides a way to measure improvements as you optimize your app's memory usage. For at least one test, use a minimal footprint—for example, the app has just been opened and shows an empty document. For other tests, include a variety of data sizes, including at least one test with a very large data set. (A complex app may require multiple sets of test data, each covering a subset of the app's features.) The goal for these tests is to measure whether the memory usage varies as the type or amount of data changes. If a particular kind of data causes the 64-bit version of your app to use dramatically more memory than its 32-bit counterpart, that is a great place to start looking for improvements.

Common Memory Usage Problems

The potential memory usage problems you may encounter are organized here, along with some guidance for handling them.

Foundation Objects May Be Expensive for Small Payloads

Many classes in Foundation offer a flexible feature set, but to provide that flexibility, they use more memory than a simpler data structure. For example, using an `NSDictionary` object to hold a single key-value pair is significantly more expensive than simply allocating a variable to hold the data. Creating thousands of such dictionaries wastes memory. Using Foundation objects when it isn't appropriate isn't a new problem, but when running in the 64-bit runtime, those objects use even more memory.

Choose a Compact Data Representation

Find places where you can use a better data representation than you have. For example, you are storing a calendar date using the following data structure:

```
struct date
{
    NSInteger second;
    NSInteger minute;
    NSInteger hour;
    NSInteger day;
    NSInteger month;
    NSInteger year;
};
```

This structure is 24 bytes long; in the 64-bit runtime, it takes 48 bytes, just for a date! A more compact representation simply stores the number of seconds since a particular time. When necessary, you convert this compact representation to the calendar date and time.

```
struct date
{
    uint32_t seconds;
};
```

Pack Data Structures

To align data structures, compilers sometimes add padding to a structure. For example:

```
struct bad
{
```

```
    char       a;        // offset 0
    int32_t    b;        // offset 4
    char       c;        // offset 8
    int64_t    d;        // offset 16
};
```

This structure includes 14 bytes of data, but because of padding, it takes up 24 bytes of space.

A better design sorts the fields from largest to smallest alignment.

```
struct good
{
    int64_t    d;        // offset 0
    int32_t    b;        // offset 8
    char       a;        // offset 12;
    char       c;        // offset 13;
};
```

This version adds no additional padding.

Use Fewer Pointers

Avoid overusing pointers in code. Consider the following data structure:

```
struct node
{
    node        *previous;
    node        *next;
    uint32_t    value;
};
```

When this structure is compiled for the 32-bit runtime, only 4 bytes out of 12 are used as payload—the rest is used for linking. If you compile that same structure for the 64-bit runtime, the structure takes 20 bytes—the links alone make up 80% of the memory used. Consider using arrays or aggregate types and storing indices instead.

Memory Allocations Are Padded to Preserve Alignment

When you call the `malloc` function directly (or when it is called indirectly, such as when an Objective-C object is allocated), additional memory may be allocated by the operating system to maintain a specific data alignment. When allocating memory for C structs, it may be more efficient for you to allocate a few large blocks of memory instead of allocating memory for each individual struct.

Cache Only When You Need To

Caching previously calculated results is a common way to improve an app's performance. Still, you may want to investigate whether caching is really helping your app. As the previous examples have shown, memory usage is higher on 64-bit systems. If your app relies too much on caching, the pressure it puts on the virtual memory system may actually result in worse performance.

Typical examples of behaviors to avoid include:

- Caching any data that a class can cheaply recompute on the fly
- Caching data or objects that you can easily obtain from another object
- Caching system objects that are inexpensive to re-create
- Caching read-only data that can be accessed via `mmap()`

Always test to ensure that caching improves the performance of your app. And be sure to build hooks into your app so that you can selectively disable caching. In that way, you can test whether disabling a particular cache has a significant effect on the memory usage or the performance of your app. Make sure you test many different data sets for your caching algorithms.

Document Revision History

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

Date	Notes
2013-09-05	New document that explains how to update your Cocoa Touch projects to support the 64-bit runtime.

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