Binary Data Programming Guide for Core Foundation

Developer

Contents

Listings

Introduction to Binary Data Programming Guide for Core Foundation

Important: This is a preliminary document for an API or technology in development. Apple is supplying this information to help you plan for the adoption of the technologies and programming interfaces described herein for use on Apple-branded products. 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 betas of the API or technology.

Binary data can be wrapped inside of Foundation and Core Foundation data objects which provides object-oriented behaviors for manipulating the data. Because data objects are bridged objects, you can use the Foundation and Core Foundation data objects interchangeably. Data objects can manage the allocation and deallocation of byte buffers automatically. Among other things, data objects can be stored in collections, written to property lists, saved to files, and transmitted over communication ports.

Organization of This Document

The following article explains how data objects work:

- Data Objects (page 5) describes how data objects are used as wrappers for byte buffers.

The following articles cover common tasks:

- Working With Binary Data (page 6) explains how to create and use binary data objects.
- Working With Mutable Binary Data (page 8) explains how to modify the bytes in mutable binary data objects.

Data Objects

Data objects are object-oriented wrappers for byte buffers. In these data objects, simple allocated buffers (that is, data with no embedded pointers) take on the behavior of other objects—that is, they encapsulate data and provide operations to manipulate that data. Data objects are typically used to store data. They are also useful in internet and intranet applications because the data contained in data objects can be copied or moved between applications.

Important: The Cocoa Foundation classes, `NSData` and `NSMutableData`, are "toll-free bridged" with their Core Foundation counterparts, `CFData` (see *CFData Reference*) and `CFMutableData` (see *CFMutableData Reference*). This means that the Core Foundation opaque type is interchangeable in function or method calls with the bridged Foundation object. In other words, in an API having an `NSData *` parameter, you can pass in a `CFDataRef`, and in an API having a `CFDataRef` parameter, you can pass in an `NSData` instance. You cannot, however, pass an `NSData` object to an API that expects a mutable `CFData` reference; you must use an `NSMutableData` object instead. This document refers to these objects as simply *data objects* or *mutable data objects* for objects that can be changed after creation.

The size of the data that an instance of `NSData` or `NSMutableData` can wrap is subject to platform-dependent limitations—see *NSData Class Reference*. When the data size is more than a few memory pages, the object uses virtual memory management. A data object can also wrap preexisting data, regardless of how the data was allocated. The object contains no information about the data itself (such as its type); the responsibility for deciding how to use the data lies with the client. In particular, it will not handle byte-order swapping when distributed between big-endian and little-endian machines. Instead, use `NSValue` for typed data.

Data objects provide an operating system–independent way to benefit from copy-on-write memory. The copy-on-write technique means that when data is copied through a virtual memory copy, an actual copy of the data is not made until there is an attempt to modify it.

Typically, you specify the bytes and the length of the bytes stored in a data object when creating that object. You can also extract bytes of a given range from a data object, compare data stored in two data objects, and write data to a URL. You use mutable data objects when you need to modify the data after creation. You can truncate, extend the length of, append data to, and replace a range of bytes in a mutable data object.

Working With Binary Data

This article contains code examples of common tasks that apply to immutable and mutable data objects, `CFData` and `CFMutableData` objects.

Creating Data Objects From Raw Bytes

Generally, you create a data object from raw bytes using the `CFDataCreate` and `CFDataCreateMutable` functions for immutable and mutable data objects respectively. These functions make a copy of the bytes you pass as an argument. The copied bytes are owned by the data object and are freed when the data object is released. It is your responsibility to free the original bytes.

In contrast, the bytes are not copied when you create a data object using `CFDataCreateWithBytesNoCopy` with a deallocator argument which is not `kCFAllocatorNull`. Instead, the data object takes ownership of the bytes passed in as an argument and frees them when the object is released. For this reason, the bytes you pass to this function must have been allocated using the allocator you provide as the deallocator argument.

If you prefer that the bytes not be copied or freed when the object is released, you can create a no-copy, no-free `CFData` object using the `CFDataCreateWithBytesNoCopy` function and passing `kCFAllocatorNull` as the deallocator argument, as in:

```
CFDataRef dictData = CFDataCreateWithBytesNoCopy(
                            NULL, bytes, length, kCFAllocatorNull);
```

Accessing and Comparing Bytes

The three basic `CFData` functions are `CFDataGetBytesPtr`, `CFDataGetBytes`, and `CFDataGetLength`. The `CFDataGetBytesPtr` function returns a pointer to the bytes contained in the data object. The `CFDataGetBytes` function puts the bytes in a supplied buffer. The `CFDataGetLength` function returns the number of bytes contained in the data object.

For example, the following code fragment initializes a data object, `myData`, with the string `myString`. It then uses `CFDataGetBytesPtr` to return the bytes as a pointer.

```
const UInt8 *myString = "Test string.";
CFDataRef myData =
        CFDataCreateWithBytesNoCopy(NULL, myString, strlen(myString),
kCFAllocatorNull);
const UInt8 *ptr = CFDataGetBytePtr(myData);
```

To create a data object that contains a subset of the bytes in another data object, pass a value for the range when calling the `CFDataGetBytes` function. For example, the following code fragment initializes a data object, `data2`, to contain a subrange of `data1`:

```
unsigned char aBuffer[20];
const UInt8 *strPtr = "ABCDEFG";
CFDataRef data1 =
       CFDataCreateWithBytesNoCopy(NULL, strPtr, strlen(strPtr), kCFAllocatorNull);
CFDataGetBytes(data1, CFRangeMake(2, 4), aBuffer);
CFDataRef data2 = CFDataCreate(NULL, aBuffer, 20);
```

To determine whether two data objects are equal, use the `CFEqual` function, which performs a byte-for-byte comparison.

Copying Data Objects

Data objects make it convenient to convert between efficient, read-only data objects and mutable data objects. You use the `CFDataCreateCopy` and `CFDataCreateMutableCopy` functions to get an immutable and mutable copy of an existing data object respectively.

Working With Mutable Binary Data

This article contains code examples of common tasks that apply specifically to mutable data objects, `CFMutableData` objects.

Modifying Bytes

The three basic `CFMutableData` functions are `CFDataGetMutableBytePtr`, `CFDataGetLength` and `CFDataSetLength`. The `CFDataGetMutableBytePtr` function returns a pointer for writing into the bytes contained in the mutable data object. The `CFDataGetLength` function returns the number of bytes contained in the data object. The `CFDataSetLength` function allows you to truncate or extend the length of a mutable data object. The `CFDataIncreaseLength` function also allows you to change the length of a mutable data object.

In Listing 1 (page 8), `CFDataGetMutableBytePtr` is used to return a pointer to the bytes in `data2`. The bytes in `data2` are then overwritten with the contents of `data1` using the `CFDataGetBytes` function.

Listing 1 Modifying bytes

```
 // Create some strings.
const UInt8 *myString = "string for data1";
const UInt8 *yourString = "string for data2";

// Declare buffers used later.
const UInt8 *data1Bytes, *data2Bytes;

// Create mutable data objects, data1 and data2.
CFMutableDataRef data1 = CFDataCreateMutable(NULL, 0);
CFMutableDataRef data2 = CFDataCreateMutable(NULL, 0);
CFDataAppendBytes(data1, myString, strlen(myString));
CFDataAppendBytes(data2, yourString, strlen(yourString));

// Get and print the data1 bytes.
```

```
data1Bytes = CFDataGetBytePtr(data1);
fprintf(stdout, "data1 before: \"%s\"\n", data1Bytes);

// Get and print the data2 bytes.
data2Bytes = CFDataGetMutableBytePtr(data2);
fprintf(stdout, "data2 before: \"%s\"\n", data2Bytes);

// Copy the bytes from data1 into the mutable bytes from data2.
CFDataGetBytes(data1,
               CFRangeMake(0, CFDataGetLength(data1)),
               data2Bytes);

// Get and print the data2 bytes again.
fprintf(stdout, "data2 after: \"%s\"\n", CFDataGetBytePtr(data2));
```

This is the output from Listing 1 (page 8):

```
data1 before: "string for data1"
data2 before: "string for data2"
data2 after: "string for data1"
```

Appending Bytes

The `CFDataAppendBytes` function lets you append bytes of the specified length to a mutable data object. For example, Listing 2 (page 9) copies the bytes in `data2` into `aBuffer` and then appends `aBuffer` to `data1`:

Listing 2 Appending bytes

```
 // Create mutable data objects, data1 and data2.
CFMutableDataRef data1 = CFDataCreateMutable(NULL, 0);
CFDataAppendBytes(data1, "ABCD", strlen("ABCD"));
CFMutableDataRef data2 = CFDataCreateMutable(NULL, 0);
CFDataAppendBytes(data2, "EFGH", strlen("EFGH"));
```

```
// Get the data2 bytes.
const UInt8 *aBuffer = CFDataGetBytePtr(data2);

// Append the bytes from data2 to data1.
CFDataAppendBytes(data1, aBuffer, CFDataGetLength(data2));
```

Replacing Bytes

You can delete a range of bytes in a mutable data object (using the `CFDataDeleteBytes` function) or replace a range of bytes with other bytes (using the `CFDataReplaceBytes` function).

In Listing 3 (page 10), a range of bytes in `data1` is replaced by the bytes in `data2` using `CFDataReplaceBytes` (the content of `data1` changes from "Liz and John" to "Liz and Larry"):

Listing 3 Replacing bytes

```
// Create mutable data objects, data1 and data2.
CFMutableDataRef data1 = CFDataCreateMutable(NULL, 0);
CFDataAppendBytes(data1, "Liz and John", strlen("Liz and John"));
CFMutableDataRef data2 = CFDataCreateMutable(NULL, 0);
CFDataAppendBytes(data2, "Larry", strlen("Larry"));

// Allocate a buffer the length of data2.
unsigned len = CFDataGetLength(data2);
unsigned char *aBuffer = malloc(len);

// Put the data2 bytes into the buffer.
CFDataGetBytes(data2, CFRangeMake(0, len), aBuffer);

// Replace John with Larry.
CFDataReplaceBytes(data1, CFRangeMake(8, CFDataGetLength(data1) - 8), aBuffer,
len);
```

Document Revision History

This table describes the changes to *Binary Data Programming Guide for Core Foundation*.

Date	Notes
2006-01-10	Changed title from "Binary Data."
2003-08-07	First release of conceptual and task material covering the support for binary data objects in Core Foundation.
2003-01-17	Converted existing Core Foundation documentation into topic format. Added revision history.

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