Memory Management Programming Guide for Core Foundation

Performance

2009-10-21

Contents

Figures and Listings

Introduction

For managing memory Core Foundation uses allocators, a reference-counting mechanism, and a policy of object ownership that is suggested by the names of functions. This topic covers related techniques for creating, copying, retaining, and releasing objects.

Memory management is fundamental to using Core Foundation effectively and efficiently. This document is essential reading for all developers who use Core Foundation.

Organization of This Document

The following concepts and tasks discuss the built in support Core Foundation provides for managing the memory allocation and deallocation of objects:

- "Ownership Policy" (page 11)
- "Core Foundation Object Lifecycle Management" (page 17)
- "Copy Functions" (page 19)
- "Allocators" (page 9)

If you need to customize your allocators then read:

- "Using Allocators in Creation Functions" (page 23)
- "Using the Allocator Context" (page 25)
- "Creating Custom Allocators" (page 27)

To find out more about byte ordering and swapping see:

- "Byte Ordering" (page 21)
- "Byte Swapping" (page 29)

See Also

The following may also be of interest:

- *Core Foundation Design Concepts*
- *Memory Management Programming Guide*

Allocators

Among the operating-system services that Core Foundation abstracts is memory allocation. It uses allocators for this purpose.

Allocators are opaque objects that allocate and deallocate memory for you. You never have to allocate, reallocate, or deallocate memory directly for Core Foundation objects—and rarely should you. You pass allocators into functions that create objects; these functions have “Create” embedded in their names, for example, `CFStringCreateWithPascalString`. The creation functions use the allocators to allocate memory for the objects they create.

The allocator is associated with the object through its life span. If reallocation of memory is necessary, the object uses the allocator for that purpose and when the object needs to be deallocated, the allocator is used for the object’s deallocation. The allocator is also used to create any objects required by the originally created object. Some functions also let you pass in allocators for special purposes, such as deallocating the memory of temporary buffers.

Core Foundation allows you to create your own custom allocators. Core Foundation also provides a system allocator and initially sets this allocator to be the default one for the current thread. (There is one default allocator per thread.) You can set a custom allocator to be the default for a thread at any time in your code. However, the system allocator is a good general-purpose allocator that should be sufficient for almost all circumstances. Custom allocators might be necessary in special cases, such as in certain situations on Mac OS 9 or as bulk allocators when performance is an issue. Except for these rare occasions, you should neither use custom allocators or set them as the default, especially for libraries.

For more on allocators and, specifically, information on creating custom allocators, see “Creating Custom Allocators” (page 27).

Ownership Policy

Applications using Core Foundation constantly access and create and dispose of objects. In order to ensure that you do not leak memory, Core Foundation defines rules for getting and creating objects.

Fundamentals

When trying to understand memory management in a Core Foundation application, it is helpful to think not in terms of memory management per se, but instead in terms of object ownership. An object may have one or more owners; it records the number of owners it has using a retain count. If an object has no owners (if its retain count drops to zero), it is disposed of (freed). Core Foundation defines the following rules for object ownership and disposal.

- If you create an object (either directly or by making a copy of another object—see "The Create Rule" (page 12)), you own it.
- If you get an object from somewhere else, you do not own it. If you want to prevent it being disposed of, you must add yourself as an owner (using `CFRetain`).
- If you are an owner of an object, you must relinquish ownership when you have finished using it (using `CFRelease`).

Naming Conventions

There are many ways in which you can get a reference to an object using Core Foundation. In line with the Core Foundation ownership policy, you need to know whether or not you own an object returned by a function so that you know what action to take with respect to memory management. Core Foundation has established a naming convention for its functions that allows you to determine whether or not you own an object returned by a function. In brief, if a function name contains the word "Create" or "Copy", you own the object. If a function name contains the word "Get", you do not own the object. These rules are explained in greater detail in "The Create Rule" (page 12) and "The Get Rule" (page 13).

Important: Cocoa defines a similar set of naming conventions for memory management (see *Memory Management Programming Guide*). The Core Foundation naming conventions, in particular use of the word "create", only apply to C functions that return Core Foundation objects. Naming conventions for Objective-C methods are governed by the Cocoa conventions, irrespective of whether the method returns a Core Foundation or Cocoa object.

The Create Rule

Core Foundation functions have names that indicate when you own a returned object:

- Object-creation functions that have "Create" embedded in the name;
- Object-duplication functions that have "Copy" embedded in the name.

If you own an object, it is your responsibility to relinquish ownership (using `CFRelease`) when you have finished with it.

Consider the following examples. The first example shows two create functions associated with CFTimeZone and one associated with CFBundle.

```
CFTimeZoneRef   CFTimeZoneCreateWithTimeIntervalFromGMT (CFAllocatorRef allocator,
 CFTimeInterval ti);
CFDictionaryRef CFTimeZoneCopyAbbreviationDictionary (void);
CFBundleRef     CFBundleCreate (CFAllocatorRef allocator, CFURLRef bundleURL);
```

The first function contains the word "Create" in its name, and it creates a new CFTimeZone object. You own this object, and it is your responsibility to relinquish ownership. The second function contains the word "Copy" in its name, and creates a copy of an attribute of a time zone object. (Note that this is different from getting the attribute itself—see "The Get Rule" (page 13).) Again, you own this object, and it is your responsibility to relinquish ownership. The third function, `CFBundleCreate`, contains the word "Create" in its name, but the documentation states that it may return an existing CFBundle. Again, though, you own this object whether or not a new one is actually created. If an existing object is returned, its retain count is incremented so it is your responsibility to relinquish ownership.

The next example may appear to be more complex, but it still follows the same simple rule.

```
/* from CFBag.h */
CF_EXPORT CFBagRef  CFBagCreate(CFAllocatorRef allocator, const void **values,
 CFIndex numValues, const CFBagCallBacks *callBacks);
CF_EXPORT CFMutableBagRef   CFBagCreateMutableCopy(CFAllocatorRef allocator,
CFIndex capacity, CFBagRef bag);
```

The CFBag function `CFBagCreateMutableCopy` has both "Create" and "Copy" in its name. It is a creation function because the function name contains the word "Create". Note also that the first argument is of type `CFAllocatorRef`—this serves as a further hint. The "Copy" in this function is a hint that the function takes a `CFBagRef` argument and produces a duplicate of the object. It also refers to what happens to the element objects of the source collection: they are copied to the newly created bag. The secondary "Copy" and "NoCopy" substrings of function names indicate how objects owned by some source objects are treated—that is, whether they are copied or not.

The Get Rule

If you receive an object from any Core Foundation function other than a creation or copy function—such as a Get function—you do not own it and cannot be certain of the object's life span. If you want to ensure that such an object is not disposed of while you are using it, you must claim ownership (with the `CFRetain` function). You are then responsible for relinquishing ownership when you have finished with it.

Consider the `CFAttributedStringGetString` function, which returns the backing string for an attributed string.

```
CFStringRef CFAttributedStringGetString (CFAttributedStringRef aStr);
```

If the attributed string is *freed*, it relinquishes ownership of the backing string. If the attributed string was the backing string's only owner, then the backing string now has no owners and it is itself freed. If you need to access the backing string after the attributed string has been disposed of, you must claim ownership (using `CFRetain`)—or make a copy of it. You must then relinquish ownership (using `CFRelease`) when you have finished with it, otherwise you create a memory leak.

Instance Variables and Passing Parameters

A corollary of the basic rules is that when you pass an object to another object (as a function parameter), you should expect that the receiver will take ownership of the passed object if it needs to maintain it.

To understand this, put yourself in the position of the implementer of the receiving object. When a function receives an object as a parameter, the receiver does not initially own the object. The object may therefore be deallocated at any time—unless the receiver takes ownership of if (using `CFRetain`). When the receiver has finished with the object—either because it is replaced a new value or because the receiver is itself being deallocated—the receiver is responsible for relinquishing ownership (using `CFRelease`).

Ownership Examples

To prevent runtime errors and memory leaks, you should ensure that you consistently apply the Core Foundation ownership policy wherever Core Foundation objects are received, passed, or returned. To understand why it may be necessary to become an owner of an object you did not create, consider this example. Suppose you get a value from another object. If the value's "containing" object is subsequently deallocated, it relinquishes ownership of the "contained" object. If the containing object was the only owner of the value, then the value has no owners and it is deallocated too. You now have a reference to a freed object, and if you try to use it your application will crash.

The following fragments of code illustrate three common situations: a Set accessor function, a Get accessor function, and a function that holds onto a Core Foundation object until a certain condition is met. First the Set function:

```
static CFStringRef title = NULL:
void SetTitle(CFStringRef newTitle) {
    CFStringRef temp = title;
    title = CFStringCreateCopy(kCFAllocatorDefault , newTitle);
    CFRelease(temp);
}
```

The above example uses a static `CFStringRef` variable to hold the retained CFString object. You could use other means for storing it but you must put it, of course, in some place that isn't local to the receiving function. The function assigns the current title to a local variable before it copies the new title and releases the old title. It releases after copying in case the CFString object passed in is the same object as the one currently held.

Notice that in the above example the object is copied rather than simply retained. (Recall that from an ownership perspective these are equivalent—see "Fundamentals" (page 11).) The reason for this is that the title property might be considered an attribute. It is something that should not be changed except through accessor methods. Even though the parameter is typed as `CFStringRef`, a reference to a CFMutableString object might be passed in, which would allow for the possibility of the value being changed externally. Therefore you copy the object so that it won't be changed while you hold it. You should copy an object if the object is or could be mutable and you need your own unique version of it. If the object is considered a relationship, then you should retain it.

The corresponding Get function is much simpler:

```
CFStringRef GetTitle() {
    return title;
}
```

By simply returning an object you are returning a weak reference to it. In other words, the pointer value is copied to the receiver's variable but the reference count is unchanged. The same thing happens when an element from a collection is returned.

The following function retains an object retrieved from a collection until it no longer needs it, then releases it. The object is assumed to be immutable.

```
static CFStringRef title = NULL:
void MyFunction(CFDictionary dict, Boolean aFlag) {
    if (!title && !aFlag) {
        title = (CFStringRef)CFDictionaryGetValue(dict, CFSTR("title"));
        title = CFRetain(title);
    }
    /* Do something with title here. */
    if (aFlag) {
        CFRelease(title);
    }
}
```

The following example shows passing a number object to an array. The array's callbacks specify that objects added to the collection are retained (the collection owns them), so the number can be released after it's added to the array.

```
float myFloat = 10.523987;
CFNumberRef myNumber = CFNumberCreate(kCFAllocatorDefault,
                                    kCFNumberFloatType, &myFloat);
CFMutableArrayRef myArray = CFArrayCreateMutable(kCFAllocatorDefault, 2,
&kCFTypeArrayCallBacks);
CFArrayAppendValue(myArray, myNumber);
CFRelease(myNumber);
// code continues...
```

Note that there is a potential pitfall here if (a) you release the array, and (b) you continue to use the number variable after releasing the array:

```
CFRelease(myArray);
CFNumberRef otherNumber = // ... ;
CFComparisonResult comparison = CFNumberCompare(myNumber, otherNumber, NULL);
```

Unless you retained the number or the array, or passed either to some other object which maintains ownership of it, the code will fail in the comparison function. If no other object owns the array or the number, when the array is released it is also deallocated, and so it releases its contents. In this situation, this will also result in the deallocation of the number, so the comparison function will operate on a freed object and thus crash.

Core Foundation Object Lifecycle Management

The life span of a Core Foundation object is determined by its reference count—an internal count of the number of clients who want the object to persist. When you create or copy an object in Core Foundation, its reference count is set to one. Subsequent clients can claim ownership of the object by calling `CFRetain` which increments the reference count. Later, when you have no more use for the object, you call `CFRelease`. When the reference count reaches 0, the object's allocator deallocates the object's memory.

Retaining Object References

To increment the reference count of a Core Foundation object, pass a reference to that object as the parameter of the `CFRetain` function:

```
/* myString is a CFStringRef received from elsewhere */
myString = (CFStringRef)CFRetain(myString);
```

Releasing Object References

To decrement the reference count of a Core Foundation object, pass a reference to that object as the parameter of the `CFRelease` function:

```
CFRelease(myString);
```

Important: You should never directly deallocate a Core Foundation object (for example, by calling `free` on it). When you are finished with an object, call the `CFRelease` function and Core Foundation will properly dispose of it.

Copying Object References

When you copy an object, the resulting object has a reference count of one regardless of the reference count of the original object. For more on copying objects, see "Copy Functions" (page 19).

Determining an Object's Retain Count

If you want to know the current reference count of a Core Foundation object, pass a reference to that object as the parameter of the `CFGetRetainCount` function:

```
CFIndex count = CFGetRetainCount(myString);
```

Note, however, that there should typically be little need to determine the reference count of a Core Foundation object, except in debugging. If you find yourself needing to know the retain count of an object, check that you are properly adhering to the ownership policy rules (see "Ownership Policy" (page 11)).

Copy Functions

In general, a standard copy operation, which might also be called simple assignment, occurs when you use the = operator to assign the value of one variable to another. The expression `myInt2 = myInt1`, for example, causes the integer contents of `myInt1` to be copied from the memory used by `myInt1` into the memory used by `myInt2`. Following the copy operation, two separate areas of memory contain the same value. However, if you attempt to copy a Core Foundation object in this way, be aware that you will not duplicate the object itself, only the *reference* to the object.

For example, someone new to Core Foundation might think that to make a copy of a CFString object she would use the expression `myCFString2 = myCFString1`. Again, this expression does not actually copy the string data. Because both `myCFString1` and `myCFString2` must have the CFStringRef type, this expression only copies the reference to the object. Following the copy operation, you have two copies of the reference to the CFString. This type of copy is very fast because only the reference is duplicated, but it is important to remember that copying a mutable object in this way is dangerous. As with programs that use global variables, if one part of your application changes an object using a copy of the reference, there is no way for other parts of the program which have copies of that reference to know that the data has changed.

If you want to duplicate an object, you must use one of the functions provided by Core Foundation specifically for this purpose. Continuing with the CFString example, you would use `CFStringCreateCopy` to create an entirely new CFString object containing the same data as the original. Core Foundation types which have "CreateCopy" functions also provide the variant "CreateMutableCopy" which returns a copy of an object that can be modified.

Shallow Copy

Copying **compound objects**, objects such as collection objects that can contain other objects, must also be done with care. As you would expect, using the = operator to perform a copy on these objects results in a duplication of the object reference. In contrast to simple objects like CFString and CFData, the "CreateCopy" functions provided for compound objects such as CFArray and CFSet actually perform a **shallow copy**. In the case of these objects, a shallow copy means that a new collection object is created, but the contents of the original collection are not duplicated—only the object references are copied to the new container. This type of copy is useful if, for example, you have an array that's immutable and you want to reorder it. In this case, you don't want to duplicate all of the contained objects because there's no need to change them—and why use up that extra memory? You just want the set of included objects to be changed. The same risks apply here as with copying object references with simple types.

Deep Copy

When you want to create an entirely new compound object, you must perform a **deep copy**. A deep copy duplicates the compound object as well as the contents of all of its contained objects. The current release of Core Foundation includes a function that performs deep copying of a property list (see

`CFPropertyListCreateDeepCopy`). If you want to create deep copies of other structures, you could perform the deep copy yourself by recursively descending into the compound object and copying all of its contents one by one. Take care in implementing this functionality as compound objects can be recursive—they may directly or indirectly contain a reference to themselves—which can cause a recursive loop.

Byte Ordering

Microprocessor architectures commonly use two different methods to store the individual bytes of multibyte numerical data in memory. This difference is referred to as "byte ordering" or "endian nature." Most of the time the endian format of your computer can be safely ignored, but in certain circumstances it becomes critically important. Mac OS X provides a variety of functions to turn data of one endianness into another.

Intel x86 processors store a two-byte integer with the least significant byte first, followed by the most significant byte. This is called little-endian byte ordering. Other CPUs, such as the PowerPC CPU, store a two-byte integer with its most significant byte first, followed by its least significant byte. This is called big-endian byte ordering. Most of the time the endian format of your computer can be safely ignored, but in certain circumstances it becomes critically important. For example, if you try to read data from files that were created on a computer that is of a different endian nature than yours, the difference in byte ordering can produce incorrect results. The same problem can occur when reading data from a network.

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

To give a concrete example around which to discuss endian format issues, consider the case of a simple C structure which defines two four byte integers as shown in Listing 1 (page 21).

Listing 1 Example data structure

```
struct {
    UInt32 int1;
    UInt32  int2;
} aStruct;
```

Suppose that the code shown in Listing 2 (page 21) is used to initialize the structure shown in Listing 1 (page 21).

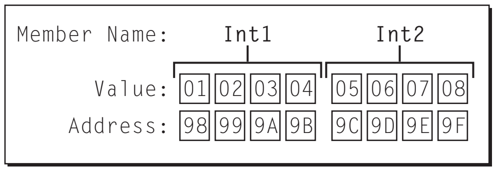
Listing 2 Initializing the example structure

```
ExampleStruct   aStruct;

aStruct.int1 = 0x01020304;
aStruct.int2 = 0x05060708;
```

Consider the diagram in Figure 1 (page 22), which shows how a big-endian processor or memory system would organize the example data. In a big-endian system, physical memory is organized with the address of each byte increasing from most significant to least significant.

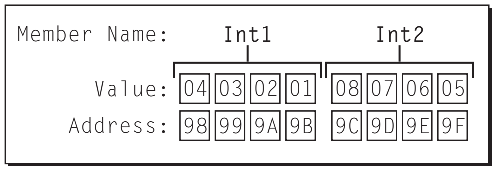
Figure 1 Example data in big-endian format



Notice that the fields are stored with the more significant bytes to the left and less significant bytes to the right. This means that the address of the most significant byte of the address field `Int1` is `0x98`, while the address `0x9B` corresponds to the least significant byte of `Int1`.

The diagram in Figure 2 (page 22) shows how a little-endian system would organize the data.

Figure 2 Example data in little-endian format



Notice that the lowest address of each field now corresponds to the least significant byte instead of the most significant byte. If you were to print out the value of `Int1` on a little-endian system you would see that despite being stored in a different byte order, it is still interpreted correctly as the decimal value `16909060`.

Now suppose the example data values initialized by the code shown in Listing 2 (page 21) are generated on a little-endian system and saved to disk. Assume that the data is written to disk in byte-address order. When read from disk by a big-endian system, the data would again be laid out in memory as illustrated in Figure 2 (page 22). The problem is that the data is still in little-endian byte order even though it is being interpreted on a big-endian system. This difference causes the values to be evaluated incorrectly. In this example, the decimal value of the field `Int1` should be `16909060`, but because of the incorrect byte ordering it is evaluated as `67305985`. This phenomenon is called byte swapping and occurs when data in one endian format is read by a system that uses the other endian format.

Unfortunately, this is a problem that can't be solved in the general case. The reason is that the manner in which you swap depends on the format of your data. Character strings typically don't get swapped at all, longwords get swapped four bytes end-for-end, words get swapped two bytes end-for-end. Any program that needs to swap data around therefore has to know the data type, the source data endian order, and the host endian order.

The functions in `CFByteOrder.h` allow you to perform byte swapping on two-byte and four-byte integers as well as floating point values. Appropriate use of these functions help you ensure that the data your program manipulates is in the correct endian order. See section "Byte Swapping" (page 29) for details on using these functions. Note that Core Foundation's byte swapping functions are available on Mac OS X only.

Using Allocators in Creation Functions

Each Core Foundation opaque type has one or more **creation functions**, functions that create and return an object of that type initialized in a particular way. All creation functions take as their first parameter a reference to an allocator object (`CFAllocatorRef`). Some functions may also have allocator parameters for specialized allocation and deallocation purposes.

You have several options for the allocator-reference parameters:

- You can pass the constant `kCFAllocatorSystemDefault`; this specifies the generic system allocator (which is the initial default allocator).
- You can pass `NULL` to specify the current default allocator (which might be a custom allocator or the generic system allocator). This is the same as passing `kCFAllocatorDefault`.
- You can pass the constant `kCFAllocatorNull` which indicates an allocator that does not allocate—it is an error to attempt to use it. Some creation functions have a parameter for a special allocator used to reallocate or free a backing store; by specifying `kCFAllocatorNull` for the parameter, you prevent automatic reallocation or deallocation.
- You can get a reference to the allocator used by another Core Foundation object with the `CFGetAllocator` function and pass that reference in. This technique allows you to put related objects in a memory "zone" by using the same allocator for allocating them.
- You can pass a reference to a custom allocator (see "Creating Custom Allocators" (page 27)).

If you are to use a custom allocator and you want to make it the default allocator, it is advisable to first get a reference to the current default allocator using the `CFAllocatorGetDefault` function and store that in a local variable. When you are finished using your custom allocator, use the `CFAllocatorSetDefault` function to reset the stored allocator as the default allocator.

Using the Allocator Context

Every allocator in Core Foundation has a **context**. A context is a structure that defines the operating environment for an object and typically consists of function pointers. The context for allocators is defined by the `CFAllocatorContext` structure. In addition to function pointers, the structure contains fields for a version number and for user-defined data

Listing 1 The CFAllocatorContext structure

```
typedef struct {
    CFIndex version;
    void * info;
    const void *(*retain)(const void *info);
    void (*release)(const void *info);
    CFStringRef (*copyDescription)(const void *info);
    void * (*allocate)(CFIndex size, CFOptionFlags hint, void *info);
    void * (*reallocate)(void *ptr, CFIndex newsize, CFOptionFlags hint, void
*info);
    void (*deallocate)(void *ptr, void *info);
    CFIndex (*preferredSize)(CFIndex size, CFOptionFlags hint, void *info);
} CFAllocatorContext;
```

The `info` field contains any specially defined data for the allocator. For example, an allocator could use the `info` field to track outstanding allocations.

Important: For the current release, do not set the value of the `version` field to anything other than 0.

If you have some user-defined data in the allocator context (the `info` field), use the `CFAllocatorGetContext` function to obtain the `CFAllocatorContext` structure for an allocator. Then evaluate or handle the data as needed. The following code provides an example of this:

Listing 2 Getting the allocator context and user-defined data

```
static int numOutstandingAllocations(CFAllocatorRef alloc) {
    CFAllocatorContext context;
    context.version = 0;
    CFAllocatorGetContext(alloc, &context);
    return (*(int *)(context.info));
}
```

Other Core Foundation functions invoke the memory-related callbacks defined in an allocator context and take or return an untyped pointer to a block of memory (`void *`):

- `CFAllocatorAllocate`, allocates a block of memory.
- `CFAllocatorReallocate` reallocates a block of memory.
- `CFAllocatorDeallocate` deallocates a block of memory.

- `CFAllocatorGetPreferredSizeForSize` gives the size of memory likely to be allocated, given a certain request.

Creating Custom Allocators

To create a custom allocator, first declare and initialize a structure of type `CFAllocatorContext`. Initialize the version field to 0 and allocate and assign any desired data, such as control information, to the `info` field. The other fields of this structure are function pointers described in "Implementing Allocator Callbacks" (page 27), below.

Once you have assigned the proper values to the fields of the `CFAllocatorContext` structure, invoke the `CFAllocatorCreate` function to create the allocator object. The second parameter of this function is a pointer to the structure. The first parameter of this function identifies an allocator to use for allocating memory for the new object. If you want to use the `allocate` callback in the `CFAllocateContext` structure for this purpose, specify the `kCFAllocatorUseContext` constant for the first parameter. If you want to use the default allocator, specify `NULL` in this parameter.

Listing 1 Creating a custom allocator

```
static CFAllocatorRef myAllocator(void) {
    static CFAllocatorRef allocator = NULL;
    if (!allocator) {
        CFAllocatorContext context =
            {0, NULL, NULL, (void *)free, NULL,
             myAlloc, myRealloc, myDealloc, NULL};
        context.info = malloc(sizeof(int));
        allocator = CFAllocatorCreate(NULL, &context);
    }
    return allocator;
}
```

Implementing Allocator Callbacks

The `CFAllocatorContext` structure has seven fields defining callback functions. If you create a custom allocator you must implement at least the `allocate` function. Allocator callbacks should be thread-safe and, if the callback invokes other functions, they should be re-entrant as well.

The retain, release, and copy-description callbacks all take as their single argument the `info` field of the `CFAllocatorContext` structure. Typed as `void *`, this field points to any data that you define for the allocator, such as a structure containing control information.

Retain Callback:

```
const void *(*retain)(const void *info);
```

Retain the data you have defined for the allocator context in `info`. This might make sense only if the data is a Core Foundation object. You may set this function pointer to `NULL`.

Release Callback:

```
void (*release)(const void *info);
```

Release (or free) the data you have defined for the allocator context. You may set this function pointer to `NULL`, but doing so might result in memory leaks.

Copy Description Callback:

```
CFStringRef (*copyDescription)(const void *info);
```

Return a reference to a CFString that describes your allocator, particularly some characteristics of your user-defined data. You may set this function pointer to `NULL`, in which case Core Foundation will provide a rudimentary description.

Allocate Callback:

```
void *   (*allocate)(CFIndex size, CFOptionFlags hint, void *info);
```

Allocate a block of memory of at least `size` bytes and return a pointer to the start of the block. The `hint` argument is a bitfield that you should currently not use. The `size` parameter should always be greater than 0. If it is not, or if problems in allocation occur, return `NULL`. This callback may not be `NULL`.

Reallocate Callback:

```
void *   (*reallocate)(void *ptr, CFIndex newsize, CFOptionFlags hint, void
*info);
```

Change the size of the block of memory pointed to by `ptr` to the size specified by `newsize` and return the pointer to the larger block of memory. Return `NULL` on any reallocation failure, leaving the old block of memory untouched. Note that the *`ptr`* parameter will never be `NULL`, and *`newsize`* will always be greater than `0`—this callback is not used unless those two conditions are met.

Leave the contents of the old block of memory unchanged up to the lesser of the new or old sizes. If the `ptr` parameter is not a block of memory that has been previously allocated by the allocator, the results are undefined; abnormal program termination can occur. The `hint` argument is a bitfield that you should currently not use. If you set this callback to `NULL` the `CFAllocatorReallocate` function returns `NULL` in most cases when it attempts to use this allocator.

Deallocate Callback:

```
void   (*deallocate)(void *ptr, void *info);
```

Make the block of memory pointed to by *`ptr`* available for subsequent reuse by the allocator but unavailable for continued use by the program. The `ptr` parameter cannot be `NULL` and if the `ptr` parameter is not a block of memory that has been previously allocated by the allocator, the results are undefined; abnormal program termination can occur. You can set this callback to `NULL`, in which case the `CFAllocatorDeallocate` function has no effect.

Preferred Size Callback:

```
CFIndex   (*preferredSize)(CFIndex size, CFOptionFlags hint, void *info);
```

Return the actual size the allocator is likely to allocate given a request for a block of memory of size `size`. The `hint` argument is a bitfield that you should currently not use.

Byte Swapping

If you need to find out the host byte order you can use the function `CFByteOrderGetCurrent`. The possible return values are `CFByteOrderUnknown`, `CFByteOrderLittleEndian`, and `CFByteOrderBigEndian`.

Byte Swapping Integers

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

Although the primitive swapping functions swap unconditionally, the higher level swapping functions are defined in such a way that they do nothing when a byte swap is not required—in other words, when the source and host byte orders are the same. For the integer types, these functions take the forms `CFSwapXXXBigToHost` and `CFSwapXXXLittleToHost`, `CFSwapXXXHostToBig`, and `CFSwapXXXHostToLittle` where `XXX` is a data type such as `Int32`. For example, if you were on a little-endian machine reading a 16-bit integer value from a network whose data is in network byte order (big-endian), you would use the function `CFSwapInt16BigToHost`. Listing 1 (page 29) demonstrates this process.

Listing 1 Swapping a 16 bit Integer

```
SInt16  bigEndian16;
SInt16  swapped16;

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

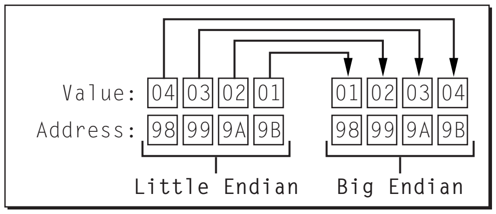
The section "Byte Ordering" (page 21) introduced the example of a simple C structure that was created and saved to disk on a little endian machine and then read from disk on a big-endian machine. In order to correct the situation, you must swap the bytes in each field. The code in Listing 2 (page 29) demonstrates how you would use Core Foundation byte-swapping functions to accomplish this.

Listing 2 Byte swapping fields in a C structure

```
// Byte swap the values if necessary.
aStruct.int1 = CFSwapInt32LittleToHost(aStruct.int1)
aStruct.int2 = CFSwapInt32LittleToHost(aStruct.int2)
```

Assuming a big-endian architecture, the functions used in Listing 2 (page 29) would swap the bytes in each field. Figure 1 (page 30) shows the effect of byte swapping on the field `aStruct.int1`. Note that the byte-swapping code would do nothing when run on a little-endian machine. The compiler should optimize out the code and leave the data untouched.

Figure 1 Four-byte little-endian to big-endian swap



Byte Swapping Floating-Point Values

Even on a single platform there can be many different representations for floating-point values. Unless you are very careful, attempting to pass floating-point values across platform boundaries causes no end of headaches. To help you work with floating-point numbers, Core Foundation defines a set of functions and two special data types in addition to the integer-swapping functions. These functions allow you to encode 32-and 64-bit floating-point values in such a way that they can later be decoded and byte swapped if necessary. Listing 3 (page 30) shows you how to encode a 64-bit floating-point number and Listing 4 (page 30) shows how to decode it.

Listing 3 Encoding a Floating Point Value

```
Float64            myFloat64;
CFSwappedFloat64    swappedFloat;

// Encode the floating-point value.
swappedFloat = CFConvertFloat64HostToSwapped(myFloat64);
```

Listing 4 Decoding a floating-point value

```
Float64            myFloat64;
CFSwappedFloat64    swappedFloat;

// Decode the floating-point value.
myFloat64 = CFConvertFloat64SwappedToHost(swappedFloat);
```

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

Document Revision History

This table describes the changes to *Memory Management Programming Guide for Core Foundation*.

Date	Notes
2009-10-21	Corrected a typographical error.
2009-08-19	Clarified the Create rule with respect to Cocoa's memory management conventions.
2008-10-15	Augmented article describing Ownership Policy.
2007-04-03	Clarified the effect of the Create Rule when an existing object is returned.
2006-01-10	Changed title from "Memory Management."
2005-12-06	Made a minor change to make memory management rules explicit.
2005-08-11	Added reference to CFPropertyListCreateDeepCopy in "Copy Functions." Consolidated two articles into "Object Lifecycle."
2003-01-17	Converted existing Core Foundation documentation into topic format. Added revision history.