Property List Programming Topics for Core Foundation

Core Foundation > Data Management

2006-02-07

Contents

Tables and Listings

Introduction to Property List Programming Topics for Core Foundation

Many applications require a mechanism for storing information that will be needed at a later time. For situations where you need to store small amounts of persistent data, less than a few hundred kilobytes, Core Foundation provides property lists. Property lists—frequently referred to as "plists"—offer a uniform and architecture-independent means of organizing, storing, and accessing data for Mac OS X applications.

Organization of This Document

Property lists organize data into named values and lists of values using several Core Foundation types: CFString, CFNumber, CFBoolean, CFDate, CFData, CFArray, and CFDictionary. These types give you the means to produce data that is meaningfully structured, transportable, storable, and accessible, but still as efficient as possible. The property list programming interface allows you to convert hierarchically structured combinations of these basic types to and from standard XML. The XML data can be saved to disk and later used to reconstruct the original Core Foundation objects. Note that property lists should be used for data that consists primarily of strings and numbers because they are very inefficient when used with large blocks of binary data.

Property lists are used frequently in Mac OS X. For example, the Mac OS X Finder—through bundles—uses property lists to store file and directory attributes. Core Foundation bundles and URL objects use property lists as well. User and application preferences also use property lists, however, you should not use the CFPropertyList API to read and modify preferences. Core Foundation provides a programming interface specifically for this purpose—see Preferences for more information.

This document describes the property list structure, and use of XML tags and specifics about numbers, and contains examples on creating, saving, and restoring property lists.

- "Property List Structure and Contents" (page 9)
- "Creating Property Lists" (page 11)
- "Saving and Restoring Property Lists" (page 13)
- "Using Numbers in Property Lists" (page 17)
- "Property List XML Tags" (page 19)

Property List Structure and Contents

Property lists are constructed from the basic Core Foundation types CFString, CFNumber, CFBoolean, CFDate, and CFData. To build a complex data structure out of these basic types, you put them inside a CFDictionary or CFArray. To simplify programming with property lists, any of the property list types can also be referred to using a reference of type `CFPropertyListRef`.

It is important to understand that CFPropertyList provides an abstraction for all the property list types—you can think of CFPropertyList in object-oriented terms as being the superclass of CFString, CFNumber, CFDictionary, and so on. When a Core Foundation function returns a `CFPropertyListRef`, it means that the value may be any of the property list types. For example, `CFPreferencesCopyAppValue` returns a `CFPropertyListRef`. This means that the value returned can be a CFString object, a CFNumber object, a CFDictionary object, and so on again. You can use `CFGetTypeID` to determine what type of object a property list value is.

In a CFDictionary object, data is structured as key-value pairs, where each key is a string and the key's value can be a CFString, a CFNumber, a CFBoolean, a CFDate, a CFData, a CFArray, or another CFDictionary object. If you use a CFDictionary object as a property list, all its keys *must* be CFString objects.

In a CFArray object, data is structured as an ordered collection of objects that can be accessed by index. In a property list, a CFArray object can contain any of the basic property list types, as well as CFDictionary objects and other CFArray objects.

Note that although CFDictionary and CFArray objects can contain data types other than the property list types, if they do, you can't use the Core Foundation property list programming interface to work with them.

Creating Property Lists

The examples in this section demonstrate how to create and work with property lists. The error checking code has been removed for clarity. In practice, it is *vital* that you check for errors because passing bad parameters into Core Foundation routines can cause your application to crash.

Listing 1 (page 11) shows you how to create a very simple property list—an array of CFString objects.

Listing 1 Creating a simple property list from an array

```
#include <CoreFoundation/CoreFoundation.h>
#define kNumFamilyMembers 5

void main () {
    CFStringRef names[kNumFamilyMembers];
    CFArrayRef  array;
    CFDataRef   xmlData;

    // Define the family members.
    names[0] = CFSTR("Marge");
    names[1] = CFSTR("Homer");
    names[2] = CFSTR("Bart");
    names[3] = CFSTR("Lisa");
    names[4] = CFSTR("Maggie");

    // Create a property list using the string array of names.
    array = CFArrayCreate( kCFAllocatorDefault,
                (const void **)names,
                kNumFamilyMembers,
                &kCFTypeArrayCallBacks );

    // Convert the plist into XML data.
    xmlData = CFPropertyListCreateXMLData( kCFAllocatorDefault, array );

    // Clean up CF types.
    CFRelease( array );
    CFRelease( xmlData );
}
```

Listing 2 (page 11) shows how the contents of `xmlData`, created in Listing 1 (page 11), would look if printed to the screen.

Listing 2 XML created by the sample program

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
        "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
    <string>Marge</string>
    <string>Homer</string>
    <string>Bart</string>
    <string>Lisa</string>
    <string>Maggie</string>
</array>
</plist>
```

CFPropertyList properly takes care of endian issues—a property list (whether represented by a stream, XML, or a CFData object) created on a PowerPC-based Macintosh is correctly interpreted on an Intel-based Macintosh, and vice versa.

Listing 1 (page 13) shows you how to create a more complex property list, convert it to XML, write it to disk, and then re-create the original data structure using the saved XML. For more information about using CFDictionary objects see Collections.

Listing 1 Saving and restoring property list data

```
#include <CoreFoundation/CoreFoundation.h>

#define kNumKids 2
#define kNumBytesInPic 10

CFDictionaryRef CreateMyDictionary( void );
CFPropertyListRef CreateMyPropertyListFromFile( CFURLRef fileURL );
void WriteMyPropertyListToFile( CFPropertyListRef propertyList,
            CFURLRef fileURL );

int main () {
   CFPropertyListRef propertyList;
   CFURLRef fileURL;

   // Construct a complex dictionary object;
   propertyList = CreateMyDictionary();

   // Create a URL that specifies the file we will create to
   // hold the XML data.
   fileURL = CFURLCreateWithFileSystemPath( kCFAllocatorDefault,
               CFSTR("test.txt"),       // file path name
               kCFURLPOSIXPathStyle,    // interpret as POSIX path
               false );                 // is it a directory?

   // Write the property list to the file.
   WriteMyPropertyListToFile( propertyList, fileURL );
   CFRelease(propertyList);

   // Recreate the property list from the file.
   propertyList = CreateMyPropertyListFromFile( fileURL );

   // Release any objects to which we have references.
   CFRelease(propertyList);
   CFRelease(fileURL);
   return 0;
}

CFDictionaryRef CreateMyDictionary( void ) {
   CFMutableDictionaryRef dict;
```

```
CFNumberRef            num;
CFArrayRef             array;
CFDataRef              data;

int                    yearOfBirth;
CFStringRef            kidsNames[kNumKids];

// Fake data to stand in for a picture of John Doe.
const unsigned char pic[kNumBytesInPic] = {0x3c, 0x42, 0x81,
         0xa5, 0x81, 0xa5, 0x99, 0x81, 0x42, 0x3c};

// Define some data.
kidsNames[0] = CFSTR("John");
kidsNames[1] = CFSTR("Kyra");

yearOfBirth = 1965;

// Create a dictionary that will hold the data.
dict = CFDictionaryCreateMutable( kCFAllocatorDefault,
         0,
         &kCFTypeDictionaryKeyCallBacks,
         &kCFTypeDictionaryValueCallBacks );

// Put the various items into the dictionary.
// Because the values are retained as they are placed into the
//  dictionary, we can release any allocated objects here.

CFDictionarySetValue( dict, CFSTR("Name"), CFSTR("John Doe") );

CFDictionarySetValue( dict,
         CFSTR("City of Birth"),
         CFSTR("Springfield") );

num = CFNumberCreate( kCFAllocatorDefault,
         kCFNumberIntType,
         &yearOfBirth );
CFDictionarySetValue( dict, CFSTR("Year Of Birth"), num );
CFRelease( num );

array = CFArrayCreate( kCFAllocatorDefault,
            (const void **)kidsNames,
            kNumKids,
            &kCFTypeArrayCallBacks );
CFDictionarySetValue( dict, CFSTR("Kids Names"), array );
CFRelease( array );

array = CFArrayCreate( kCFAllocatorDefault,
            NULL,
            0,
            &kCFTypeArrayCallBacks );
CFDictionarySetValue( dict, CFSTR("Pets Names"), array );
CFRelease( array );

data = CFDataCreate( kCFAllocatorDefault, pic, kNumBytesInPic );
CFDictionarySetValue( dict, CFSTR("Picture"), data );
CFRelease( data );

return dict;
```

```
}

void WriteMyPropertyListToFile( CFPropertyListRef propertyList,
            CFURLRef fileURL ) {
   CFDataRef xmlData;
   Boolean status;
   SInt32 errorCode;

   // Convert the property list into XML data.
   xmlData = CFPropertyListCreateXMLData( kCFAllocatorDefault, propertyList );

   // Write the XML data to the file.
   status = CFURLWriteDataAndPropertiesToResource (
               fileURL,                  // URL to use
               xmlData,                  // data to write
               NULL,
               &errorCode);

   CFRelease(xmlData);
}

CFPropertyListRef CreateMyPropertyListFromFile( CFURLRef fileURL ) {
   CFPropertyListRef propertyList;
   CFStringRef       errorString;
   CFDataRef         resourceData;
   Boolean           status;
   SInt32            errorCode;

   // Read the XML file.
   status = CFURLCreateDataAndPropertiesFromResource(
               kCFAllocatorDefault,
               fileURL,
               &resourceData,            // place to put file data
               NULL,
               NULL,
               &errorCode);

   // Reconstitute the dictionary using the XML data.
   propertyList = CFPropertyListCreateFromXMLData( kCFAllocatorDefault,
               resourceData,
               kCFPropertyListImmutable,
               &errorString);

   CFRelease( resourceData );
   return propertyList;
}
```

Listing 2 (page 15) shows how the contents of `xmlData`, created in Listing 1 (page 13), would look if printed to the screen.

Listing 2 XML file contents created by the sample program

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
        "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Year Of Birth</key>
    <integer>1965</integer>
    <key>Pets Names</key>
    <array/>
    <key>Picture</key>
    <data>
        PEKBpYGlmYFCPA==
    </data>
    <key>City of Birth</key>
    <string>Springfield</string>
    <key>Name</key>
    <string>John Doe</string>
    <key>Kids Names</key>
    <array>
        <string>John</string>
        <string>Kyra</string>
    </array>
</dict>
</plist>
```

Using Numbers in Property Lists

You cannot use C numeric data values directly in Core Foundation property lists. Core Foundation provides the function `CFNumberCreate` to convert C numerical values into CFNumber objects, the form that is required to use numbers in property lists.

A CFNumber object serves simply as a wrapper for C numeric values. Core Foundation includes functions to create a CFNumber, obtain its value, and compare two CFNumber objects. Note that CFNumber objects are immutable with respect to value, but type information may not be maintained. You can get information about a CFNumber object's type, but this is the type the CFNumber object used to store your value and *may not* be the same type as the original C data.

When comparing CFNumber objects, conversion and comparison follow human expectations and not C promotion and comparison rules. Negative zero compares less than positive zero. Positive infinity compares greater than everything except itself, to which it compares equal. Negative infinity compares less than everything except itself, to which it compares equal. Unlike standard practice, if both numbers are NaNs, then they compare equal; if only one of the numbers is a NaN, then the NaN compares greater than the other number if it is negative, and smaller than the other number if it is positive.

Listing 1 (page 17) shows how to create a CFNumber object from a 16-bit integer and then get information about the CFNumber object.

Listing 1 Creating a CFNumber object from an integer

```
Int16               sint16val = 276;
CFNumberRef         aCFNumber;
CFNumberType        type;
Int32               size;
Boolean             status;

// Make a CFNumber from a 16-bit integer.
aCFNumber = CFNumberCreate(kCFAllocatorDefault,
                           kCFNumberSInt16Type,
                           &sint16val);

// Find out what type is being used by this CFNumber.
type = CFNumberGetType(aCFNumber);

// Now find out the size in bytes.
size = CFNumberGetByteSize(aCFNumber);

// Get the value back from the CFNumber.
status = CFNumberGetValue(aCFNumber,
                          kCFNumberSInt16Type,
                          &sint16val);
```

Listing 2 (page 18) creates another CFNumber object and compares it with the one created in Listing 1 (page 17).

Listing 2 Comparing two CFNumber objects

```
CFNumberRef         anotherCFNumber;
CFComparisonResult  result;

// Make a new CFNumber.
sint16val = 382;
anotherCFNumber = CFNumberCreate(kCFAllocatorDefault,
                        kCFNumberSInt16Type,
                        &sint16val);

// Compare two CFNumber objects.
result = CFNumberCompare(aCFNumber, anotherCFNumber, NULL);

switch (result) {
    case kCFCompareLessThan:
        printf("aCFNumber is less than anotherCFNumber.\n");
        break;
    case kCFCompareEqualTo:
        printf("aCFNumber is equal to anotherCFNumber.\n");
        break;
    case kCFCompareGreaterThan:
        printf("aCFNumber is greater than anotherCFNumber.\n");
        break;
}
```

Property List XML Tags

When property lists convert a collection of Core Foundation objects into an XML property list, it wraps the property list using the document type tag `<plist>`. The other tags used for the Core Foundation data types are listed in Table 1 (page 19).

Table 1 Core Foundation Types with XML Equivalents

CF type	XML tag
CFString	`<string>`
CFNumber	`<real>` or `<integer>`
CFDate	`<date>`
CFBoolean	`<true/>` or `<false/>`
CFData	`<data>`
CFArray	`<array>`
CFDictionary	`<dict>`

When encoding the contents of a CFDictionary object, each member is encoded by placing the dictionary key in a `<key>` tag and immediately following it with the corresponding value in the appropriate tag from Table 1. See "Saving and Restoring Property Lists" (page 13) for an example XML data generated from a property list.

The XML data format is documented here strictly for help in understanding property lists and as a debugging aid. These tags may change in future releases so you shouldn't rely on them directly. You should not edit the XML data by hand unless you are very familiar with XML syntax and the format of property lists. If you want to modify the contents of a property list saved on disk as XML data, use the Property List Editor application.

Document Revision History

This table describes the changes to *Property List Programming Topics for Core Foundation*.

Date	Notes
2006-02-07	Consolidated articles about using numbers. Changed title from "Property Lists Programming Topics."
2003-08-07	Corrected XML tag descriptions.
2003-01-17	Converted existing Core Foundation documentation into topic format. Added revision history.