XML Programming Topics for Core Foundation

Core Foundation > Data Management

2008-10-15

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

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction to XML Programming Topics for Core Foundation

Core Foundation provides support for parsing XML documents into structured objects you can use in your programs. XML is a platform-independent and extensible markup language.

This document describes the Core Foundation objects that you use to parse XML documents.

Note: The Core Foundation XML API is not the preferred API to use when developing applications. Instead use the Cocoa NSXML API to get the most modern XML features. See *Tree-Based XML Programming Guide for Cocoa* for details.

Organization of This Document

Core Foundation provides an XML parser you can use to read and extract data from XML documents. Core Foundation provides two APIs with which to access the parser. A tree-based API converts XML data into the Core Foundation collection `CFXMLTree`, and an event-driven and callback-based API allows you to perform any action you wish on each XML structure as it is encountered by the parser. This topic provides a brief introduction to XML and goes on to describe both of the XML parser interfaces in detail.

You need to understand the following concepts to use the XML objects:

- "About XML" (page 9)
- "Core Foundation XML Parser" (page 11)

The following task demonstrates how to parse a simple XML document using each of the Core Foundation XML parser interfaces:

- "Parsing XML Documents" (page 15)

About XML

Extensible Markup Language, or XML, is a scripting language for representing structured data in a text file. The structured data you want to represent using XML can be virtually anything—address books, configuration parameters, spreadsheets, Web pages, financial transactions, technical drawings, and so on. XML defines a set of rules for designing text formats for such data. By storing data in a structured text format, XML allows you to look at data without the program that produced it. XML files are easy for computers to generate and read, they are unambiguous, and they avoid common pitfalls of text data formats, such as lack of extensibility, lack of support for internationalization and localization, and platform dependency.

XML is a complex subject whose thorough treatment is beyond the scope of this topic. Developers new to XML concepts can find the XML 1.0 specification and supporting material at the website maintained by the World Wide Web Consortium at `http://www.w3c.org/XML`. The Organization for the Advancement of Structured Information Standards hosts two excellent sites on XML at `http://www.xml.org/` and `http://www.oasis-open.org/cover/`. There is also a great deal of information about XML and its various uses at the following corporate sites: `http://www.ibm.com/developer/xml/`, `http://msdn.microsoft.com/xml/default.asp`, and `http://java.sun.com/xml/`.

XML Syntax

Like HTML, XML is based on the Standard Generalized Markup Language, or SGML. This common heritage renders XML familiar in look and feel to those accustomed to HTML. Unlike HTML, though, XML syntax requires the use of matching start and end tags, such as `<string>` and `</string>`, to demarcate logical sections of a document or data sets. A unit of information enclosed by tags is called an **element**. As a shortcut, if an element has no content between its start and end tags, the tags can be merged into a single tag that ends with `"/>"`, such as `<true/>`. This simple syntax is easy to process by a computer, with the added benefit of remaining understandable to humans.

The best way to illustrate the basic features of XML is with a simple example. The document shown in Listing 1 contains the XML representation of a customer object that might have been exported from a customer database.

Listing 1 A simple XML document

```
<?xml version="1.0" encoding="UTF-8"?>
<customer>
    <name>Jane Doe</name>
    <address region="USA">
        <street>6236 Nicolet Rd</street>
        <city>Richmond</city>
        <state>VA</state>
        <postal>23225</postal>
    </address>
    <birthday>
        <month>10</month>
        <day>11</day>
```

```
        <year>1949</year>
    </birthday>
</customer>
```

This example document contains the basic XML structural features. First there is the required prolog—also called the XML declaration—containing XML version and character encoding information. (In the absence of an encoding attribute, Core Foundation assumes UTF-8.) The remainder of the document is simply the listing of elements that constitute the customer information.

XML Parsers

In computing terms, a parser is a program that takes input in the form of sequential instructions, tags, or some other defined sequence of tokens, and breaks them up into easily manageable parts. An XML parser is designed to read and, in a sense, interpret XML documents. As it executes, the parser recognizes and responds to each XML structure it encounters by taking some specified action based on the structure type. Many XML parsers, called tree-based parsers, convert an XML document into a tree structure that reflects the structural hierarchy of the XML data. This tree is then made available to your application, which is free to interpret and modify the data as appropriate. Other parsers are event-driven, and report to their client each XML construct they encounter.

In addition to being event-driven or tree-based, XML parsers can be validating or nonvalidating. Validating parsers check a document's contents against a set of specific rules stating what elements are allowed in a document and in what order they must appear. These rules appear in an XML document either as an optional XML structure called a document type definition, or DTD, or as an XML Schema. Nonvalidating parsers are smaller and faster, but they don't check documents against the DTD; they only check if the document is structurally well-formed.

Core Foundation XML Parser

Core Foundation provides a parser that your applications can use to read data in XML format. Core Foundation's XML parser has two programming interfaces, one tree-based and the other event-driven. The tree-based interface parses an XML document and returns the data to you in the form of a `CFXMLTreeRef` object. There is also a configurable, callback-based API that allows event-driven parsing of an XML document. Event-driven parsing allows you to customize the parser's behavior so your application can respond only to the specific XML constructs that interest you. Event-driven parsing is also useful for large documents because the parser doesn't have to build the entire tree in memory. However, tree-based parsing allows you to add or modify nodes in the tree structure, and thus modify the original XML document.

CFXMLNode Objects

Both of the XML parser interfaces rely on a single data structure to return XML data to your application: the `CFXMLNodeRef` opaque object. This Core Foundation type describes an individual XML construct, such as a element, a comment, an attribute, or a string of character data.

Each `CFXMLNode` object contains three main pieces of information—the node's type, the data string, and a pointer to a data structure with additional information. You extract this data using simple accessor functions. The node's type is encoded as an enumeration constant describing the type of XML structure. The data string is always a `CFString` object; the meaning of the string depends on the node's type ID. The format of the additional data also depends on the node's type; there is a specific structure for each type that requires additional data.

As it processes an XML document, the parser converts each XML construct it encounters into a `CFXMLNode` object that represents that construct. For example, when parsing the document shown in Listing 1 (page 9), the parser would respond to the tag `<birthday>` by creating a new `CFXMLNode` whose node type would be set to the identifier `kCFXMLNodeTypeElement`. The `CFXMLNode` data string would contain the `CFString` object "birthday", and the additional data pointer would point to a `CFXMLElementInfo` structure containing information about the element's attributes.

In order to handle some of the more complex XML entities, Core Foundation defines several additional data structures. The structures that contain additional information are described briefly in Table 1.

Table 1 XML parser additional information structures

Structure	Content Description
`CFXMLElementInfo`	A list of element attributes.
`CFXMLProcessingInstructionInfo`	The processing instruction.
`CFXMLDocumentInfo`	The source URL for the document along with character encoding information.

Structure	Content Description
`CFXMLDocumentTypeInfo`	The system and public IDs for the DTD.
`CFXMLNotationInfo`	The system and public IDs for the notation.
`CFXMLElementTypeDeclarationInfo`	The string that describes the element's permissible content.
`CFXMLAttributeDeclarationInfo`	The name of the attribute being declared, the string describing the attribute's type, and the attribute's default value.
`CFXMLAttributeList-DeclarationInfo`	A list of `CFXMLAttributeDeclarationInfo` structures.
`CFXMLEntityInfo`	The type of the entity, the text to be substituted for the entity when referenced, the location of the entity (for external entities), and the name of the entity's notation if the entity is not parsed.
`CFXMLEntityReferenceInfo`	The type of the entity reference.

To briefly illustrate how these structures are used by the parser, consider once again the XML document shown in Listing 1 (page 9). The fourth line of the document contains the tag `<address region="USA">`. The string `region="USA"` defines an **element attribute** called `region` whose string value is `USA`. Element attributes are a way to associate additional data with a given element.

The XML parser returns a tag's attributes to your application as a `CFXMLElementInfo` structure. This structure is shown in Listing 1.

Listing 1 The CFXMLElementInfo structure

```
typedef struct {
    CFDictionaryRef attributes;
    CFArrayRef attributeOrder;
    Boolean isEmpty;
} CFXMLElementInfo;
```

When parsing this tag, the parser creates a `CFXMLNode` object whose type code is `kCFXMLNodeTypeElement`, and whose data string is `"address"`. The additional information pointer is set to point to a `CFXMLElementInfo` structure describing the element and its attributes. The `attributes` field contains a `CFDictionary` object holding the attribute data in the key/value format. The `attributeOrder` field contains a `CFArray` object holding the `attributes` dictionary keys in the order they were encountered. The Boolean value of the `isEmpty` field indicates whether the element is empty. See *Collections Programming Topics for Core Foundation* for more information about `CFDictionary` and `CFArray`.

Tree-Based Parser API

The tree-based parser API provides a very simple method for reading XML data. One call to the function `CFXMLTreeCreateFromData` reads an entire XML document—specified by a pointer to XML data in memory, or by a URL string—and returns the XML data to you in the form of a `CFXMLTree` object. A `CFXMLTree` object is simply a `CFTree` object that contains a pointer to a `CFXMLNode` object in each node's context. See *Collections Programming Topics for Core Foundation* for more information about `CFTreeRef` and its API.

Once the `CFXMLTree` object has been created, you can use the `CFTree` API to examine the tree and extract information from a given node. Core Foundation also provides convenience functions that make it even easier to access the content of a `CFXMLTree` object. For example, `CFXMLTreeGetNode` takes a reference to one of the tree's nodes and returns a pointer to that node.

The section "Using the Tree-Based Parser Interface" (page 15) shows you how to parse, examine, and modify an XML document using the tree-based parser API.

Event-Driven Parser API

The tree-based XML parser API is sufficient for many needs. However, there are some cases where using the event-driven interface of `CFXML` is appropriate:

- You want fine-tuned control of the parsing process.
- You need access to data within a very large XML document and converting the entire document into a `CFXMLTree` object requires too much memory.
- A `CFXMLTree` object is inappropriate for your application's needs, and you want to build a custom data structure from the contents of an XML document.
- You wish to provide additional error checking as parsing progresses.
- You wish to control when and how external entities are loaded.

For these and other situations you can use the callback-based event-driven API. This API is somewhat more complex to use, but provides much more flexibility than the tree-based API.

Conceptually, the event-driven API is simple. You first define a set of callback functions that are invoked as the parsing process proceeds. As the parser encounters each XML structure, your functions are called, giving you an opportunity to handle the data however you wish.

Parser Callbacks

In order to use the event-driven parser, you must implement three of the five callbacks described in this section—`CFXMLParserCreateXMLStructureCallBack`, `CFXMLParserAddChildCallBack`, and `CFXMLParserEndXMLStructureCallBack`. The other callbacks are optional.

The `CFXMLParserCreateXMLStructureCallBack` function is called when the parser encounters a new XML structure. It passes a pointer to a CFXMLNode. If the function returns `NULL`, the parser skips the structure.

The `CFXMLParserAddChildCallBack` function is called when the parser encounters a child structure. It notifies you of the parent–child relationship and passes the data you returned from `CFXMLParserCreateXMLStructureCallBack` for both the parent and child.

The `CFXMLParserEndXMLStructureCallBack` function is called when the parser exits an XML structure reported by `CFXMLParserCreateXMLStructureCallBack`. It passes the data you returned from `CFXMLParserCreateXMLStructureCallBack`.

The `CFXMLParserResolveExternalEntityCallBack` function is called when the parser encounters an XML external entity reference. It passes the `publicID` and `systemID` data for the entity. It is up to you to load the data if you wish and return it as a CFData. Not currently supported.

The `CFXMLParserHandleErrorCallBack` is called when the parser encounters an error condition. It passes an error code indicating the nature of the error. From within your error handler, you can use the function `CFXMLParserCopyErrorDescription` to get a CFString describing the error condition. You can also use the functions `CFXMLParserGetLineNumber` and `CFXMLParserGetLocation` to learn the exact location of the error within the XML document.

At any point during the parsing you can use the function `CFXMLParserGetStatusCode` to find out what the parser is doing. You can also call `CFXMLParserAbort` to signal an error.

Parser Option Flags

There are various options you can use to configure the parser's behavior. An option flag of `0`, or `kCFXMLParserNoOptions`, leaves the XML as "intact" as possible. In other words, this option causes the parser to report all structures and performs no entity replacements. To make the parser do the most work, returning only the pure element tree, set the option flag to `kCFXMLParserAllOptions`.

Table 2 Parser option Flags

Flag	Description	Status
`kCFXMLParserValidate-Document`	Validate the document against its DTD schema, reporting any errors.	Not supported
`kCFXMLParserSkipMetaData`	Silently skip over metadata constructs (the DTD and comments).	Supported
`kCFXMLParserReplace-PhysicalEntities`	Replace declared entities like `<`.	Not supported
`kCFXMLParserSkip-Whitespace`	Skip over all whitespace that does not abut non-whitespace character data. For example, given `<foo> <bar> blah </bar></foo>`, the whitespace between foo's open tag and bar's open tag would be suppressed, but the whitespace around blah would be preserved.	Supported
`kCFXMLParserAdd-ImpliedAttributes`	Where the DTD specifies implied attribute-value pairs for a particular element, add those pairs to any occurrences of the element in the element tree.	Not Supported
`kCFXMLParserAllOptions`	All of the supported options.	Supported
`kCFXMLParserNoOptions`	No options.	Supported

The section "Using the Event-Driven Parser Interface" (page 17) shows you how to parse an XML document using the event-driven parser API.

The document shown in Listing 1 contains the XML representation of a very simple Core Foundation property list created using `CFPropertyList`. Note that a property list was chosen purely for the purposes of illustrating XML parser usage in a Core Foundation context. `CFPropertyList` has convenience functions for converting property lists to and from XML format, so in most cases your application would not need to parse an XML property list using the XML parser directly (see *Property List Programming Topics for Core Foundation* for more information).

Listing 1 A Core Foundation property list in XML format

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist SYSTEM "file://localhost/System/Library/DTDs/PropertyList.dtd">
<plist version="0.9">
<dict>
    <key>Jane Doe</key>
    <integer>1999</integer>
    <key>John Doe</key>
    <integer>2000</integer>
</dict>
</plist>
```

In this example XML document, the data consists of two names and associated birth years. The `<plist>` tag declares that the enclosed data is a property list that corresponds to the Core Foundation data type `CFPropertyList`. The `<dict>` tag declares that its enclosed data corresponds to a CFDictionary. Finally, the name and birth year data are listed in the key/value pair format required for a `CFDictionary` object.

Using the Tree-Based Parser Interface

Listing 2 shows how you would use the high level XML API to convert the sample XML data in Listing 1 (page 15) into a `CFXMLTree` object. This example assumes that `sourceURL` is a valid `CFURL` object and refers to the XML document.

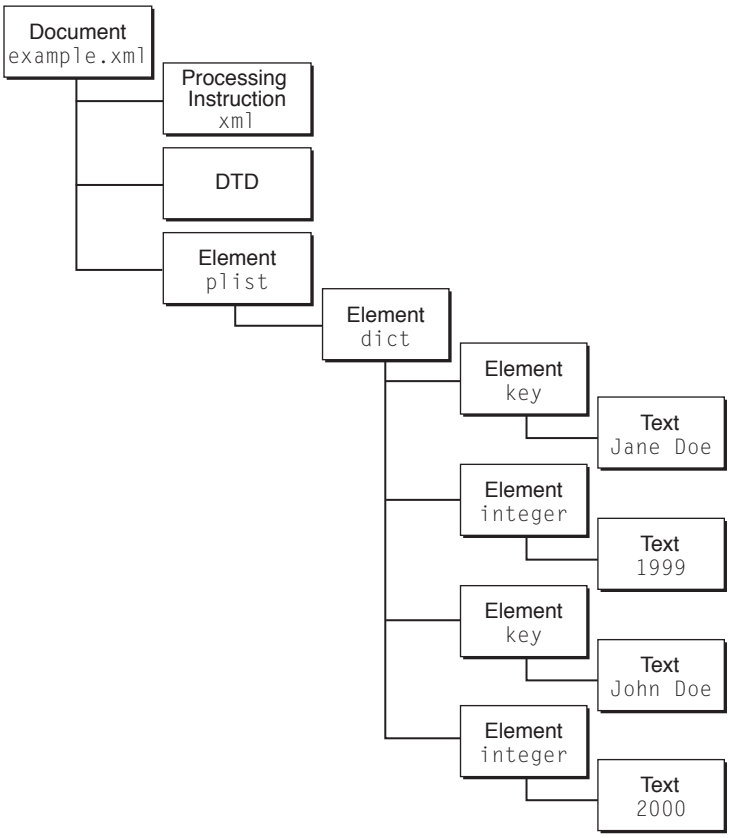
Listing 2 Using the tree-based parser API

```
CFXMLTreeRef    cfXMLTree;
CFDataRef       xmlData;

// Load the XML data using its URL.
CFURLCreateDataAndPropertiesFromResource(kCFAllocatorDefault,
                sourceURL, &xmlData, NULL, NULL, NULL)

// Parse the XML and get the CFXMLTree.
cfXMLTree = CFXMLTreeCreateFromData(kCFAllocatorDefault,
                    xmlData,
                    sourceURL,
                    kCFXMLParserSkipWhitespace,
```

```
                    kCFXMLNodeCurrentVersion);
```

Figure 1 illustrates the structure of the `CFXMLTree` object produced by the code in Listing 2 (page 15). As you would expect, it exactly reflects the structure of the original XML document. The diagram displays the data type code and data string from each `CFXMLNode` object.

Figure 1 The structure of a CFXMLTree



The example in Listing 3 shows how to use some of the XML convenience functions to examine the top level of a `CFXMLTree` object and print out each node's data string contents.

Listing 3 Obtaining information from a CFXMLTree

```
CFXMLTreeRef    xmlTreeNode;
CFXMLNodeRef    xmlNode;
int             childCount;
int             index;

// Get a count of the top level node's children.
childCount = CFTreeGetChildCount(cfXMLTree);

// Print the data string for each top-level node.
for (index = 0; index < childCount; index++) {
    xmlTreeNode = CFTreeGetChildAtIndex(cfXMLTree, index);
    xmlNode = CFXMLTreeGetNode(xmlTreeNode);
    CFShow(CFXMLNodeGetString(xmlNode));
}
```

Using the Event-Driven Parser Interface

The event-driven parser API gives you complete flexibility to do whatever you wish with the data in an XML document. To use the event-driven parser API, you define a set of callback functions that the parser invokes as it encounters specific structures in the XML document. The code in this section shows how to use the event-driven parser to print the data in an XML document. A sample implementation for each callback function is shown, and then the code to create and run the parser.

The code in Listing 4 implements the first—and by far the longest—callback function, `CFXMLParserCreateXMLStructureCallBack`. This example implementation prints the contents of each new XML structure's additional information data as it is encountered.

Listing 4 Implementing the CFXMLParserCreateXMLStructureCallBack function

```
void *createStructure(CFXMLParserRef parser,
            CFXMLNodeRef node, void *info) {

    CFStringRef myTypeStr;
    CFStringRef myDataStr;
    CFXMLDocumentInfo *docInfoPtr;

    // Use the dataTypeID to determine what to print.
    switch (CFXMLNodeGetTypeCode(node)) {
        case kCFXMLNodeTypeDocument:
            myTypeStr = CFSTR("Data Type ID: kCFXMLNodeTypeDocument\n");
            docInfoPtr = CFXMLNodeGetInfoPtr(node);
            myDataStr = CFStringCreateWithFormat(NULL,
                        NULL,
                        CFSTR("Document URL: %@\n"),
                        CFURLGetString(docInfoPtr->sourceURL));
            break;
        case kCFXMLNodeTypeElement:
            myTypeStr = CFSTR("Data Type ID: kCFXMLNodeTypeElement\n");
            myDataStr = CFStringCreateWithFormat(NULL, NULL,
                    CFSTR("Element: %@\n"), CFXMLNodeGetString(node));
            break;
        case kCFXMLNodeTypeProcessingInstruction:
            myTypeStr = CFSTR("Data Type ID:
                    kCFXMLNodeTypeProcessingInstruction\n");
            myDataStr = CFStringCreateWithFormat(NULL, NULL,
                    CFSTR("PI: %@\n"), CFXMLNodeGetString(node));
            break;
        case kCFXMLNodeTypeComment:
            myTypeStr = CFSTR("Data Type ID: kCFXMLNodeTypeComment\n");
            myDataStr = CFStringCreateWithFormat(NULL, NULL,
                    CFSTR("Comment: %@\n"), CFXMLNodeGetString(node));
            break;
        case kCFXMLNodeTypeText:
            myTypeStr = CFSTR("Data Type ID: kCFXMLNodeTypeText\n");
            myDataStr = CFStringCreateWithFormat(NULL, NULL,
                    CFSTR("Text:%@\n"), CFXMLNodeGetString(node));
            break;
```

```
        case kCFXMLNodeTypeCDATASection:
            myTypeStr = CFSTR("Data Type ID: k
                    CFXMLDataTypeCDATASection\n");
            myDataStr = CFStringCreateWithFormat(NULL, NULL,
                    CFSTR("CDATA: %@\n"), CFXMLNodeGetString(node));
            break;
        case kCFXMLNodeTypeEntityReference:
            myTypeStr = CFSTR("Data Type ID:
                    kCFXMLNodeTypeEntityReference\n");
            myDataStr = CFStringCreateWithFormat(NULL, NULL,
                    CFSTR("Entity reference: %@\n"),
                    CFXMLNodeGetString(node));
            break;
        case kCFXMLNodeTypeDocumentType:
            myTypeStr = CFSTR("Data Type ID: kCFXMLNodeTypeDocumentType\n");
            myDataStr = CFStringCreateWithFormat(NULL, NULL,
                    CFSTR("DTD: %@\n"), CFXMLNodeGetString(node));
            break;
        case kCFXMLNodeTypeWhitespace:
            myTypeStr = CFSTR("Data Type ID: kCFXMLNodeTypeWhitespace\n");
            myDataStr = CFStringCreateWithFormat(NULL, NULL,
                    CFSTR("Whitespace: %@\n"), CFXMLNodeGetString(node));
            break;
        default:
            myTypeStr = CFSTR("Data Type ID: UNKNOWN\n");
            myDataStr = CFSTR("Unknown type.\n");
        }

    // Print the contents.
    printf("---Create Structure Called--- \n");
    CFShow(myTypeStr);
    CFShow(myDataStr);

    // Return the data string for use by the addChild and
    // endStructure callbacks.
    return myDataStr;
}
```

Notice that the `CFXMLParserCreateXMLStructureCallBack` function returns the data string created using the `dataString` field of the newly encountered structure. This return value can actually be anything, but is kept by the parser and passed back to you by both the `CFXMLParserAddChildCallBack` and `CFXMLParserEndXMLStructureCallBack` functions described below. Note that if your `CFXMLParserCreateXMLStructureCallBack` function returns `NULL`, `CFXMLParserAddChildCallBack` and `CFXMLParserEndXMLStructureCallBack` will *not* be called. The only exception is `CFNodeTypeDocument`; `CFXMLParserEndXMLStructureCallBack` will be called for it even if you return `NULL` from `CFXMLParserCreateXMLStructureCallBack`.

The parser invokes the `CFXMLParserAddChildCallBack` when it encounters a child of the most recently parsed structure. In this example, the `CFXMLParserAddChildCallBack` callback shown in Listing 5 simply prints out both of the strings to make clear the parent–child relationships of the XML structures being parsed.

Listing 5 Implementing the CFXMLParserAddChildCallBack function

```
void addChild(CFXMLParserRef parser, void *parent, void *child, void *info) {
    printf("---Add Child Called--- \n");
    printf("Parent being added to: "); CFShow((CFStringRef)parent);
    printf("Child being added: "); CFShow((CFStringRef)child);
```

```
}
```

The parser calls the `CFXMLParserEndXMLStructureCallBack` function, implemented in Listing 6, when it moves beyond a given structure. The `xmlType` parameter is a pointer to whatever data the `CFXMLParserCreateXMLStructureCallBack` function returned when the structure's open tag was first encountered. In this example implementation, the callback prints out a string indicating which structure has ended.

Listing 6 Implementing the endStructure callback

```
void endStructure(CFXMLParserRef parser, void *xmlType, void *info) {
    // Leave evidence that we were called.
    printf("---End Structure Called for \n"); CFShow((CFStringRef)xmlType);

    // Now that the structure and all of its children have been parsed,
    // we can release the string.
    CFRelease(xmlType);
}
```

The parser calls the `CFXMLParserResolveExternalEntityCallBack` function when it encounters an external entity reference. The example XML data in this section contains no entity references so this callback is not invoked. Listing 7 shows a minimal implementation.

Listing 7 Implementing the CFXMLParserResolveExternalEntityCallBack function

```
CFDataRef resolveEntity(CFXMLParserRef parser, CFStringRef publicID,
        CFURLRef systemID, void *info) {
    printf("---resolveEntity Called---\n");
    return NULL;
}
```

The parser calls the `CFXMLParserHandleErrorCallBack` callback when it encounters an error condition. As shown in Listing 8, you can use the XML API to get both the error string and error location information from the parser. If you return `false` from this callback, the parser aborts. If you return `true` and the error is nonfatal, the parser continues processing.

Listing 8 Implementing the handleError CFXMLParserHandleErrorCallBack function

```
Boolean handleError(CFXMLParserRef parser, SInt32 error, void *info) {
    char buf[512], *s;

    // Get the error description string from the Parser.
    CFStringRef description = CFXMLParserCopyErrorDescription(parser);
    s = (char *)CFStringGetCStringPtr(description,
                    CFStringGetSystemEncoding());

    // If the string pointer is unavailable, do some extra work.
    if (!s) {
        CFStringGetCString(description, buf, 512,
                    CFStringGetSystemEncoding());
    }

    CFRelease(description);

    // Report the exact location of the error.
    fprintf(stderr, "Parse error (%d) %s on line %d, character %d\n",
                    (int)error,
```

```
                    s,
                    (int)CFXMLParserGetLineNumber(parser),
                    (int)CFXMLParserGetLocation(parser));

    return false;
}
```

Listing 9 demonstrates how to create and invoke the parser.

Listing 9 Creating and invoking the XML parser

```
// First, set up the parser callbacks.
CFXMLParserCallBacks callbacks = {0, createStructure, addChild, endStructure,
resolveEntity, handleError};

// Create the parser with the option to skip whitespace.
parser = CFXMLParserCreate(kCFAllocatorDefault, xmlData, urlOut,
kCFXMLParserSkipWhitespace, kCFXMLNodeCurrentVersion, &callbacks, NULL);

// Invoke the parser.
if (!CFXMLParserParse(parser)) {
    printf("parse failed\n");
}
```

As you can see, once the callbacks have been implemented, the code to create and call the parser is quite simple. "`Parser output`" shows the output generated by the code in "`Creating and invoking the XML parser`" (page 20).

Listing 10 Parser output

```
---Create Structure Called---
   Data Type ID: kCFXMLNodeTypeDocument, Document: file://localhost/myPlist.xml
---Create Structure Called---
    Data Type ID: kCFXMLNodeTypeProcessingInstruction, PI: xml
---Add Child Called---
    Parent being added to: Document: file://localhost/myPlist.xml
    Child being added: PI: xml
---End Structure Called for PI: xml
---Create Structure Called---
    Data Type ID: kCFXMLNodeTypeDocumentType, DTD
---Add Child Called---
    Parent being added to: Document: file://localhost/myPlist.xml
    Child being added: DTD
---End Structure Called for DTD
---Create Structure Called---
    Data Type ID: kCFXMLNodeTypeElement, Element: plist
---Add Child Called---
    Parent being added to: Document: file://localhost/myPlist.xml
    Child being added: Element: plist
---Create Structure Called---
    Data Type ID: kCFXMLNodeTypeElement, Element: dict
---Add Child Called---
    Parent being added to: Element: plist
    Child being added: Element: dict
---Create Structure Called---
    Data Type ID: kCFXMLNodeTypeElement, Element: key
---Add Child Called---
    Parent being added to: Element: dict
```

```
    Child being added: Element: key
---Create Structure Called---
    Data Type ID: kCFXMLNodeTypeText, Text: Jane Doe
---Add Child Called---
    Parent being added to: Element: key
    Child being added: Text: Jane Doe
---End Structure Called for Text: Jane Doe
---End Structure Called for Element: key
---Create Structure Called---
    Data Type ID: kCFXMLNodeTypeElement, Element: integer
---Add Child Called---
    Parent being added to: Element: dict
    Child being added: Element: integer
---Create Structure Called---
    Data Type ID: kCFXMLNodeTypeText, Text: 1999
---Add Child Called---
    Parent being added to: Element: integer
    Child being added: Text: 1999
---End Structure Called for Text: 1999
---End Structure Called for Element: integer
---Create Structure Called---
    Data Type ID: kCFXMLNodeTypeElement, Element: key
---Add Child Called---
    Parent being added to: Element: dict
    Child being added: Element: key
---Create Structure Called---
    Data Type ID: kCFXMLNodeTypeText, Text: John Doe
---Add Child Called---
    Parent being added to: Element: key
    Child being added: Text: John Doe
---End Structure Called Text: John Doe
---End Structure Called for Element: key
---Create Structure Called---
    Data Type ID: kCFXMLNodeTypeElement, Element: integer
---Add Child Called---
    Parent being added to: Element: dict
    Child being added: Element: integer
---Create Structure Called---
    Data Type ID: kCFXMLNodeTypeText, Text: 2000
---Add Child Called---
    Parent being added to: Element: integer
    Child being added: Text: 2000
---End Structure Called for Text: 2000
---End Structure Called for Element: integer
---End Structure Called for Element: dict
---End Structure Called for Element: plist
```

Document Revision History

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

Date	Notes
2008-10-15	Made various small corrections.
2006-10-03	Updated terminology, reorganized, and edited. Changed title from "XML".
2003-09-10	Removed erroneous call to `CFRelease()` in implementation of `createStructure`.
2003-01-17	Converted existing Core Foundation documentation into topic format. Added revision history.