IBM® Workplace Forms™ Server - API

Version 2.6

C API User's Manual

S229-1526-00

Note
Before using this information and the product it supports, read the information in "Notices," on page 207.

First Edition (September 2006)

This edition applies to version 2.6 of IBM Workplace Forms Server - API (product number L-DSED-6JLR37) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Introduction

Welcome to the C Edition of the user's manual for the IBM® Workplace Forms™ Server — API. The API extends the capabilities of Workplace Forms by enabling you to:

- Manipulate XFDL forms from new or existing applications.
- Create custom-built functions that may be integrated into XFDL forms.

This section discusses the organization and format of this manual. To learn more about the API, refer to "About the API" on page 3.

About This Manual

This manual has been organized as both an instruction manual and a quick reference. It describes the functions available in the API and provides examples of their use.

This manual contains the following major sections:

Section	Page
Introduction — introduces you to the features of the API.	"Introduction"
Overview of the Form Structure — explains how XFDL forms are stored in memory.	"Overview of the Form Structure" on page 7
Getting Started with the Form Library — provides a detailed tutorial demonstrating how to create a simple application that interacts with an XFDL form.	"Introduction to the Form Library" on page 15
Form Library Quick Reference — a reference to the functions contained in the Form Library. Each method description includes sample code.	"Form Library Quick Reference Guide" on page 27
Getting Started with the FCI Library — provides a detailed tutorial demonstrating how to create a simple function that you can call from an XFDL form.	"Introduction to the FCI Library" on page 163
FCI Quick Reference Guide — a reference to the Java methods contained in the FCI API. Each method description includes sample code.	"FCI Library Quick Reference Guide" on page 179

Who Should Read This Manual

The API is designed to be easy to use for any moderately experienced programmer. However, the skill level required to develop particular functions may be quite high. This document is intended for developers who have a working knowledge of:

- C programming and syntax.
- Extensible Forms Description Language (XFDL) and syntax. Refer to the *Extensible Forms Description Language Specification* for more information.

Document Conventions

The following conventions appear throughout this manual:

- Sample code is presented in a monospaced font, and is indented to make the code stand out:

```
r_short C_ExtensionInit(Extension* theExtension,IFX theIFX)
   FunctionCall *theFC;
   r_short theError;
```

- Text in bold italics represents information that you need to supply:

```
<label sid="firstName">
   <value>your first name here</value>
<label>
```

- The hash symbol (#) represents a number.
- Angle brackets enclose placeholders. For example, <*API Program Folder*> represents the actual folder in which you installed the API.
- Braces indicate optional items. The following example indicates that the item tag (including the period after it) is optional:

```
{itemtag.} option
```

- "*xx*" or "*xxx*" appears in place of the two or three digit version number of the API. In particular, these placeholders appear when referring to file names, folders, and directories that contain the API's version number.
- Brackets are used to indicate a sequence of choices, and the pipe symbol (|) is used to indicate "or". The following example indicates that you can use a number or a name:

```
(number|name)
```

About the API

The Workplace Forms Server — Application Programmer Interface (API) consists of a collection of programming tools to help you develop applications that can interact with XFDL forms. These tools are available for both C and Java programming environments. The API enables you to access and manipulate forms as structured data types.

The API is divided into two libraries: the Form Library and the Function Call Interface (FCI) Library. The Form Library allows you to create applications that:

- Read and write forms.
- Retrieve information from form elements.
- Add cells to certain form items.
- Insert information into form elements.

For more information about the Form Library refer to the "Form Library Quick Reference Guide" on page 27.

The Function Call Interface (FCI) Library provides additional methods that:

- Create, duplicate, or delete form elements.
- Manipulate and verify digital signatures.
- Handle attachments.
- Create custom functions for use within XFDL forms.

For more information about the FCI Library refer to "Introduction to the FCI Library" on page 163.

Where the API Fits in Your System

IBM provides a powerful suite of forms software for creating, using and transmitting forms over the Internet. The main components of this suite are:

Workplace Forms Viewer — Use the Viewer to view XFDL forms just as you would use a web browser to view HTML pages. You can also use the Viewer to fill out forms and submit them for review.

Workplace Forms Designer — The Designer provides an easy to use WYSIWYG design environment for creating XFDL forms. Use the Designer to create forms quickly and easily.

Workplace Forms API — The API is made up of Form and FCI functions. Use the Form Library of functions to develop applications that manipulate XFDL forms. Use the FCI functions to develop customized functions that can be called from within forms.

Differences Between the C, Java, and COM Editions of the API

The various editions of the API differ in the following ways:

- The Java and COM editions offer an object-oriented interface.
- The COM edition does not support the FCI Library.
- The COM edition does not include the following Form Library functions:
  - GetInfoEx
  - GetAttributeList
- The COM edition includes the following Form library functions that the other editions do not:
  - GetType
  - GetIdentifier
  - ReadFormFromASPRequest
  - WriteFormToASPResponse

In all other respects, the different editions of the API provide the same functionality, and use the same memory model for forms.

The API Data Types

The C Edition of the API uses custom defined types. These data types allow applications that use the API to be easily ported between various 32-bit systems without requiring any source code to be rewritten or recompiled. The following table lists the pre-defined data types:

Data Type	Description
r_short	signed **short** (16 bit integer)
r_u_short	unsigned **short** (16 bit integer)
r_long	signed **long** (32 bit integer)
r_u_long	unsigned **long** (32 bit integer)
r_charP	character pointer (**String**)
r_byte	byte
rc_char	**constant** character
rc_charP	**constant** character pointer (**String**)
rc_short	**constant** signed **short** (16 bit integer)
rc_u_short	**constant** unsigned **short** (16 bit integer)
rc_long	**constant** signed **long** (32 bit integer)
rc_u_long	**constant** unsigned **long** (32 bit integer)
rc_longlong	**constant** signed **longlong** (64 bit integer)
rc_u_longlong	**constant** unsigned **longlong** (64 bit integer)
r_voidP	**void** character pointer

formNodeP

The functions in the Form Library store forms in memory as a series of linked nodes. Each node, regardless of its level in the hierarchy, is represented by a **formNodeP**, which is an object-like structure.

The functions in the Form Library are responsible for creating and populating these nodes, and for freeing the memory they occupy.

About Memory Use

The Form methods are responsible for creating and populating these nodes. Furthermore, once you are done working with a form, you must use the **destroy** method on the root node of the form to remove it from memory.

r_charP Strings

Literal values associated with a **formNodeP** are stored using the **r_charP** structure. This structure is a character pointer that can be used to identify a string. A typical C style pointer has been replaced to ensure compatibility across all supported operating systems.

The API Memory and String Functions

The following memory and string functions have been developed to ensure cross-platform consistency in memory handling. You should always use these functions when you are programming with the other functions in this API. The calls for these functions parallel the calls for the regular ANSI C functions, and you should refer to your ANSI C reference material for more complete explanations

ANSI C Function	Equivalent API Function
free	For strings: **void cp_free(r_charP** *theString***)** For other data: **void pe_free**(**r_byte*** *theObject*)
malloc	For strings: **r_charP cp_malloc(long** *theSize***)** For other data: **r_byte* pe_malloc**(**long** *theSize*)
memcpy	**r_byte* pe_memcpy(r_byte*** *theDest*, **r_byte*** *theSrc*, **long** *n***)**
realloc	For strings: **r_charP cp_realloc(r_charP** *oldString*, **long** *newSize*, **long** *oldSize***)** For other data: **r_byte* pe_realloc**(**r_byte*** oldString, **long** newSize, **long** *oldSize*) **Note** This function takes three parameters, rather than the standard two.
strcat	**r_charP cp_strcat(r_charP** *theDest*, **r_charP** *theSrc***)**
strchr	**r_charP cp_strchr(r_charP** *theString*, **ra_char** *theChar***)**
strdup	**r_charP cp_strdup(r_charP** *theString***)**
strcmp	**r_int cp_strcmp(r_charP** *string1*, **r_charP** *string2***)**
strcpy	**r_charP cp_strcpy(r_charP** *theDest*, **r_charP** *theSrc***)**
strlen	**long cp_strlen(r_charP** *theString***)**

ANSI C Function	Equivalent API Function
strncmp	**int cp_strncmp(r_charP** *string1*, **r_charP** *string2*, **long** *n*)
strrchr	**r_charP cp_strrchr(r_charP** *theString*, **ra_char** *theChar*)
strstr	**r_charP cp_strstr (r_charP** *theString*, **r_charP** *searchString*)

About the API Constants

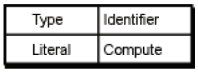
Several of the Form Library functions accept parameters whose values are constants (for example, the *referenceType* parameter of **UFLDereferenceEx**). All constants have been defined in the API and are available by including xfdlib.h at the top of your .c source file.

Overview of the Form Structure

This section provides an overview of an XFDL form as it is represented in memory. Developers must understand the memory structure of a form to effectively develop applications using the API.

The Node Structure

When a form is loaded into memory, it is constructed as a series of linked nodes. Each node represents an element of the form, and together these nodes create a tree that describes the form. The following diagram illustrates the general composition of a single node.



Each node within the tree has the following properties:

- **Type** — For page and item nodes, this describes the type of node, such as *button*, *line*, *field*, and so on. Page nodes are always of type *page*.
- **Literal** — The literal value of the node (for example, a literal string). If the node has a formula, the result of the formula will be stored here.
- **Identifier** — The page tag, item tag, option name, or custom name assigned to the node.
- **Compute** — The compute assigned to the node (for example, ″field_1.value + field_2.value″). The result of the compute will be stored in the literal of the node.

Depending on the node type, some of these properties may be null.

The Node Hierarchy

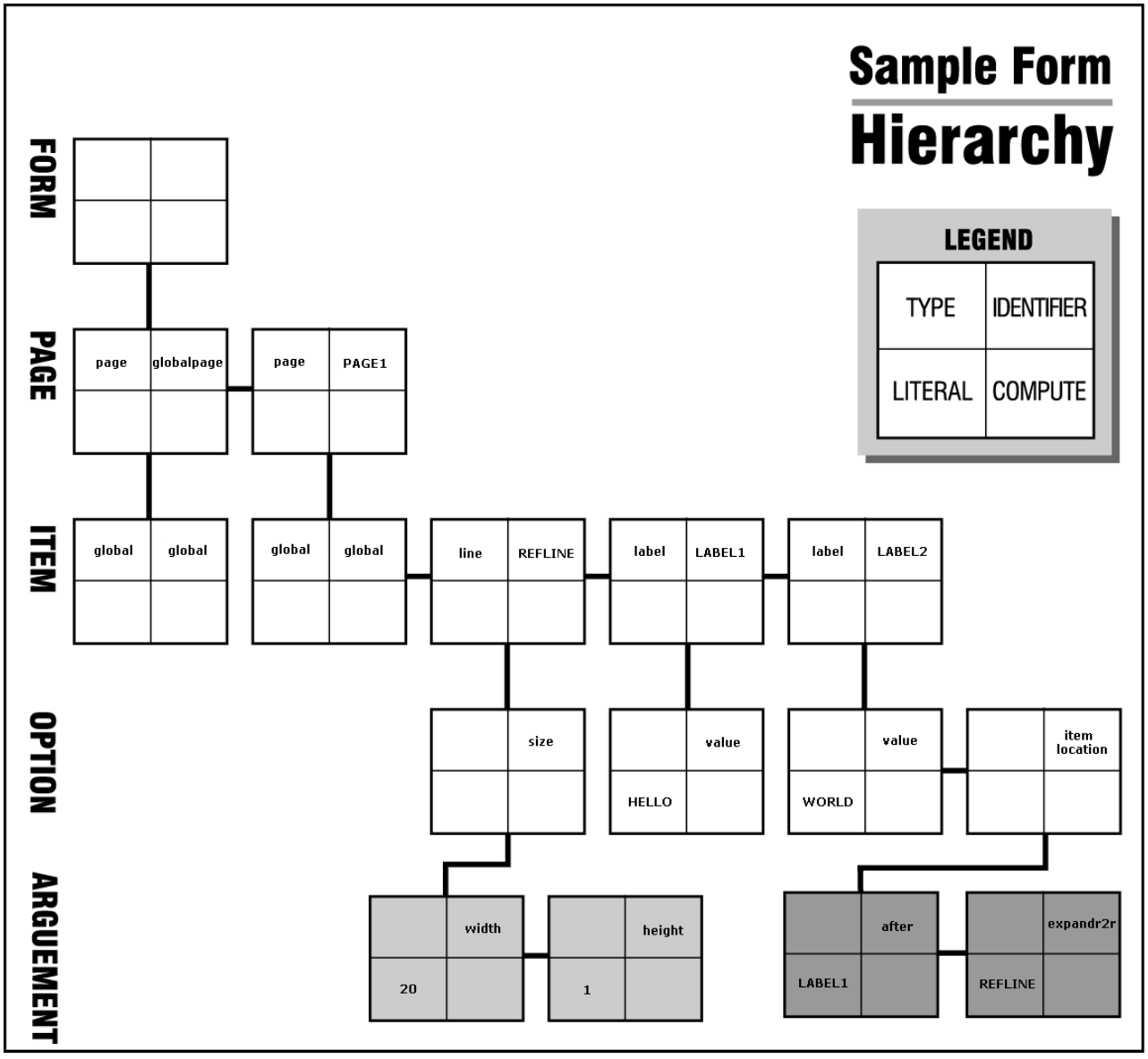
Every node is part of an overall hierarchy that describes the complete form. This hierarchy follows a standard tree structure, with the top of the tree being the top (or root) of the hierarchy.

The diagram on the following page illustrates the typical tree structure for a simple form.

The elements of the hierarchy, in descending order, are:

- **Form** — Each form has one form level node. This is the root node of the tree.
- **Page** — Each form contains pages, which are represented as children of the form node. Each form has at least two page nodes - one for the globalpage, which stores the global settings, and one for the first page of the form.
- **Item** — Each page contains items, which are represented as children of the page node. An item node is created for each item, including the global item which stores page settings.
- **Option** — Each item contains options, which are represented as children of the item node. An option node is created for each option.
- **Argument** — Options often contain further settings, or arguments, which are represented as children of the option node or as children of other argument

nodes. There may be more than one level of argument node created below an option node, depending on the option's settings. The easiest way to access a particular node in the hierarchy is to use a reference. References allow you to locate a specific node without first having to locate the parent of that node.



References

References allow you to identify a specific page, item, option, or argument by providing a "path" to that element. This means that you can access an element directly without having to locate any of its ancestors. The syntax of a reference follows this general pattern:

```
page.item.option[argument]
```

Each element of the reference is constructed as follows:

- **Page** and **Item** — Pages and items are identified by their scope identifiers (sid). For example, *Page1* or *Field1*.

- **Options** — Options are identified by their tag name. For example, *value* or *itemlocation*.
- **Arguments** — Arguments are identified by their tag name or a zero-based numeric index. Argument references are always enclosed in brackets. For example, *[1]* or *[message]*.

  Arguments can also have any depth. For example, you might have an argument that contains arguments. You can reference additional levels of depth by adding another bracketed reference. For example, to refer to the first argument in the first argument of the *printsettings* option, you could use either *[0][0]* or the tag names in brackets, such as *[pages][filter]*.

You can create references to any level of the node hierarchy. For example, the following table illustrates a number of references starting at different levels of the form:

Start At	Ref to Page	Ref to Item	Ref to Option	Ref to Argument
Page	Page1	Page1.Field1	Page1.Field1.format	Page1.Field1.format[message]
Item	—	Field1	Field1.format	Field1.format[message]
Option	—	—	format	format[message]
Argument	—	—	—	[message]

Dereferencing

When making a reference to an item node, there may be times when you do not know which node to reference because it depends on some action from the user of the form. Consider a situation in which a user selects a cell from a list. Because you don't know beforehand which cell the user will choose, it is not possible to explicitly reference the item node for the chosen cell. In such cases you would use *dereferencing* to retrieve the node indirectly.

Essentially, dereferencing allows you to make a dynamic reference that is evaluated at runtime. This is accomplished by placing the -> symbol to the right of the dynamic reference.

For example, consider a list item called *List1* that has three cells called *Cell1*, *Cell2*, *Cell3*. If you wanted to access the item node of the cell selected by the user, we would use the following reference string:

```
List1.value->
```

At runtime, the portion of the expression that is to the left of the dereference symbol is evaluated and replaced. If the user chose the second cell, *List1.value* would be evaluated and replaced with:

```
Cell2
```

As a result, the item node for *Cell2* would be returned.

In some cases, instead of accessing the item node of the chosen cell, you may want to access one of the cell's option nodes. Again, dereferencing is used. The reference string would be:

```
List1.value->value
```

As before, the above expression is evaluated at runtime. The expression to the left of the dereference symbol is evaluated and replaced, just as before. So if the second

cell was selected, *List1.value* would be evaluated as *Cell2*. This value is then concatenated with the expression to the right of the dereference symbol. This would produce:

```
Cell2.value
```

As a result, the option node for *Cell2.value* would be returned.

Note: Do not include any spaces before or after the dereference symbol (->).

Namespace in References

References that include options or arguments in any namespace other than XFDL normally require the inclusion of the namespace prefix in the reference. For example, if you were referencing ″myOption″ in the ″custom″ namespace, you would refer to that option as ″custom:myOption″ as shown:

```
page_1.myItem.custom:myOption
```

If you are referencing named arguments, you should also use the appropriate namespace. For example:

```
page_1.myItem.custom:myOption[custom:myArgument]
```

However, if you are referencing an argument by index number you do not need to worry about namespace. All arguments, regardless of namespace, are indexed in order. For example, if ″myOption″ contained two arguments, the first in the XFDL namespace and the second in the custom namespace, you would use the following reference for the second argument:

```
page_1.myItem.custom:myOption[1]
```

Note: Page and item references never require a namespace prefix because they are uniquely identified by their sid.

The null Namespace

In some cases, forms may have no default namespace or may have a default namespace that is explicitly set to an empty string. In these cases, you can use *null* as the prefix for the empty namespace. For example, the following field declares a default namespace that is empty:

```
<page sid="Page1">
   <field sid="myField" xmlns="">
      <value>Test Value</value>
   </field>
</page>
```

In this case, to reference the value of the field, you would use the null prefix as shown:

```
Page1.null:myField.null:value
```

Advanced Information about the Node Structure

When an XFDL form is stored in memory, it exists as a series of nodes that are linked in a tree structure. As described in “The Node Hierarchy” on page 7, the tree structure follows this hierarchy: form, page, item, option, and argument.

Within a single branch of the tree, all elements of the same level are treated as siblings, each of which has a common parent, and each of which may have its own children.

The following example illustrates the node structure of a simple form, and gives a top-down description of the node structure.

A Sample Hierarchy

The following XFDL code creates the node hierarchy shown in . The result is a simple form that contains three items (a line and two labels).

```
<?xml version = "1.0"?>
<XFDL xmlns="http://www.ibm.com/xmlns/prod/XFDL/7.0"
   xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0">
   <globalpage sid="global">
      <global sid="global"></global>
   </globalpage>
   <page sid = "PAGE1">
      <global sid="global"></global>

      <line sid = "REFLINE">
         <size>
            <width>20</width>
            <height>0</height>
         </size>
      </line>
      <label sid = "LABEL1">
         <value>Hello</value>
      </label>
      <label sid = "LABEL2">
         <value>World</value>
         <itemlocation>
            <after>LABEL1</ae>
            <expandr2r>REFLINE</expandr2r>
         </itemlocation>
      </label>
   </page>
</XFDL>
```

The Sample Tree Structure

Each tree begins with the form, or root, node. This node contains no information - it simply represents the starting point of the tree structure.

Below the form node are the page nodes. In the previous example, there are two page nodes: ″global″ and ″PAGE1″. The ″global″ page node stores any global settings that apply to the form while ″PAGE1″ stores the contents of the first form page. Any additional pages would also be stored as children of the form node.

Below each page node are the item nodes. As illustrated in the previous example, the first item node for any page is always the ″global″ item. The ″global″ item stores any page settings that are applied to the items in that page. Each additional item in the page is stored as a sibling of the global item.

Note: The ″global″ page node will always have one child: the global item. This global item will always store the XFDL version number used to create the form, and is also used to store any global settings that are applied to the form.

Below each item node are the option nodes. Each option node represents an option setting for that item, such as a background color or font setting.

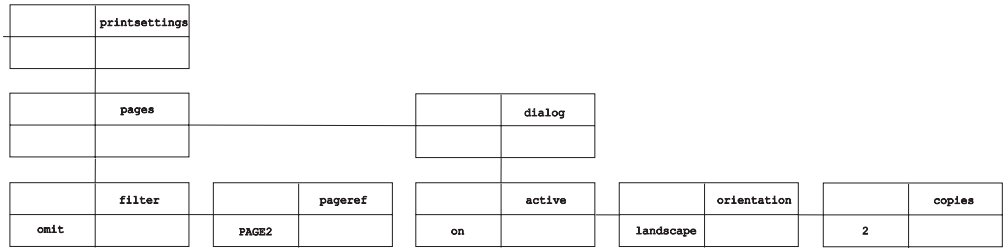
Below each option node are the argument nodes. These nodes contain the settings for the parent option. For example, the background color might be set to ″blue″.

There can be an infinite number and depth of these nodes, depending upon the number and depth of the settings for that option.

For instance, in the sample form, the *size* node for "REFLINE" has two argument nodes: one for the width and one for the height. In contrast, the *printsettings* option can have multiple argument nodes which themselves have argument nodes as children. The following is an example of the node structure of the *printsettings* option:

```
<printsettings>
   <pages>                                  argument node level I
      <filter>omit</filter>                 argument node level II
      <pageref>page2</pageref>              argument node level II
   </pages>
   <dialog>                                 argument node level I
      <active>on</active>                   argument node level II
      <orientation>landscape<orientation>   argument node level II
      <copies>2</copies>                    argument node level II
   </dialog>
</printsettings>
```

printsettings Node Structure



Thus, in storing the *printsettings* option, two levels of argument nodes are created. The first level describes the number of array elements in the option (two). The second level gives the arguments for each element.

Due to their potential complexity, pay careful attention to the mapping of argument nodes.

Note: In cases where an option has multiple elements in an array (for example, *printsettings*), there will be a single option node, but a separate argument node for each element in the array.

Node Properties

There are several levels of nodes in an XFDL form: form (or root), page, item, option, and argument (which can have an infinite number of levels). Each node has four properties: literal, type, identifier, and compute. A node does not necessarily contain information for every property.

For example, a page node can never have values for the compute or literal properties. And while a value for the user data property is optional, a page node must always have values for the type and identifier properties.

The following table illustrates what properties may be in use for each node level.

	Node Property			
Level	**Literal**	**Type**	**Identifier**	**Compute**
Form	no	no	no	no
Page	no	always	always	no
Item	no	always	always	no
Option	yes	no	always	yes
Argument (at any level)	yes	no	yes	yes

yes — node can have that property
always — node always has that property
no — node cannot have that property

Introduction to the Form Library

The Form Library is a collection of functions for developing applications that manipulate XFDL forms. Using the functions in the Form Library, your applications can:

- Read and write forms.
- Retrieve information contained in a form's elements.
- Assign information to the elements of a form.
- Create new elements within a form.
- Remove elements from a form.
- Extract images or enclosures from a form.
- Verify digital signatures.

Essentially, an XFDL form may be thought of as a structured data type, with the API as the means for accessing this data structure.

Getting Started with the Form Library

This section provides a detailed tutorial to help you understand how to use the Form Library. By working through the tutorial, you will perform all of the steps involved in creating a simple application that uses the API functions, including:

- Initializing the Form Library.
- Reading a form into your application.
- Setting and retrieving form data.
- Removing a form from memory.

The sample application in this tutorial reads an input form called CalculateAge.xfd into memory. It retrieves the user's birth day, month, and year as well as the current date from the form. It then places these values into hidden fields in the form. This triggers the form to compute the user's age and display the result. When complete, the application saves the changes made to calculateAge.xfd as a new form called Output.xfd.

Note: The sample application described in this tutorial is included with the API and can be found in the folder: <*API Program Folder*>\Samples\Msc32\Form\Demo\Calculate_Age\

The tutorial describes the following tasks:

Note: Before you can build applications using the Form Library, you must install the API and set up your development environment. Refer to the *IBM Workplace Forms Server - API Installation and Setup Guide* for more information.

Setting Up Your Application

1. A set of template project files for applications have been provided for you. Copy the Form template files into the directory in which you will be building your application.
   - Files that are compatible with the Visual C++ development environment can be found in: <*API Program folder*>\Samples\Msc32\Form\Template.

- If you prefer to set up the application development environment yourself, follow the procedure outlined in "Appendix A: Setting Up Your Development Environment" on page 205.

2. Create a new .c source file named calculateAge.c and include any necessary header files. For every application that uses the Form Library, you must include pe_api.h. In calculateAge.c:

```
#include "pe_api.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

Note: You must always include pe_api.h in any source file that uses the API.

3. Declare all the functions implemented in calculateAge.c.
   - The initialize function initializes the API.

```
r_short initialize(
#ifdef PROTO
   void
#endif
);
```

   - The **loadForm** function calls the API function **UFLReadForm** that will read a specified form into memory.

```
r_short loadForm(
#ifdef PROTO
   formNodeP *form
#endif
);
```

   - The function **getBirthDate** uses the API function **UFLGetLiteralByRefEx** to retrieve date information from the form loaded in memory.

```
r_short getBirthDate(
#ifdef PROTO
   formNodeP form,
   int *birYear,
   int *birMonth,
   int *birDay
#endif
);
```

   - The **saveForm** function saves the form to disk by calling the API function **UFLWriteForm**.

```
r_short saveForm(
#ifdef PROTO
   formNodeP form
#endif
);
```

   - The **reportError** function reports errors by printing the error codes associated with a particular error.

```
void reportError(
#ifdef PROTO
   char *theMessage,
   r_short theCode
#endif
);
```

4. Create your **main** function and declare any variables needed for your application. The **main** function in calculateAge.c will declare variables for the birth day, birth month, and birth year. It will also be responsible for calling the functions declared previously. Finally, **main** is responsible for freeing any memory used by the application by making a call to the function **UFLDestroy**.

```
   #ifdef WINDOWS
   #ifndef OLD_STYLE_PARAMS
      int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance, LPSTR
      lpszCmdParam, int nCmdShow)
   #else
      int PASCAL WinMain(hInstance, hPrevInstance,lpszCmdParam, nCmdShow)
      HANDLE hInstance;
      HANDLE hPrevInstance;
      LPSTR lpszCmdParam;
      int nCmdShow;
   #endif

   #else
   #ifndef OLD_STYLE_PARAMS
      int main(int argc, char **argv)
   #else
      int main(argc, argv)
      int argc;
      char **argv;
   #endif
   #endif
   {
   formNodeP theForm;
   int birthYear;
   int birthMonth;
   int birthDay;
   #ifndef WINDOWS
      void *hInstance = NULL;
   #else
   #ifndef MSC
   #ifndef __FLAT__
      _InitEasyWin();
   #endif
   #endif
   #endif

/* First, we call a function that will initialize the special memory and
      error handling. */

      if (initialize((void *)hInstance) != OK)
      {
         reportError("Error in initialize function.\n", 0);
         exit (1);
      }

/* Next, we call a function that will load the form into memory. */

      if (loadForm(&theForm) != OK)
      {
         reportError("Error in loadForm function.\n", 0);
         exit (1);
      }

/* This function retrieves the birth year, month and day values (as
      integers) from the original form. */

      if (getBirthDate(theForm, &birthYear, &birthMonth, &birthDay) != OK)
      {
         reportError("Error in the getBirthDate function.\n", 0);
         exit (1);
      }

/* This function writes the birth year, month, and day into hidden fields
      in the original form whose values are the arguments for the compute for
      PAGE1.SHOWAGE.value */

      if (setBirthDate(theForm, birthYear, birthMonth, birthDay) != OK)
```

```
      {
         reportError("Error in the setBirthDate function.\n", 0);
         exit (1);
      }

/* This function saves the processed form in the current directory as
      'output.xfd'. */

      if (saveForm(theForm) != OK)
      {
         reportError("Error in the saveForm function.\n", 0);
         exit (1);
      }

/* Now that we are done with the form, we can free the memory by calling
      UFLDestroy. The parameter, 'theForm', is a pointer to the root node of
      the form. This causes the root node and all of its children (the
      complete form) to be deleted from memory. */

      UFLDestroy(theForm);
      return(OK);
   }
```

Note: For more information about the API function **UFLDestroy** refer to "UFLDestroy" on page 48.

Initializing the Form Library

All applications that use the API functions must initialize the Form Library to ensure correct error and memory handling behavior. The sample application does this in a separate method called **initialize**. In turn, **initialize** calls the Form Library function **IFSInitialize** and passes it the name of the current program.

Define the **initialize** function to call the function **IFSInitialize**. **IFSInitialize** initializes the API environment.

```
#ifndef OLD_STYLE_PARAMS
   r_short initialize(void *theInstance)
#else
r_short initialize(theInstance)
   void *theInstance;
#endif
{
r_error error;

/* Call IFSInitialize. The parameters are:
   1. SampleAp : the name of the application being run.
   2. 1.0.0 : the version of the application being run.
   3. 2.6.0 : the version of the API to use by default.
   An error code will be returned if there is a problem. */

   error = IFSInitialize("SampleAp", "1.0.0", "2.6.0");
   if (error != OK)
   {
      reportError("IFSInitialize error %hd.\n", (r_short)error);
      return(NOTOK);
   }
   return(OK);
}
```

Note: For detailed information about the **IFSInitialize** function, including a description of its parameters, refer to "IFSInitialize" on page 124.

Loading a Form

Before your program can begin working with a form, you must load it into memory. CalculateAge does this by defining a **loadForm** function to handle these tasks.

Call **UFLReadForm** within the implementation of your **loadForm** function to read in the form, calculateAge.xfd.

```
#ifndef OLD_STYLE_PARAMS
   r_short loadForm(formNodeP *form)
#else
   r_short loadForm(form)
   formNodeP *form;
#endif
{

/* Call UFLReadForm. The parameters are:
   1. calculateAge.xfd : indicates the file on the local drive to read the
      form from.
   2. 0 : because we do not need the #include lines to be resolved.
   The UFLReadForm will return a pointer to the root node of the new form
   structure once it is loaded into memory. */

   if ((*form = UFLReadForm("calculateAge.xfd", 0)) == NULL)
   {
      reportError("Could not load form.\n", 0);
      return(NOTOK);
   }
   return(OK);
}
```

Note: For more information about the **UFLReadForm** function, refer to "UFLReadForm" on page 160.

Retrieving A Value from a Form

Once you have set up and initialized your application with the API and loaded a form into memory, your application is ready to start working with the form. The following code uses **UFLGetLiteralByRefEx** to get a specific value from the form:

Retrieve the birth date indicated on the form by implementing the **getBirthDate** function. **getBirthDate** will make three calls to **UFLGetLiteralByRefEx**.

```
#ifndef OLD_STYLE_PARAMS
   r_short getBirthDate(formNodeP form, int *birYear, *birMonth,
   int *birDay)
#else
   r_short getBirthDate(form, birYear, birMonth, birDay)
   formNodeP form;
   int *birYear;
   int *birMonth;
   int *birDay;
#endif
{
r_charP temp=NULL;
r_short error;

/* Call UFLGetLiteralByRefEx to get the literal information for the
   'PAGE1.BIRTHYEAR.value' node. An error code will be returned if there
   is a problem. */

   error = UFLGetLiteralByRefEx(form, NULL, "PAGE1.BIRTHYEAR.value", 0,
```

```
      NULL, NULL, &temp);
   if (error != OK)
   {
      reportError("UFLGetLiteralByRefEx error %hd.\n", error);
      return(NOTOK);
   }

   /* If a literal value was returned, convert it into an integer value.
   Otherwise, indicate that no value was entered into the field, and exit
   the program. */

   if (temp != NULL)
      *birYear = atoi((char *)temp);
   else
   {
      reportError("The birth year was not entered.\n",0 );
      return(NOTOK);
   }

   /* Call UFLGetLiteralByRefEx to get the literal information for the
   'PAGE1.BIRTHMONTH.value' node. An error code will be returned if there
   is a problem. */

   error = UFLGetLiteralByRefEx(form, NULL, "PAGE1.BIRTHMONTH.value",
      0, NULL, NULL, &temp);
   if (error != OK)
   {
      reportError("UFLGetLiteralByRefEx error %hd.\n", error);
      return(NOTOK);
   }

/* If a literal value was returned, convert it into an integer value.
   Otherwise, indicate that no value was entered into the field, and exit
   the program. */

   if (temp != NULL)
      *birMonth = atoi((char *)temp);
   else
   {
      reportError("The birth month was not entered.\n",0 );
      return(NOTOK);
   }

/* Call UFLGetLiteralByRefEx to get the literal information for the
   'PAGE1.BIRTHDAY.value' node. An error code will be returned if there is
   a problem. */

   error = UFLGetLiteralByRefEx(form, NULL, "PAGE1.BIRTHDAY.value", 0,
      NULL, NULL, &temp);
   if (error != OK)
   {
      reportError("UFLGetLiteralByRefEx error %hd.\n", error);
      return(NOTOK);
   }

/* If a literal value was returned, convert it into an integer value.
   Otherwise, indicate that no value was entered into the field, and exit
   the program. */

   if (temp != NULL)
      *birDay = atoi((char *)temp);
   else
   {
      reportError("The birth day was not entered.\n", 0);
```

```
      return(NOTOK);
   }
   return(OK);
}
```

Note: For detailed information about the **UFLGetLiteralByRefEx** method, including a description of its parameters, refer to "UFLGetLiteralByRefEx" on page 66.

Setting a Value in a Form

Once a form is loaded into memory, a developer can set the values associated with any of the item or option nodes located in the form by calling **UFLSetLiteralByRefEx**.

Change the values of hidden fields in the original form. These hidden fields are arguments for a compute in the SHOWAGE label that calculates the user's age.

```
#ifndef OLD_STYLE_PARAMS
   r_short setBirthDate(formNodeP form, int birYear, int birMonth,
   int birDay)
#else
r_short setBirthDate(form, birYear, birMonth, birDay)
   formNodeP form;
   int birYear;
   int birMonth;
   int birDay;
#endif
{
r_short error;
char *temp[100];

/* Convert the int to a r_charP and set the value of PAGE1.HIDDENYEAR.value */

   sprintf(temp, "%d", birYear);

/* Call UFLSetLiteralByRefEx. The parameters are:
   1. form : the formNodeP to use as the starting point for the search.
   2. XFDL : the scheme used to write the reference parameter.
   3. PAGE1.HIDDENYEAR.value : the reference to the node.
   3. 0 : must be zero.
   4. NULL : use the ANSI character set.
   5. NULL : do not use a namespace node.
   6. temp : the value to assign to the literal.
   An error is returned if there is a problem. */

   error = UFLSetLiteralByRefEx(form, NULL, "PAGE1.HIDDENYEAR.value",
      0, NULL, NULL, (r_charP)temp);
   if(error!= OK)
   {
      reportError("UFLSetLiteralByRefEx error%hd.\n", error);
      return(NOTOK);
   }

/* Convert the int to an r_charP and set the value of
   PAGE1.HIDDENMONTH.value*/

   sprintf(temp, "%d", birMonth);
   error = UFLSetLiteralByRefEx(form, NULL, "PAGE1.HIDDENMONTH.value",
      0, NULL, NULL, (r_charP)temp);
   if(error!= OK)
   {
      reportError("UFLSetLiteralByRefEx error%hd.\n", error);
      return(NOTOK);
```

```
    }

/* Convert the int to an r_charP and set the value of PAGE1.HIDDENDAY.value */

    sprintf(temp, "%d", birDay);
    error = UFLSetLiteralByRefEx(form, NULL, "PAGE1.HIDDENDAY.value", 0,
       NULL, NULL, (r_charP)temp);
    if(error!= OK)
    {
       reportError("UFLSetLiteralByRefEx error%hd.\n", error);
       return(NOTOK);
    }
    return(OK);
}
```

Note: For detailed information about **UFLSetLiteralByRefEx**, including a description of its parameters, refer to "UFLSetLiteralByRefEx" on page 102.

Writing a Form to Disk

Once you have finished making the desired changes to the form, you should save it to disk. If you want to retain the original form (calculateAge.xfd), you should save the modified form under a new name. This program saves the modified form as Output.xfd.

The following example implements the function saveForm. This function calls the API function **UFLWriteForm** that writes a form to disk.

```
#ifndef OLD_STYLE_PARAMS
   r_short saveForm(formNodeP form)
#else
   r_short saveForm(form)
   formNodeP form;
#endif
{
r_short error;

/* Call UFLWriteForm. The parameters are:
   1. form : a pointer to the root node of the form.
   2. output.xfd : the filename you want to use (you could also use a path
      here).
   3. NULL : since we do not want to set a triggeritem.
   4. 0 : since we do not want to allow the transmit options to work.
   An error code is returned if there is a problem. */

   error = UFLWriteForm(form, "output.xfd", NULL, 0);
   if (error != OK)
   {
      reportError("UFLWriteForm error %hd.\n", error);
      return(NOTOK);
   }
   return(OK);
}
```

Note: For detailed information about **UFLWriteForm**, including a description of its parameters, refer to "UFLWriteForm" on page 118.

Closing a Form

Next, you must free the memory used by the form itself. This is the last operation in the main function of the program.

Free the memory by using **UFLDestroy**, which destroys the root node of the form and all of its children:

```
/* Now that we are done with the form, we can free the memory by
   calling UFLDestroy. The parameter, 'theForm', is a pointer to
   the root node of the form. This causes the root node and all
   of its children (the complete form) to be deleted from memory. */
   UFLDestroy(theForm);
   return(OK);
}
```

Note: For detailed information about **UFLDestroy**, including a description of its parameters, refer to "UFLDestroy" on page 48.

Reporting Errors

The final step in developing your application is to implement an error reporting system such as the **reportError** function.

```
#ifndef OLD_STYLE_PARAMS
      void reportError(char *theMessage, r_short theCode)
#else
      void reportError(theMessage, theCode)
      char *theMessage;
      r_short theCode;
#endif
{
static char theRealMessage[1024];
      sprintf(theRealMessage, theMessage, theCode);
#ifdef WINDOWS
      MessageBox(NULL, theRealMessage, "An error occurred in SampleAp",
         MB_OK);
#else
      fprintf(stderr, theRealMessage);
#endif
}
```

Compiling Your Application

Once you have generated the source files for your application, you must compile the source code.

- Use an appropriate compiler that is supported by this API to compile your files. Refer to the *IBM Workplace Forms Server - API Installation and Setup Guide* for more information about compatible development environments.
- Before building your application you should have a source file that represents your application. After compiling the file you will have a file an executable (or class file) with the same name.
- The details of compiling your source code are not included in this manual. Consult your development environment's documentation for specific information on how to use your compiler.
- Make sure that the compiler uses the -I option when searching the directory containing the API include files.

Testing your Application

Use the sample form that accompanies the API to test the Calculate Age application.

1. Copy the file calculateAge.xfd to the folder containg your application. The file is located in the following folder:

<*API Program folder*>\Samples\Msc32\Form\Demo\Calculate_Age\

2. Open the form in the Viewer to see the original settings.
3. Run the application that you have just created.
4. A new file will be created called Output.xfd.

Note: To view the forms provided with this API, you must have a licensed or evaluation copy of the IBM Workplace Forms Viewer installed.

Distributing Applications That Use the Form Library

If you distribute applications that use the Form Library, you will also need to distribute a number of API files. Refer to the *IBM Workplace Forms Server - API Installation and Setup Guide* for information about distributing applications that use the Form Library.

Summary

By working through this section you have successfully built the Calculate Age application. In the process, you have learned how to initialize, compile, and test form applications using the following methods from the Form Library:

- IFSInitialize
- UFLReadForm
- UFLGetLiteralByRefEx
- UFLSetLiteralByRefEx
- UFLWriteForm
- UFLDestroy

The source code for the Calculate Age application is included with this API and can be found in the following folder:

<*API Program folder*>\Samples\Msc32\Form\Demo\Calculate_Age\

For a longer example using the Form Library of methods refer to the other sample application installed with the API. The source files for this application are located in the following folder:

<*API Program folder*>\Samples\Msc32\Form\Demo\Sample_Application\

To view the forms provided with the sample applications, you must have a copy of the Viewer installed.

Note: The sample files provided are compatible with the Visual C++ development environment. For more information about compatible development environments for the API refer to the *IBM Workplace Forms Server - API Installation and Setup Guide.*

Form Library Quick Reference Guide

This section provides detailed information about the Form Library. The available functions are divided into the following categories:

Within each section, the functions are presented alphabetically.

Form Library Functions

The Form Library includes the following functions:

Function Type	Description	Functions
Certificate	The **Certificate** function gets information from digital certificates.	Certificate_GetBlob Certificate_GetDataByPath Certificate_GetIssuer
formNodeP	The **formNodeP** functions create and populate nodes and free memory.	UFLAddNamespace UFLCreateCell UFLDeleteSignature UFLDereferenceEx UFLDestroy UFLDuplicate UFLEncloseFile UFLEncloseInstance UFLExtractFile UFLExtractInstance UFLExtractXFormsModel

Function Type	Description	Functions
formNodeP (continued)	The **formNodeP** functions create and populate nodes and free memory.	UFLGetAttribute UFLGetAttributeList UFLGetCertificateList UFLGetChildren UFLGetFormVersion UFLGetInfoEx UFLGetLiteralEx UFLGetLiteralByRefEx UFLGetLocalName UFLGetNamespaceURI UFLGetNamespaceURIFromPrefix UFLGetNext UFLGetNodeType UFLGetParent UFLGetPrefix UFLGetPrefixFromNamespaceURI UFLGetPrevious UFLGetReferenceEx UFLGetSecurityEngineName UFLGetSigLockCount UFLGetSignature UFLGetSignatureVerificationStatus UFLIsSigned UFLIsXFDL UFLRemoveEnclosure UFLReplaceXFormsModel UFLSetActiveForComputationalSystem UFLSetAttribute UFLSetFormula UFLSetLiteralEx

Function Type	Description	Functions
formNodeP (continued)	The **formNodeP** functions create and populate nodes and free memory.	UFLSetLiteralByRefEx UFLSignForm UFLValidateHMACWithSecret UFLValidateHMACWithHashedSecret UFLVerifyAllSignatures UFLVerifySignature UFLWriteForm UFLXMLModelUpdate
Hash	The **Hash**functions hash data.	Hash_Hash
Initialization	The **Initialization** functions initialize the API.	IFSGetGlobalIFX IFSInitialize
LocalizationManager	The **LocalizationManager** functions control which locale (language) the API uses.	LocalizationManager_GetCurrentThread Locale LocalizationManager_GetDefaultLocale LocalizationManager_SetCurrentThread Locale LocalizationManager_SetDefaultLocale
SecurityManager	The **SecurityManager** functions provide hashing algorithms.	SecurityManager_GetSingleton SecurityManager_LookupHashAlgorithm
Signature	The **Signature** functions get information from signatures.	Signature_GetDataByPath Signature_GetSigningCert
Utility	The **Utility** functions are used to perform a variety of tasks within the API.	MUGetThreadSafeFlag
XFDL	The **XFDL** functions create the root nodes of XFDL forms and handle administrative tasks related to the API.	UFLCreate UFLGetEngineCertificateList UFLIsDigitalSignaturesAvailable UFLReadForm

About the Function Descriptions

The functions in this reference guide are listed according to the functionality they provide and are described using the following format:

- **Description:** Provides a general description of what the function does.
- **Function:** Lists the function's signature and type of value returned (if any).
- **Parameters:** Lists and describes each parameter in detail.
- **Returns:** Indicates what value is returned by the function.

- **Notes:** Provides additional information to help you use the function.
- **Example:** Provides sample code that uses the function in question.

Using Signatures with the Form Library

Computed options often contain their current computed value. If this value is signed, it will not change, even if something in the form changes that would normally trigger the compute.

The literal value is stored as simple character data in the computed option, as shown below:

```
<field sid="FIELD1">
   <value compute="page1.nameField.value">Jane E. Smith</value>
</field>
```

The node structure for this *value* option is:

field	FIELD1
	value
Jane E. Smith	Page1.nameField.value

The Viewer sets this literal value when a form is signed, submitted, or saved (and discards any old value if necessary). When **UFLReadForm** is invoked, the current value (cval) is set and cannot be changed. Because a digitally signed formula never fires after being signed, the current value for the option is always the same - and therefore it is possible to reference the option and get the signed literal value.

The Certificate Functions

The **Certificate** functions allow you to work with *Certificate* objects.

- You must include the following header file in any .c source file that calls a Certificate function:

```
#include "Certificate.h"
```

Certificate_GetBlob

Description

This function extracts a binary long object (Blob). This Blob is a DER-encoded certificate.

Function

```
r_error Certificate_GetBlob(
   Certificate *theCertificate,
   SecurityUserStatusType *theStatus,
   r_byte **theBlob,
   r_long *blobSize);
```

Parameters

Expression	Type	Description
theCertificate	**Certificate***	A pointer to the certificate object.
theStatus	**SecurityUserStatusType***	A pointer that is set with the status of the operation. This will be one of the following: **SUSTATUS_OK** — The operation was successful. **SUSTATUS_CANCELLED** — the operation was cancelled by the user. **SUSTATUS_INPUT_REQUIRED** — the operation required user input, but could not receive it (for example, it was run on a server with no user).
theBlob	**r_byte****	A pointer that is set with the Blob.
blobSize	**r_long***	A pointer that is set with the length of the Blob.

Returns

OK on success or an error code on failure.

Example

The following function extracts the Blob from a certificate, checks the status to make sure the operation was successful, then returns the Bloband size as parameters.

```
   r_error extractBlob(Certificate *theCert, r_byte **theBlobPtr, r_long
*blobSizePtr)
   {
```

```
SecurityUserStatusType theStatus;

   /* Get the Blob from the certificate. */

   if ((Certificate_GetBlob(theCert, &theStatus, theBlobPtr, blobSizePtr)
      != OK)
   {
      fprintf(stderr, "Certificate_GetBlob call failed.\n");
      return(NOTOK);
   }

   /* Check the status to ensure the function worked correctly. */

   if (theStatus != SUSTATUS_OK)
   {
      fprintf(stderr, "GetBlob exited with wrong status.\n");
      return(NOTOK);
   }
   return(OK);
}
```

Certificate_GetDataByPath

Description

This function retrieves a piece of data from a certificate object.

Function

```
r_error Certificate_GetDataByPath(
   Certificate *theCertificate,
   r_charP thePath,
   r_boolean tagData,
   r_boolean *encoded,
   r_charP *theData);
```

Parameters

Expression	Type	Description
theCertificate	**Certificate***	The certificate object you want to query.
thePath	**r_charP**	The path to the data you want to retrieve. See the Notes section below for more information on data paths.
tagData	**r_boolean**	OK if the path should be prepended to the data, or NOTOK if not. If the path is prepended, an equals sign (=) is used as a separator. For example, suppose the path is ″Signing Cert: Issuer: CN″ and the data is ″IBM″. If OK, the path will be prepended, producing ″CN=IBM″. If NOTOK, the path will not be prepended, and the result will be ″IBM″.
encoded	**r_boolean***	OK if the return data is base 64 encoded, or NOTOK if not. The function returns binary data in base 64 encoding.
theData	**r_charP***	The data that the function locates. This is NULL if no data is found. Note that this string must be freed.

Notes

About Data Paths

Data paths describe the location of information within a certificate, just like file paths describe the location of files on a disk. You describe the path with a series of colon separated tags. Each tag represents either a piece of data, or an object that contains further pieces of data (just like directories can contain files and subdirectories).

For example, to retrieve the version of a certificate, you would use the following data path:

```
version
```

However, to retrieve the subject's common name, you first need to locate thesigning certificate, then the subject, then the common name within the subject, as follows:

```
SigningCert: Subject: CN
```

Some tags may contain more than one piece of information. For example, the issuer's organizational unit may contain a number of entries. You can either retrieve all of the entries as a comma separated list, or you can specify a specific entry by using a zero-based element number.

For example, the following path would retrieve a comma separated list:

```
Issuer: OU
```

While adding an element number of 0 would retrieve the first organizational unit in the list, as shown:

```
Issuer: OU: 0
```

Certificate Tags

The following table lists the tags available in a certificate object:

Tag	Description
Subject	The subjects distinguished name. This is an object that contains further information, as detailed in *Distinguished Name Tags*.
Issuer	The issuer's distinguished name. This is an object that contains further information, as detailed in *Distinguished Name Tags*.
IssuerCert	The issuer's certificate. This is an object that contains the complete list of certificate tags.
Engine	The security engine that generated the certificate. This is an object that contains further information, as detailed in *Security Engine Tags*.
Version	The certificate version.
BeginDate	The date on which the certificate became valid.
EndDate	The date on which the certificate expires.
Serial	The certificates serial number.
SignatureAlg	The signature algorithm used to sign the certificate.
PublicKey	The certificates public key.

Tag	Description
FriendlyName	The certificates friendly name.

Distinguished Name Tags

The following table lists the tags available in a distinguished name object:

Tag	Description
CN	The common name.
E	The e-mail address.
T	The title.
O	The organization.
OU	The organizational unit.
C	The country.
L	The locality.
ST	The state.
All	The entire distinguished name.

Security Engine Tags

The following table lists the tags available in the security engine object:

Tag	Description
Name	The name of the security engine used by the server.
Help	The help text for the security engine.
HashAlg	A hash algorithm supported by the security engine.

Returns

OK on success or an error code on failure.

Example

The following function uses **UFLDereferenceEx** to locate a signature button in the form. It then uses **UFLGetCertificateList** to get a list of valid certificates for that button. Next, the function cycles through the returned certificates, uses **Certificate_GetDataByPath** to get the common name for each certificate, and identifies the certificate with a common name of "Workplace Forms Server". Then the function uses **UFLSignForm** to sign the form with the server's certificate. Finally, the certificate and signature objects are released.

```
r_error serverSign(formNodeP form)
{
SecurityUserStatusType theStatus;
formNodeP buttonNode;
Certificate **certList;
Signature *theSignature;
r_charP signerCommonName;
r_boolean encodedResult;
r_long certCount;
r_long correctCert = -1;
r_error error;
```

```
r_long i;

   if ((buttonNode = UFLDereferenceEx(form, NULL, "PAGE1.SIGNBUTTON",
      0, UFL_ITEM_REFERENCE, NULL)) == NULL)
   {
      fprintf(stderr, "Could not find SIGNBUTTON node.\n");
      return(NOTOK);
   }

   if ((error = UFLGetCertificateList(buttonNode, NULL, &theStatus,
      &certList, &certCount)) != OK)
   {
      fprintf(stderr, "UFLGetCertificateList error %hd.\n", error);
      return(NOTOK);
   }

   /* Check the status, in case the process required user input. */
   if (theStatus != SUSTATUS_OK)

   {
      fprintf(stderr, "User input required to retrieve certificate list.
         /n");
      return(NOTOK);
   }

   for (i=0; i<certCount; i++)
   {
      if ((error = Certificate_GetDataByPath(certList[i],
         "SigningCert: Subject: CN", NOTOK, &encodedResult,
         &signerCommonName)) != OK)
      {
         fprintf(stderr, "Certificate_GetDataByPath error %hd./n",
            error);
         return(NOTOK);
      }

      if (cp_strcmp(signerCommonName, "Workplace Forms Server") == OK)
      {
         correctCert = i;
         cp_free(signerCommonName);
         break;
      }

      cp_free(signerCommonName);

   }

   if (correctCert == -1)
   {
      fprintf(stderr, "Could not locate required certificate.\n");
      return(NOTOK);
   }

   if ((error = UFLSignForm(buttonNode, certList[correctCert], NULL,
      &theStatus, &theSignature)) != OK)
   {
      fprintf(stderr, "UFLSignForm error %hd.\n", error);
      return(NOTOK);
   }

   /* Check the status in case the process required user input. */
   if (theStatus != SUSTATUS_OK)
   {
      fprintf(stderr, "User input required to sign form.\n");
      return(NOTOK);
   }
```

```
   /* Release each certificate object in the array */

   for(i=0; i<certCount; i++)
   {
      IFSObject_ReleaseRef((IFSObject*)certList[i]);
   }

   /* Free the array */

   pe_free(certList);

   /* Release the signature object */

   IFSObject_ReleaseRef((IFSObject*)theSignature);

   return(OK);
}
```

Certificate_GetIssuer

Description

This function extracts the issuer certificate from the certificate provided.

Function

```
r_error Certificate_GetIssuer(
   Certificate *theCertificate,
   SecurityUserStatusType *theStatus,
   Certificate **issuerCert);
```

Parameters

Expression	Type	Description
theCertificate	**Certificate***	A pointer to the certificate object from which you want to extract the issuer certificate.
theStatus	**SecurityUserStatusType***	A pointer that is set with the status of the operation. This will be one of the following: **SUSTATUS_OK** — The operation was successful. **SUSTATUS_CANCELLED** — the operation was cancelled by the user. **SUSTATUS_INPUT_REQUIRED** — the operation required user input, but could not receive it (for example, it was run on a server with no user).
issuerCert	**Certificate****	A pointer that is set with the issuers certificate.

Returns

OK on success or an error code on failure.

Example

The following example gets the signing certificate from a signature object, then iterates through the certificate issuers until it reaches the end of the chain. During the iteration, each certificate is passed to a function that processes them.

```
r_error processCertChain(Signature *theSig)
{
Certificate *theCert, *issuerCert;
SecurityUserStatusType theStatus;

/* Get the signing certificate from the signature. */

   if (Signature_GetSigningCert(theSig, &theCert) != OK)
   {
      fprintf(stderr, "Could not get signing certificate.\n");
      return(NOTOK);
   }

   /* Loop through the certificate chain, passing each certificate to the
      ProcessCert function. The loop ends when the issuer certificate is
      NULL. */

while (theCert != NULL)
      {

         /* Pass the certificate to the processCert function.  Note that
            this is not an API function, but rather a function you would
            write to process the certificate in some way. */

         if (ProcessCert(theCert) != OK)
         {
            fprintf(stderr, "Could not process certificate.\n");
            return(NOTOK);
         }

         /* Get the issuer certificate from theCert. */

         if (Certificate_GetIssuer(theCert, &theStatus, &issuerCert) != OK)
         {
            fprintf(stderr, "Could not get issuer certificate.\n");
            return(NOTOK);
         }

         /* Check to ensure the function exited with the correct status. */

         if (theStatus != SUSTATUS_OK)
         {
            fprintf(stderr, "GetIssuer exited with wrong status.\n");
            return(NOTOK);
         }

         /* Free theCert object. */

         IFSObject_ReleaseRef(theCert);

         /* Assign theCert to equal the issuerCert for next iteration of the
            loop. */

         theCert = issuerCert;
      }
      return(OK);
   }
```

The formNodeP Functions

The **formNodeP** functions apply to particular instances of a form and the items in that form.

- Each node in a form, regardless of its level in the node hierarchy, is represented by a **formNodeP** structure. For more information about the node structure of XFDL forms refer to "Overview of the Form Structure" on page 7.
- To use the **formNodeP** functions in an application, include the following header file as the first include in any .c files which call **formNodeP** functions:

```
#include "xfdllib.h"
```

formNodeP Constants

The following table lists the constants that are used by the formNodeP functions along with a short description of each constant:

Named Constants	Description
UFL_DS_CERTEXPIRED	The certificate has expired.
UFL_DS_CERTNOTFOUND	The certificate was not found.
UFL_DS_CERTNOTTRUSTED	The certificate is no longer trusted.
UFL_DS_CERTREVOKED	The certificate has been revoked.
UFL_DS_F2MATCHSIGNER	The name in the form did not match the name in the signature.
UFL_DS_HASHCOMPFAILED	The hash of the document did not match the hash in the signature.
UFL_DS_ISSUERNOTFOUND	The issuer could not be found.
UFL_DS_ISSUERSIGFAILED	The verification of the issuer's certificate failed.
UFL_DS_SIGNATUREALTERED	The signature has been altered.
UFL_DS_UNEXPECTED	Generic error.

UFLAddNamespace

Description

This function adds a namespace declaration to the node it is passed. Each namespace is defined in the form by a namespace declaration, as shown:

```
xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
xmlns:custom="http://www.ibm.com/xmlns/prod/XFDL/Custom"
```

Each namespace declaration defines both a prefix and a URI for the namespace. In this sample, the prefix for the XFDL namespace is *xfdl* and the URI is *http://www.ibm.com/xmlns/prod/XFDL/7.0*.

Tags within the form are assigned specific namespaces by using the defined prefix. For example, to declare that an option was in the custom namespace you would use the prefix *custom* as shown:

```
<field sid="testField">
   <custom:custom_option>value</custom:custom_option>
</field>
```

Function

```
r_error UFLAddNamespace(
   formNodeP theNode,
   r_charP theURI,
   r_charP thePrefix);
```

Parameters

Expression	Type	Description
theNode	**formNodeP**	Any node in the form.
theURI	**r_charP**	The namespace URI. For example: `http://www.ibm.com/xmlns/prod/XFDL/7.0`
thePrefix	**r_charP**	The prefix for the namespace. For example, *xfdl*.

Returns

OK on success or an error code on failure.

Example

The following function uses **UFLAddNamespace** to add a custom namespace to a form. It then locates the global item in the global page and adds a custom option to that item which marks the status of the form as ″Processed″.

```
r_error addStatus(formNodeP theForm)
{
XFDL theXFDL;
formNodeP tempNode;
r_error error;

   /* Add the custom namespace to the form. */

   if ((error = UFLAddNamespace(theForm,
      "http://www.ibm.com/xmlns/prod/XFDL/Custom", "custom")) != OK)
   {
      fprintf(stderr, "Could not create custom namespace.\n");
      return(error);
   }

   /* Locate the global item in the global page so we can add a global
      option. */

   if ((tempNode = UFLDereferenceEx(theForm, NULL, "global.global", 0,
      UFL_ITEM_REFERENCE, NULL)) == NULL)
   {
      fprintf(stderr, "Could not locate global item node.\n");
      return(NOTOK);
   }

   /* Create a new option node as a child of the global item.  This node
      is created in the custom namespace, called "Status", and given a
      value of "Processed". */

   if ((tempNode = UFLCreate(tempNode, UFL_APPEND_CHILD, NULL,
      "Processed", NULL, "custom:Status") == NULL)
   {
      fprintf(stderr, "Could not create option node.\n");
```

```
      return(NOTOK);
   }
   return(OK);
}
```

UFLCheckValidFormats

Description

This function checks the format of all items in the form and returns the number of items whose format is invalid. You can also set the function to create a list of the invalid items.

This function does not support XForms nodes.

Function

```
r_error UFLCheckValidFormats(
   formNodeP theFormNode,
   r_boolean *returnPtr)
r_long *returnSizePtr;
```

Parameters

Expression	Type	Description
theFormNode	**formNodeP**	The node to check.
returnPtr	**r_boolean***	A pointer that is set to store an array of nodes that represent items with invalid formats.
returnSizePtr	**r_long***	A pointer that is set with the size of the return.

Returns

OK on success or an error code on failure.

Example

The following function uses recursion to traverse the entire node structure and check whether the nodes have the correct format. This function assumes that you are passing in the root node of the form.

```
r_error checkFormats(formNodeP theForm,);
{
formNodeP *invalidItems;
r_long i, invalidItemsSize;
r_error error;

   UFLCheckValidFormats(theForm, &invalidItems, &invalidItemsSize);
   if (invalidItemsSize == 0)

   {
      printf("All items in the form have correctly formatted values. \n");
   }

   else
   {
      printf("One or more of the items in the form have incorrectly formated
         values.\n");
      for (i=0' i<invalidItemsSize; i++)
      {
         UFLGetReferenceEx(invalidItems[i], NULL, NULL, NULL, OK, &theReference);
```

```
            UFLGetLiteralByRefEx(invalidItems[i], NULL, "value", 0, NULL, NULL, &theValue);
            printf("The item %s has an invalid value: %s\n", theReference, theValue);
            cp_free(theReference);
            cp_free(theValue);         }
         }
      return(OK);
   }
```

UFLCreateCell

Description

Use this function to create a new cell item for a *combobox*, *list*, or *popup*. **UFLCreateCell** adds one new cell to a specific *group* on a specific page in the form. Note that this function can only assign a name to the new cell; it cannot set the cell's *value*. To set the value of a cell, you must use the **UFLSetLiteralByRefEx** function.

Function

```
r_error UFLCreateCell(
   formNodeP thePageNode
   r_charP theCellName
   r_charP theGroupName
   formNodeP *theCellPointer);
```

Parameters

Expression	Type	Description
thePageNode	**formNodeP**	A page level node. The cell is created in this page. Note that this cannot be the global page.
theCellName	**r_charP**	The name of the new cell being created.
theGroupName	**r_charP**	The name of the *group* option to which the new cell will be added.
theCellPointer	**formNodeP***	The pointer to the address where the new cell will be stored.

Returns

OK on success or an error code on failure.

Example

This sample code makes two calls to the **UFLCreateCell** function to add two new cells to the same group:

```
/* Create a cell with name green and of group POPUP_GROUP_COLORS. Since
   theForm is the root of the form, we can pass the child's next formNodeP
   in the first parameter */

if((UFLCreateCell(UFLGetNext(UFLGetChildren(theForm)), "GREEN_CELL",
   "POPUP_GROUP_COLORS", &theCell)) !=OK)
{
   fprintf(stderr, "Could not create green cell.\n");
   return(NOTOK);
}

/* Set the value of the new cell to "green" */
```

```
if((UFLSetLiteralByRefEx(theCell, NULL, "value", 0, NULL, NULL,
   "green")) !=OK)
{
   fprintf(stderr, "Could not set literal on green cell.\n");
   return(NOTOK);
}

/* Creates the second cell with name purple and group POPUP_GROUP_COLORS,
   and then sets the value of the new cell to "purple".*/

if((UFLCreateCell(UFLGetNext(UFLGetChildren(theForm)), "PURPLE_CELL",
   "POPUP_GROUP_COLORS ", &theCell)) != OK)
{
   fprintf(stderr, "Could not create purple cell.\n");
   return(NOTOK);
}

if((UFLSetLiteralByRefEx(theCell, NULL, "value", 0, NULL, NULL,
   "purple")) != OK)
{
   fprintf(stderr, "Could not set literal for purple cell.\n");
   return(NOTOK);
}
```

UFLDeleteSignature

Description

This function deletes the specified digital signature in the form. For security reasons, the form must meet certain criteria before this is allowed. None of the following should be locked by another signature: the signature, its descendants, the associated signature button, and its signer option. If these criteria are met, then the signature's locks are removed, and the signature item is deleted. Then, and the signer of the associated signature button is set to empty ("").

Function

```
r_short UFLDeleteSignature(
   formNodeP theForm,
   formNodeP signatureItem);
```

Parameters

Expression	Type	Description
theForm	**formNodeP**	The root node of the form containing the signature to delete.
signatureItem	**formNodeP**	The signature node to delete.

Returns

OK on success or an error code on failure.

Notes

If the *signature* item contains a *layoutinfo* option, **UFLDeleteSignature** will not remove the entire signature from the form. Instead, the *signature* item and the *layoutinfo* option will remain. To completely delete the signature item, you must delete the remaining nodes manually by using **UFLDestroy** to delete the signature item.

Example

In the following example, **UFLDereferenceEx** is used to locate the node of a signature button. **UFLDeleteSignature** is then used to delete the signature from the form.

```
r_error deleteSignature(formNodeP form)
{
formNodeP tempNode;
r_boolean layoutinfo;

   /* Locate the signature node. */

   if ((tempNode = UFLDereferenceEx(form, NULL, "PAGE1.SIGNATURE", 0,
      UFL_ITEM_REFERENCE, NULL)) == NULL)
   {
      fprintf(stderr, "Could not locate SIGNATURE node.\n");
      return(NOTOK);
   }

   /* Check to see if the signature contains a layoutinfo option. Set
      layoutinfo to OK if it does or NOTOK if it doesn't. */

   layoutinfo = OK;
   if(UFLDereferenceEx(tempNode, NULL, "layoutinfo", 0,
      UFL_OPTION_REFERENCE, NULL) == NULL)
      layoutinfo = NOTOK;

   /* Delete the signature. */

   if(UFLDeleteSignature(form, tempNode) != OK)
   {
      fprintf(stderr, "Could not delete signature.");
      return(NOTOK);
   }

   /* If there was a layoutinfo option, destroy the remaining signature
      item. */

   if (layoutinfo == OK)
   {
      if(UFLDestroy(tempNode) != OK)
      {
         fprintf(stderr, "Could not destroy signature node.");
         return(NOTOK);
      }
   }
   return(OK);
}
```

UFLDereferenceEx

Description

Use this function to locate a particular formNodeP, to locate a cell in a particular group, or to locate a data item in a particular datagroup.

Note: It is not necessary to call this function when you are using XForms. The UFLReplaceXFormsInstance and UFLExtractXFormsInstancefunctions perform this task automatically.

Function

```
formNodeP UFLDereferenceEx(
   formNodeP aNode,
   r_charP theScheme,
   r_charP theReference,
   r_short theReferenceCode,
   r_u_long referenceType,
   formNodeP theNSNode);
```

Parameters

Expression	Type	Description
aNode	**formNodeP**	The node to use as the starting point for the search.
theScheme	**r_charP**	Reserved. This must be NULL.
theReference	**r_charP**	The reference string.
theReferenceCode	**r_short**	Reserved. This must be 0.
referenceType	**r_short**	One of the following constants: **UFL_OPTION_REFERENCE** **UFL_ITEM_REFERENCE** **UFL_PAGE_REFERENCE** **UFL_ARRAY_REFERENCE** **UFL_GROUP_REFERENCE** **UFL_DATAGROUP_REFERENCE** If it is an option or argument reference, bitwise **OR** (|) with one of: **UFL_SEARCH** **UFL_SEARCH_AND_CREATE** If it is a group or datagroup reference, bitwise **OR** (|) with one of: **UFL_FIRST**
theNSNode	**formNodeP**	A node that is used to resolve the namespaces in *theReference* parameter (see the note about namespace below). Use NULL if the node that this function is operating on has inherited the necessary namespaces.

Returns

On success: the **formNodeP** defined by the reference string or **NULL** if the referenced node does not exist and **UFL_SEARCH_AND_CREATE** is not specified.

Notes

formNodeP

Before you decide which formNodeP to use this function on, be sure you understand the following:

1. The formNodeP supplied can never be more than one level in the hierarchy above the starting point of the reference string. For example, if the reference string begins with an option, then the formNodeP can be no higher in the hierarchy than an item.
2. If the formNodeP is at the same level or lower in the hierarchy than the starting point of the reference string, the function will attempt to locate a common ancestor. The function will locate the ancestor of the formNodeP that is one level in the hierarchy above the starting point of the reference string. The function will then attempt to follow the reference string back down through the hierarchy. If the reference string cannot be followed from the located ancestor (for example, if the ancestor is not common to both the formNodeP and the reference string), the function will fail. For example, given a formNodeP that represents *field_1* and a reference of *field_2*, the function will access the *page* node above *field_1*, and will then try to locate *field_2* below that node. If the two fields were not on the same page, the function would fail.
3. UFLDereferenceEx does not support the XForms scheme.

Creating a Reference String

For general information about creating a reference string, see "References" on page 8.

Reference strings for groups or datagroups follow this format:

```
page.group or page.datagroup
```

In both cases, the page component is optional, and is only required if you want to search a different page than the one containing your reference node.

For example, to refer to the "State" group of cells on PAGE1 of the form, you would use:

```
PAGE1.State
```

Locating Cells or Data Items

If you want to locate a cell or a data item, you must perform a bitwise OR with UFL_FIRST or UFL_NEXT. UFL_FIRST will locate the first cell or data item in the page. UFL_NEXT will locate the next cell or data item. This allows you to loop through all the cells or data item on a page until you have found the one you want.

Note that groups and datagroups are limited to a single page, and that your search will likewise be limited to a single page.

Creating a Node

For an option or argument reference, you can have the library create a node that does not exist. To do so, perform a bitwise OR of UFL_SEARCH_AND_CREATE to the *referenceType* parameter; otherwise, perform a bitwise OR of UFL_SEARCH to the *referenceType* variable and the function will return null if the node does not exist.

Determining Namespace

In some cases, you may want to use the **UFLDereferenceEx** function to locate a node that does not have a globally defined namespace. For example, consider the following form:

```
<label sid="Label1">
   <value>Field1.processing:myValue</value>
</label>
<field sid="Field1" xmlns:processing="URI">
   <value></value>
   <processing:myValue>10<processing:myValue>
</field>
```

In this form, the *processing* namespace is declared in the *Field1* node. Any elements within *Field1* will understand that namespace; however, elements outside of the scope of *Field1* will not.

In cases like this, you will often start your search at a node that does not understand the namespace of the node you are trying to locate. For example, you might want to locate the node referenced in the value of *Label1*. In this case, you would first locate the *Label1* value node and get its literal. Then, from the *Label1* value node, you would attempt to locate the *processing:myValue* node as shown:

```
error = UFLDereferenceEx(Label1Node, NULL, "Field1.processing:myValue",
   0, UFL_OPTION_REFERENCE, NULL)
```

In this example, the **UFLDereferenceEx**function would fail. The function cannot properly resolve the *processing* namespace because this namespace is not defined for the *Label1* value node. To correct this, you must also provide a node that understands the *processing* namespace (in this case, any node in the scope of *Field1*) as the last parameter in the function:

```
error = UFLDereferenceEx(Label1Node, NULL, "Field1.processing:myValue",
   0, UFL_OPTION_REFERENCE, Field1Node)
```

Example

In the following example, **UFLDereferenceEx** is used to locate a specific node. **UFLDestroy** is then used to remove that node from the form structure.

```
r_error removeRadios(formNodeP form)
{
formNodeP tempNode;
r_error error;

   if ((tempNode = UFLDereferenceEx(form, NULL, "PAGE1.MALERADIO", 0,
      UFL_ITEM_REFERENCE, NULL)) == NULL)
   {
      fprintf(stderr, "Could not locate MALERADIO node.\n");
      return(NOTOK);
   }

   error = UFLDestroy(tempNode);
   if (error != OK)
   {
      fprintf(stderr, "UFLDestroy error %hd.\n",
         error);
      return(NOTOK);
   }

   /* additional code removed */

}
```

UFLDestroy

Description

This function destroys the indicated formNodeP. All children of the specified formNodeP are also destroyed.

Function

```
r_short UFLDestroy(
   formNodeP aNode);
```

Parameters

Expression	Type	Description
aNode	**formNodeP**	The form node to destroy. It is valid to pass NULL, in which case nothing happens.

Returns

OK on success or an error code on failure.

Notes

Digital Signatures

You cannot destroy a signed item, except in the case of destroying an entire signed form. Destroying a signed item breaks the digital signature, resulting in an error.

Example

In the following example, **UFLDereferenceEx** is used to locate a particular node. **UFLDestroy** is then used to remove that node from the structure.

```
r_error removeRadios(formNodeP form)
{
formNodeP tempNode;
r_error error;

   if ((tempNode = UFLDereferenceEx(form, NULL, "PAGE1.MALERADIO", 0,
      UFL_ITEM_REFERENCE, NULL)) == NULL)
   {
      fprintf(stderr, "Could not locate MALERADIO node.\n");
      return(NOTOK);
   }

   error = UFLDestroy(tempNode);
   if (error != OK)
   {
      fprintf(stderr, "UFLDestroy error %hd.\n", error);
      return(NOTOK);
   }

   /* additional code removed */
   return(OK);
}
```

UFLDuplicate

Description

This function makes a copy of a node. The duplicate node can be attached to any other node as either a sibling or a child, or can be stored as a separate node structure (that is, as a separate form). The new node can also be assigned a new identifier, as indicated by the *theIdentifier* parameter. All of the properties of the original node are duplicated, including any children and any namespace settings.

Note: If you duplicate a node that is not in the XFDL namespace, the namespace is copied as part of the duplicated node, but is not set globally.

Function

```
formNodeP UFLDuplicate(
   formNodeP origNode,
   formNodeP baseNode,
   r_short where,
   r_charP theIdentifier);
```

Parameters

Expression	Type	Description
origNode	**formNodeP**	The **formNodeP** to copy.
baseNode	**formNodeP**	The **formNodeP** to attach the new copy to. If NULL, then *origNode* is used as the *baseNode*.
where	**r_short**	A constant that describes the location in relation to the supplied *'baseNode'* in which the new node should be placed. Can be one of: **UFL_APPEND_CHILD** — adds the new node as the last child of the *'baseNode'*. **UFL_AFTER_SIBLING** — adds the new node as a sibling of the *'baseNode'*, placing it immediately after that node in the form structure. **UFL_BEFORE_SIBLING** — adds the new node as a sibling of the *'baseNode'*, placing it immediately before that node in the form structure. **UFL_ORPHAN** — copies the node to a new form structure, effectively creating a separate form.
theIdentifier	**r_charP**	A new identifier for this node. If NULL, the same identifier that was on the original node is used.

Returns

The duplicate node on success or **NULL** on failure.

Example

In the following example, **UFLDereferenceEx** is used to locate a specific node. **UFLDuplicate** is then used to duplicate that node. Finally, **UFLSetLiteralByRefEx** is used to change the literal of the duplicate "value" node.

```
r_error createMailing(formNodeP form)
{
formNodeP tempNode;
r_error error;

   if ((tempNode = UFLDereferenceEx(form, NULL,
      "PAGE1.ADDRESSFIELD", 0, UFL_ITEM_REFERENCE, NULL)) == NULL)
   {
      fprintf(stderr, "Could not locate the ADDRESSFIELD node.\n");
      return(NOTOK);
   }
   if ((tempNode = UFLDuplicate(tempNode, NULL, UFL_AFTER_SIBLING,
      "MAILINGFIELD")) == NULL)
   {
      fprintf(stderr, "Could not duplicate ADDRESSFIELD.\n");
      return(NOTOK);
   }

   error = UFLSetLiteralByRefEx(form, NULL, PAGE1.MAILINGFIELD.label",
      0, NULL, NULL, (r_charP)"Mailing Address:");
   if (error != OK)
   {
      fprintf(stderr, "UFLSetLiteralByRefEx error %hd.\n", error);
      return(NOTOK);
   }
   return(OK);
}
```

UFLEncloseFile

Description

This function encloses a file in a form. The file must be accessible on the local computer. The formNodeP may refer to either a page node or an item node. If the formNodeP is a page node, the function creates a data item in that page to contain the enclosure. If the formNodeP is an item node, it must be a data item, and the function encloses the file in that node.

The file is enclosed using base64-gzip encoding.

Function

```
formNodeP UFLEncloseFile(
   formNodeP aNode,
   r_charP theFile,
   r_charP mimeType,
   r_charP dataGroup,
   r_charP theIdentifier);
```

Parameters

Expression	Type	Description
aNode	**formNodeP**	The **formNodeP**, which will contain the enclosure, or a page **formNodeP**, in which a new data item **formNodeP** will be created to contain the enclosure.
theFile	**r_charP**	The path to the file on the local drive to enclose.
mimeType	**r_charP**	The MIME type of the file. If NULL, the library will attempt to find a suitable MIME type for the file.

Expression	Type	Description
dataGroup	**r_charP**	The data group to which this file should belong. If the *aNode* parameter is a page node, you must provide this parameter. If *aNode* parameter is an item node, you may use NULL to keep the current *datagroup* option or provide a different value to overwrite the option.
theIdentifier	**r_charP**	The identifier to assign to the new data item if one is created. If NULL, either the current name is used or a unique name is automatically generated for the new data item.

Returns

The item **formNodeP** that contains the enclosure on success or **NULL** on failure.

Example

In the following example, **UFLEncloseFile** is used to enclose the image file Male.jpg into the root node of a form.

```
if ((tempNode = UFLEncloseFile(aNode, "male.jpg", NULL, NULL, "PICDATA"))
   == NULL)
{
   fprintf(stderr, "Could not enclose image file.\n");
   return(NOTOK);
}
```

UFLEncloseInstance

Description

This function modifies one instance in the data model, either updating information or appending information. Note that the form must have an existing data model.

Note: Use caution when calling this function. It can be used to overwrite signed instance data.

Function

```
r_short UFLEncloseInstance(
formNodeP theNode,
r_charP theInstanceID,
r_charP theFile,
r_u_long theFlags,
r_charP theScheme,
r_charP theRootReference,
formNodeP theNSNode,
r_short replaceNode);
```

Parameters

Expression	Type	Description
theNode	**formNodeP**	The root node of the form or an XML instance node.

Expression	Type	Description
theInstanceID	**r_charP**	The ID of the instance node to create or replace. This is defined by the *id* attribute of that node, and is case sensitive. If *theNode* parameter is the instance node you want to replace, set this parameter to NULL.
theFile	**r_charP**	The path to the file on the local drive that contains the XML instance.
theFlags	**r_u_long**	Reserved. Must be 0.
theScheme	**r_charP**	Reserved. Must be NULL.
theRoot Reference	**r_charP**	A reference to the node you want to replace or append children to. This reference is relative to the instance node. Use NULL to default to the instance node.
theNSNode	**formNodeP**	A node that inherits the namespaces used in the reference. This node defines the namespaces for the function. Use NULL if the node that this function is operating on has inherited the necessary namespaces.
replaceNode	**r_short**	If OK, the node specified by *theRootReference* is replaced with data. If NOTOK, the data is appended as the last child of *theRootReference* node.

Returns

OK on success or an error code on failure.

Example

The following example shows a function that takes the root node of a form and updates the XML instance called ″data″.

```
r_error updateDataInstance(formNodeP theForm)
{
   if (UFLEncloseInstance(theForm, "data",
      "c:\Instance Files\Personnel\tempdata.dat", 0, NULL, NULL, NULL,
      OK) != OK)
   {
      fprintf(stderr, "Could not enclose data instance.");
      return(NOTOK);
   }
   return(OK);
}
```

UFLExtractFile

Description

This function will extract an enclosure contained in a node and save it to a file on the local computer. Note that this function does not remove the enclosure from the form.

Function

```
r_short UFLExtractFile(
   formNodeP aNode,
   r_charP theFile);
```

Parameters

Expression	Type	Description
aNode	**formNodeP**	The item **formNodeP** that contains the enclosure.
theFile	**r_charP**	The path showing where to store the file on the local drive. Any existing file will be overwritten.

Returns

OK on success or an error code on failure.

Example

In the following example, **UFLDereferenceEx** is used to locate a specific data item node. **UFLExtractFile** is then used to write the image data to the local drive.

```
r_error exportImage(formNodeP form)
{
formNodeP tempNode=NULL;

r_error error=OK;
   if ((tempNode = UFLDereferenceEx(form, NULL, "PAGE1.LOGODATA ", 0,
      UFL_ITEM_REFERENCE, NULL)) == NULL)
   {
      fprintf(stderr, "Could not find LOGODATA node.\n");
      return(NOTOK);
   }

   error = UFLExtractFile(tempNode, "logo.jpg");
   if (error != OK)
   {
      fprintf(stderr, "UFLExtractFile error %hd.\n");
      return(NOTOK);
   }
   return(OK);
}
```

UFLExtractInstance

Description

This function copies an instance from a form's XML model to a file. Note that this function does not remove the instance from the form.

Function

```
r_short UFLExtractInstance(
   formNodeP theNode,
   r_charP theInstanceID,
   formNodeP theFilter,
   r_charP includedNamespaces,
   r_charP theFile,
   r_u_long theFlags,
   r_charP theScheme,
   r_charP theRootReference,
   formNodeP theNSNode);
```

Parameters

Expression	Type	Description
theNode	**formNodeP**	The root node of the form or an XML instance node.
theInstanceID	**r_charP**	The ID of the instance node to extract. This is defined by the *id* attribute of that node. If *theNode* parameter is the instance node you want to extract, set this parameter to **NULL**.
theFilter	**formNodeP**	An item in the form, such as a button or cell, that defines the filtering for the instance. Filtering of elements is controlled by the transmit filters in the item. If all of an element's bound options are filtered out, then the element is also filtered out. Use **NULL** for no filtering.
included Namespaces	**r_charP**	If set to NULL, a definition for each inherited namespace is added to the root node of the instance when it is extracted. To filter the namespaces, list the prefixes for those namespaces you want to include in the instance, separated by spaces. For example, to include only the *xfdl* and *custom* namespaces, you would set this parameter to: `xfdl custom` Use *#default* to indicate the default namespace for the instance. Use an empty string (*""*) to include only those namespaces that are used by the instance. Namespaces that are used in the instance are always included, regardless of this setting.
theFile	**r_charP**	The path to the file on the local drive that will contain the XML instance.
theFlags	**r_u_long**	Reserved. This must be 0.
theScheme	**r_charP**	Reserved. Must be NULL.
theRootReference	**r_charP**	A reference to the root node you want to extract. This reference is relative to the instance node. Use NULL to default to the instance node.
theNSNode	**formNodeP**	A node that inherits the namespaces used in the reference. This node defines the namespaces for the function. Use NULL if the node that this function is operating on has inherited the necessary namespaces.

Returns

OK on success or an error code on failure.

Example

The following example shows a function that takes the root node of a form and extracts an XML instance.

```
r_error updateDataInstance(formNodeP theForm)
{
   if (UFLExtractInstance(theForm, "data", NULL, NULL,
      "c:\Instance Files\Personnel\tempdata.dat", 0, NULL, NULL, NULL)
      != OK)
   {
      fprintf(stderr, "Could not extract data instance.");
      return(NOTOK);
   }
   return(OK);
}
```

UFLExtractXFormsInstance

Description

This function copies an XForms instance to a file or a memory block. This function does not remove the instance from the form.

Note: This function automatically updates the XForms data model.

Function

```
r_short UFLExtractXFormsInstance(
   formNodeP theObject,
   r_charP theModelID,
   r_charP theNodeRef,
   r_short writeRelevant,
   r_short ignoreFailures,
   formNodeP *theNSNode,
   r_charP theFilename,
   r_byte *returnPtr,
   r_long *returnSizePtr);
```

Parameters

Expression	Type	Description
theObject	**formNodeP**	The formNodeP object.
theModelID	**r_charP**	The ID of the model to extract. Use NULL to extract the default model.
theNodeRef	**r_charP**	An XPath reference to a node in the instance. This node and all of its children are copied. Leave blank to extract the entire instance.
writeRelevant	**r_short**	If OK, writes only relevant instance data.
ignoreFailures	**r_short**	If OK, ignores constraint or validation failures.
theNSNode	**formNodeP**	A node that inherits the namespaces used in the reference. This node defines the namespaces for the function. Use NULL if the node that this function is operating on has inherited the necessary namespaces.
theFilename	**r_charP**	The name and path of the file to write to. Use NULL to write to the output memory block.

Expression	Type	Description
returnPtr	**r_byte**	A pointer that is set with the memory block that represents the instance or NULL.
returnSizePtr	**r_long**	A pointer that is set with the size of the memory block.

Returns

OK on success or an error code on failure.

Example

The following example shows a function that takes the root node of a form, extracts an XForms instance, and writes it to a file called ″InstanceData.xml″.

```
r_error updateDataInstance(formNodeP theForm)
{
   if (UFLExtractXFormsInstance(theForm, "model1",
      "instance('instance1')/loanrecord/user_personal_info", OK, OK,
       NULL, "c:\\Instance Files\Personnel\tempdata.dat", NULL, NULL)
      != OK)
   {
      fprintf(stderr, "Could not extract data instance.");
      return(NOTOK);
   }
   return(OK);
}
```

UFLGetAttribute

Description

This function returns the value of a specific attribute for a node. For example, the following XFDL represents a MIME data node:

```
<mimedata encoding="base64"></mimedata>
```

In this sample, you could use **UFLGetAttribute** to obtain the value of the encoding attribute, which would be ″base64″.

Function

```
r_error UFLGetAttribute(
   formNodeP theNode,
   r_charP theNamespaceURI,
   r_charP theAttribute,
   r_charP *returnPtr);
```

Parameters

Expression	Type	Description
theNode	**formNodeP**	The node containing the attribute.
theNamespaceURI	**r_charP**	The namespace URI for the attribute. For example: `http://www.ibm.com/xmlns/prod/XFDL/7.0`
theAttribute	**r_charP**	The local name of the attribute. For example, *encoding*.

Expression	Type	Description
**returnPtr*	**r_charP**	A pointer that to stores the value of the attribute. If the attribute is empty or does not exist, this is set to NULL.

Returns

OK on success or an error code on failure.

Notes

Namespaces

If you refer to an attribute with a namespace prefix, *getAttribute* first looks for a complete match, including both prefix and attribute name. If it does not find such a match, it will look for a matching attribute name that has no prefix but whose containing element has the same namespace.

For example, assume that the *custom* namespace and the *test* namespace both resolve to the same URI. In the following case, looking for the *id* attribute would locate the second attribute (test:id), since it has an explicit namespace declaration:

```
<a xmlns:custom="ABC" xmlns:test="ABC">
   <custom:myElement id="1" test:id="2">
</a>
```

However, in the next case, the *id* attribute does not have an explicit namespace declaration. Instead, it inherits the custom namespace. However, since the inherited namespace resolves to the same URI, the *id* attribute is still located:

```
<custom:myElement id="1">
```

Special Attributes

Forms generally use three special attributes that are not in an explicitly defined namespace and which require special commands to retrieve.

The first is the default namespace attribute, which looks like this:

```
xmlns="http://www.ibm.com/xmlns/prod/XFDL/7.0"
```

To retrieve this attribute, you must use a namespace URI of NULL and the attribute name *xmlns*.

The second special attribute is a namespace declaration, which looks like this:

```
xmlns:custom="http://www.ibm.com/xmlns/prod/XFDL/Custom"
```

To retrieve this sort of attribute, you must use the namespace URI *http://www.w3.org/2000/xmlns* and the appropriate attribute name, such as *custom*.

Finally, there is the language attribute, which looks like this:

```
xml:lang="en-GB"
```

To retrieve this sort of attribute, you must use the namespace URI *http://www.w3.org/XML/1998/namespace* and the attribute name *lang*.

Example

The following example shows a shortcut function that gets the value of the encoding attribute for a specific node. A node is passed to the function which then uses **UFLGetAttribute** to get the value of encoding attribute. This sample function assumes that the attribute is always in the XFDL namespace.

```
r_charP getEncodingType(formNodeP theNode)
{
r_charP theEncodingType;
r_error error;
   if ((error = UFLGetAttribute(theNode,
      "http://www.ibm.com/xmlns/prod/XFDL/7.0", "encoding", &theEncodingType))
      != OK)
   {
      fprintf(stderr, "Could not retrieve attribute's value.\n.");
      return(NULL);
   }
   return(theEncodingType);
}
```

UFLGetAttributeList

Description

This function returns a list of attributes and a list of corresponding namespaces for a given node. For example, the following XFDL represents a *mimedata* node:

```
<mimedata encoding="base64"></mimedata>
```

In this sample, *getAttributeList* would return a list of attributes that contained *encoding* and a list of namespaces that contained *http://www.ibm.com/xmlns/prod/XFDL/7.0*.

Function

```
r_error UFLGetAttributeList(
   formNodeP theNode,
   r_charP **theNamespaceList,
   r_long *theNamespaceListSize,
   r_charP **theAttributeList,
   r_long *theAttributeListSize);
```

Parameters

Expression	Type	Description
theNode	**formNodeP**	The form node containing the attributes.
theNamespaceList	**r_charP****	A pointer that contains a list of namespace URIs. For example: `http://www.ibm.com/xmlns/prod /XFDL/7.0` Each URI corresponds to the attribute in the same position in the attribute list.
theNameSpaceListSize	**r_long***	A pointer that contains the number of entries in the namespace list.

Expression	Type	Description
theAttributeList	**r_charP****	A pointer that contains a list of attributes. For example, *compute*, *encoding*, and so on. Each attribute corresponds to a URI in the same position in the namespace list.
theAttributeListSize	**r_long***	A point that contains the number of entries in the attribute list.

Returns

OK on success or an error code on failure.

Example

The following function uses **UFLGetAttributeList t**o retrieve the list of a node's attributes. It then searches through the list looking for a compute attribute. When if locates a compute attribute, it uses **UFLRemoveAttribute t**o remove the compute from the node.

```
r_error stripComputes(formNodeP theNode)
{
r_long counter;
r_charP *theNamespaces;
r_long theNamespaceCount;
r_charP *theAttributes;
r_long theAttributeCount;
r_error error;
   /* Retrieve the list of attributes for the supplied node. */
   if ((error = UFLGetAttributeList(theNode, &theNamespaces,
      &theNamespaceCount, &theAttributes, &theAttributeCount)) != OK)
   {
      fprintf(stderr, "Could not retrieve attribute list.\n");
      return(error);
   }
   /* Step through the list searching for the compute attribute. If the
      compute attributes is found, delete it. */
   for (counter = 0; counter < theAttributeCount; counter++)
   {
      if (cp_strcmp(theAttributes[counter], "compute") == OK)
      {
         if ((error = UFLRemoveAttribute(theNode,
            theNamespaces[counter], theAttributes[counter])) != OK)
         {
            fprintf(stderr, "Could not remove attribute.\n");
            return(error);
         }
      }
   }
   /* Free memory. */
   for (counter = 0; counter < theAttributeCount; counter++)
   {
      cp_free(theNamespaces[counter]);
      cp_free(theAttributes[counter]);
   }
   pe_free((r_voidP)theNamespaces);
   pe_free((r_voidP)theAttributes);
   return(OK);
}
```

UFLGetCertificateList

Description

This function locates all available certificates that can be used by a particular signature button. The certificates are filtered according to the signature engine defined in the *signformat* option of the button, and according to the filters defined in the *signdetails* option of the button.

This function returns the valid certificates in an undetermined order. This means that you cannot rely on the certificates being listed in the same order each time you call this function.

Note that each certificate in the list is tracked by reference counts. Once you are done with the certificates, you must release the reference counts to the certificates and free the array (see the example for details).

Function

```
r_error UFLGetCertificateList(
   formNodeP buttonNode,
   r_charP theFilters,
   SecurityUserStatusType *theStatus,
   Certificate ***theCertificate,
   r_long *certCount);
```

Parameters

Expression	Type	Description
buttonNode	**formNodeP**	The node that represents the signature button.
theFilters	**r_charP**	A string that is used to filter the subject attribute of the certificate. If the subject attribute include this substring, then that certificate will be listed. For example, you might filter against a name, such as ″John Doe″, or an e-mail address, such as ″jdoe@ibm.com″. Note that this filter is in addition to the other filters defined in the *signdetails* option of the button. If NULL is passed, then only the filters in the *signdetails* option are used.
theStatus	**SecurityUserStatusType***	This is a status flag that reports whether the operation was successful. Possible values are: **SUSTATUS_OK** — the operation was successful. **SUSTATUS_CANCELLED** — the operation was cancelled by the user. **SUSTATUS_INPUT_REQUIRED** — the operation required user input, but could not receive it (for example, it was run on a server with no user).

Expression	Type	Description
certList	**Certificate*** **	The list of certificates that the function locates. Note that each certificate object is tracked by reference counts, and must be released. Furthermore, the array must also be freed.
certCount	**r_long***	The number of certificates that the function located.

Returns

OK on success or an error code on failure.

Example

In the following example, **UFLDereferenceEx** is used to locate a specific signature button node. **UFLGetCertificateList** is then used to get a list of valid certificates for that button. Next, **Certificate_GetDataByPath** is used to search for the Workplace Forms Server certificate, which **UFLSignForm** then uses to sign the button. Finally, the reference counts to the certificate list and the signature object are released.

```
r_error signButton(formNodeP form)
{
SecurityUserStatusType theStatus;
formNodeP buttonNode;
Certificate **certList;
Signature *theSignature;
r_charP signerCommonName;
r_boolean encodedResult;
r_error error;
r_long certCount;
r_long correctCert = -1;
r_long i;

   if ((buttonNode = UFLDereferenceEx(form, NULL, "PAGE1.SIGNBUTTON",
      0, UFL_ITEM_REFERENCE, NULL)) == NULL)
   {
      fprintf(stderr, "Could not find SIGNBUTTON node.\n");
      return(NOTOK);
   }

   if ((error = UFLGetCertificateList(buttonNode, NULL, &theStatus,
      &certList, &certCount)) != OK)
   {
      fprintf(stderr, "UFLGetCertificateList error %hd.\n", error);
      return(NOTOK);
   }

   /* Check the status, in case the process required user input. */

   if (theStatus != SUSTATUS_OK)
   {
      fprintf(stderr, "User input required to retrieve certificate list.
         /n");
      return(NOTOK);
   }
   /* Iterate through the certificates to find one with a common name of
      Workplace Forms Server */

   for (i=0; i<certCount; i++)
   {
      if ((error = Certificate_GetDataByPath(certList[i],
```

```
            "SigningCert: Subject: CN", NOTOK, &encodedResult,
            &signerCommonName)) != OK)
         {
            fprintf(stderr, "Certificate_GetDataByPath error %hd./n",
               error);
            return(NOTOK);
         }
         if (cp_strcmp(signerCommonName, "Workplace Forms Server") == OK)
         {
            correctCert = i;
            cp_free(signerCommonName);
            break;
         }
         cp_free(signerCommonName);
      }

      /* If the correct certificate was not located, report the problem and
         exit the function. */

      if (correctCert == -1)
      {
         fprintf(stderr, "Could not locate required certificate.");
         return(NOTOK);
      }

      /* Use the Workplace Forms Server certificate to sign the form */

      if ((error = UFLSignForm(buttonNode, certList[correctCert], NULL,
         &theStatus, &theSignature)) != OK)
      {
         fprintf(stderr, "UFLSignForm error %hd.\n", error);
         return(NOTOK);
      }

      /* Check the status in case the process required user input. */

      if (theStatus != SUSTATUS_OK)
      {
         fprintf(stderr, "User input required to sign form./n");
         return(NOTOK);
      }

      /* Release each certificate object in the array */

      for(i=0; i<certCount; i++)
      {
         IFSObject_ReleaseRef((IFSObject*)certCount[i]);
      }

      /* Free the array */

      pe_free(certList);

      /* Relase the signature object */

      IFSObject_ReleaseRef((IFSObject*)theSignature);

      return(OK);
   }
```

UFLGetChildren

Description

This function, along with **UFLGetParent**, is used to traverse vertically along the form hierarchy. **UFLGetChildren** returns the first child of the indicated node. If the

node has no children, NULL is returned. All children of a particular formNodeP can be traversed using an iterator, such as a while loop, in combination with **UFLGetNext**.

Function

```
formNodeP UFLGetChildren(
   formNodeP theParentNode);
```

Parameters

Expression	Type	Description
theParentNode	**formNodeP**	The **formNodeP** to retrieve its child from.

Returns

The **formNodeP** that represents the child or **NULL** if no such child exists.

Example

The following example uses **UFLGetChildren** in a while loop to vertically traverse down the form hierarchy until the last node on the branch is reached. The bottom node is returned.

getChildren returns the first child node of *PAGE1.NAMELABEL* that is *PAGE1.NAMELABEL.value*.

```
formNodeP getBottomNode(formNodeP *theNode)
{
formNodeP theBottomNode = NULL;
formNodeP theChild;

   if(*theNode != NULL)
   {
      theChild = UFLGetChildren(*theNode);
   }
   while (theChild != NULL)
   {
      reportNodeInfo(&theChild);
      theBottomNode = theChild;
      theChild = UFLGetChildren(theChild);
   }
   return theBottomNode;
}
```

UFLGetFormVersion

Description

This function determines the XFDL version of a form.

Function

```
r_u_long UFLGetFormVersion(
   formNodeP theFormNode);
```

Parameters

Expression	Type	Description
theFormNode	**formNodeP**	Any form node that belongs to the XFDL namespace.

Returns

A long in the form of 0xMMmm0300, where MM is the major number and mm is the minor number. For example, a version 6.3 form would return: 0x06030300.

Example

The following function accepts a form node and returns a boolean that indicates whether the form is version 6.5 or higher.

```
r_boolean checkVersion(formNodeP theNode)
{
   if (UFLGetFormVersion(theNode) >= 0x06050300)
   {
      return(OK);
   }
   else
   {
      return(NOTOK);
   }
}
```

UFLGetInfoEx

Description

This function retrieves the information about a **formNodeP**. If you do not want information about a particular property, simply set it to NULL.

Function

```
r_short UFLGetInfoEx(
   formNodeP aNode,
   r_charP *theTypePtr,
   r_charP *theLiteralPtr,
   r_charP *theFormulaPtr,
   r_charP *theIdentifierPtr,
   r_charP theCharSet);
```

Parameters

Expression	Type	Description
aNode	***formNodeP***	The **formNodeP** to retrieve information about.
theTypePtr	***r_charP****	A pointer that will store the type of the **formNodeP**. If the type is empty or does not exist, the string is set to NULL. This string must be freed by the caller.

Expression	Type	Description
theLiteralPtr	***r_charP****	A pointer that will store the literal of the **formNodeP**. If the literal is empty or does not exist, the string is set to NULL. This string must be freed by the caller.
theFormulaPtr	***r_charP****	A pointer that will store the formula of the **formNodeP**. If the formula is empty or does not exist, the string is set to NULL. This string must be freed by the caller.
theIdentifierPtr	***r_charP****	A pointer that will store the identifier of the **formNodeP**. If the identifier is empty or does not exist, the string is set to NULL. This string must be freed by the caller.
theCharSet	***r_charP***	The character set you want to use to view the results. Use **NULL** or **ANSI** for ANSI. Use **Symbol** for Symbol.

Returns

OK on success or an error code on failure.

Notes

If you are getting information about a node that is not in the XFDL namespace, **UFLGetInfoEx** may return values that include namespace prefixes as follows:

- Any item node in a non-XFDL namespace will return a *Type* that includes a namespace prefix. For example, *myNamespace:Field1*.
- Any option node in a non-XFDL namespace will return an *Identifier* that includes a namespace prefix. For example, *myNamespace:value*.

Example

In the following example, **UFLDereferenceEx** is used to locate a specific node. **UFLGetInfoEx** is then used to get the four values from that node. The four values are then printed out.

```
r_error checkTotalNode(formNodeP form)
{
formNodeP tempNode=NULL;
r_charP theType=NULL;
r_charP theLiteral=NULL;
r_charP theFormula=NULL;
r_charP theIdentifier=NULL;
r_error error = OK;
      if ((tempNode = UFLDereferenceEx(form, NULL, "PAGE1.TOTALFIELD",0,
         UFL_ITEM_REFERENCE, NULL)) == NULL
      {
         fprintf(stderr, "Could not locate TOTALFIELD node.\n");
         return(NOTOK);
      }
      error = UFLGetInfoEx(tempNode, &theType, &theLiteral, &theFormula,
         &theIdentifier, NULL);
```

```
      if (error != OK)
      {
         fprintf(stderr, "UFLGetInfoEx error %hd.\n", error);
         return(NOTOK);
      }
      /* Print the information */
      fprintf(stderr, "The type of this node is: %s.\n", theType);
      fprintf(stderr, "The literal of this node is: %s.\n", theLiteral);
      fprintf(stderr, "The formula of this node is: %s.\n", theFormula);
      fprintf(stderr, "The identifier of this node is: %s.\n",
         theIdentifier);
      /* Free memory. */
      cp_free(theType);
      cp_free(theLiteral);
      cp_free(theFormula);
      cp_free(theIdentifier);
      theType = NULL;
      theLiteral = NULL;
      theFormula = NULL;
      theIdentifier = NULL;
      return(OK);
}
```

UFLGetLiteralByRefEx

Description

This function finds a particular formNodeP on the basis of a reference string. You must provide a node that is used as the starting point for the search unless you provide an absolute reference. Once the formNodeP is found, its literal is retrieved.

Note: It is not necessary to call this function when you are using XForms. The UFLReplaceXFormsInstance and UFLExtractXFormsInstancefunctions perform this task automatically.

Function

```
r_short UFLGetLiteralByRefEx(
   formNodeP aNode,
   r_charP theScheme,
   r_charP theReference,
   r_u_long theReferenceCode,
   r_charP theCharSet,
   formNodeP theNSNode,
   r_charP *theLiteralPtr);
```

Parameters

Expression	Type	Description
aNode	**formNodeP**	A **formNodeP** that determines which form to search, and provides a starting point for the search if an absolute reference is not provided.
theScheme	**r_charP**	Reserved. This must be NULL.
theReference	**r_charP**	The reference string.
theReferenceCode	**r_u_long**	Reserved. This must be 0.
theCharSet	**r_charP**	The character set you want to use to view the literal string. Use **NULL** or **ANSI** for ANSI. Use **Symbol** for Symbol.

Expression	Type	Description
theNSNode	**formNodeP**	A node that is used to resolve the namespaces in *theReference* parameter (see the note about namespace below). Use NULL if the node that this function is operating on has inherited the necessary namespaces.
theLiteralPtr	**r_charP***	A pointer that will store the literal string. If the literal is empty or does not exist, the string is set to **NULL**. This string must be freed by the caller.

Returns

OK on success or an error code on failure.

Notes

formNodeP

Before you decide which formNodeP to use as the *aNode* parameter, be sure you understand the following:

1. The formNodeP supplied can never be more than one level in the hierarchy above the starting point of the reference string. For example, if the reference string begins with an option, then the formNodeP can be no higher in the hierarchy than an item.
2. If the formNodeP is at the same level or lower in the hierarchy than the starting point of the reference string, the function will attempt to locate a common ancestor. The function will locate the ancestor of the formNodeP that is one level in the hierarchy above the starting point of the reference string. The function will then attempt to follow the reference string back down through the hierarchy. If the reference string cannot be followed from the located ancestor (for example, if the ancestor is not common to both the formNodeP and the reference string), the function will fail.

   For example, given a formNodeP that represents ″field_1″ and a reference of ″field_2″, the function will access the ″page″ node above ″field_1″, and will then try to locate ″field_2″ below that node. If the two fields are not on the same page, the function will fail.
3. If the formNodeP is at the argument level, the search will not start from that point. Instead, the nearest ancestor that is at the option level will be used as the starting point for the search.

Creating a Reference String

For more information about creating a reference, see “References” on page 8.

Determining Namespace

In some cases, you may want to use the **UFLGetLiteralByRefEx**function to get the literal of a node that does not have a globally defined namespace. For example, consider the following form:

```
<label sid="Label1">
   <value>Field1.processing:myValue</value>
</label>
```

```
<field sid="Field1" xmlns:processing="URI">
   <value></value>
   <processing:myValue>10<processing:myValue>
</field>
```

In this form, the *processing* namespace is declared in the *Field1* node. Any elements within *Field1* will understand that namespace; however, elements outside of the scope of *Field1* will not.

In cases like this, you will often start your search at a node that does not understand the namespace of the node you are trying to locate. For example, you might want to locate the node referenced in the value of *Label1*. In this case, you would first locate the *Label1* value node and get its literal. Then, from the *Label1* value node, you would attempt to locate the *processing:myValue* node as shown:

```
error = UFLGetLiteralByRefEx(Label1Node, NULL,
   "Field1.processing:myValue", 0, NULL, NULL, &theLiteral)
```

In this example, the **UFLGetLiteralByRefEx** function would fail. The function cannot properly resolve the *processing* namespace because this namespace is not defined for the *Label1* value node. To correct this, you must also provide a node that understands the *processing* namespace (in this case, any node in the scope of *Field1*) as a parameter in the function:

```
error = UFLGetLiteralByRefEx(Label1Node, NULL,
   "Field1.processing:myValue", 0, NULL, Field1Node, &theLiteral)
```

Example

The following example uses **UFLGetLiteralByRefEx** to get the literal value from a specific node. That value is then converted into an integer.

```
r_error getCurrentDate(formNodeP form, int *curMonth, int *curDay)
{
r_charP temp=NULL;
r_error error;

   error = UFLGetLiteralByRefEx(form, NULL, PAGE1.CURRENTMONTH.value",
      0, NULL, NULL, &temp);
   if (error != OK)
   {
      fprintf(stderr, "UFLGetLiteralByRefEx error %hd.\n", error);
      return(NOTOK);
   }

/* If a literal value was returned, convert it into an integer value;
   otherwise, indicate that no value was entered into the field and throw
   an exception. */

   if (temp != NULL)
      *curMonth = atoi((char *)temp);
   else
   {
      fprintf(stderr, "The current month was not entered.\n");
      return(NOTOK);
   }

   /* additional code removed */

   /* Free memory */

   cp_free(temp);
   temp = NULL;
   return(OK);
}
```

UFLGetLiteralEx

Description

This function retrieves the literal of a node. The literal is returned in the specified character set.

Note: It is not necessary to call this function when you are using XForms. The UFLReplaceXFormsInstance and UFLExtractXFormsInstancefunctions perform this task automatically.

Function

```
r_short UFLGetLiteralEx(
   formNodeP aNode,
   r_charP theCharSet,
  r_charP *theLiteral);
```

Parameters

Expression	Type	Description
aNode	**formNodeP**	The **formNodeP** to retrieve the literal from.
theCharSet	**r_charP**	The character set you want to use to view the literal string. Use **NULL** or **ANSI** for ANSI. . Use **Symbol** for Symbol.
theLiteral	**r_charP***	A pointer that stores the literal string. If the literal is empty or does not exist, the string is set to NULL. This string must be freed by the caller.

Returns

OK on success or an error code on failure.

Example

The following example uses **UFLDereferenceEx** to locate a specific node. **UFLGetLiteralEx** is then used to get the literal value for that node.

```
if ((tempNode = UFLDereferenceEx(form, NULL, "PAGE1.MALERADIO.value", 0,
   UFL_OPTION_REFERENCE, NULL)) == NULL)
{
   fprintf(stderr, "Could not locate MALERADIO node.\n");
   return(NOTOK);
}

error = UFLGetLiteralEx(tempNode, NULL, &temp);
if (error != OK)
{
   fprintf(stderr, "Could not get literal of MALERADIO value node. Error
      code: %hd.\n", error);
   return(NOTOK);
}
```

UFLGetLocalName

Description

This function returns the *local name* of a given node. The local name is determined by the XML tag that represents that node. For example, examine the following XML fragment:

```
<page sid="PAGE1">
   <global sid="global"></global>
   <field sid="testField">
      <value>Hello</value>
      <bgcolor>
         <ae>120</ae>
         <ae>120</ae>
         <ae>120</ae>
      <bgcolor>
   </field>
</page>
```

In this sample, the name of the page node is ″page″, the name of the field node is ″field″, the name of the value node is ″value″, and the name of the bgcolor node is ″bgcolor″. The bgcolor node is also the parent of three array element nodes, all of which are named ″ae″.

Note that the local name does not include any namespace prefix that might exist. For example, you might have a custom option in a different namespace as shown:

```
<field sid="testField">
   <custom:my_option>value</custom:my_option>
</field>
```

In this case, the local name of the custom option is returned without the prefix, resulting in ″my_option″.

Function

```
r_error UFLGetLocalName(
   formNodeP theNode,
   r_charP *localName);
```

Parameters

Expression	Type	Description
theNode	**formNodeP**	The form node.
**localName*	**r_charP**	A pointer that contains the local name of the node.

Returns

OK on success or an error code on failure.

Example

The following function takes the root node of the form and uses recursion to step through each node in the form. The function uses **UFLIsXFDL** and **UFLGetLocalName** to locate all label nodes in the XFDL namespace and changes the background color of those nodes to green.

```
r_error changeLabelColor(formNodeP theNode)
{
formNodeP tempnode, bgcolorNode;
```

```
r_boolean XFDLNode;
r_charP localName;
r_error error;

   /* Use recursion to step through each node in the form. */

   if ((tempNode = UFLGetChildren(theNode)) == NULL)
   {
      fprintf(stderr, "Could not locate child node.\n");
      return(NOTOK);
   }

   while(tempNode != NULL)
   {
      if ((error = changeLabelColor(tempNode)) != OK)
      {
         fprintf(stderr, "Could not call changeLabelColor.\n");
         return(error);
      }
   }

   /* Check to see if the node is in the XFDL namespace. */

   if ((error = UFLIsXFDL(theNode, &XFDLNode)) != OK)
   {
      fprintf(stderr, "Could not determine whether node is in XFDL
         namespace");
      return(error);
   }

   /* Get the local name of the node. */

   if ((error = UFLGetLocalName(theNode, &localName)) != OK)
   {
      fprintf(stderr, "Could not get local name of node.\n");
      return(error);
   }

   /* If the node is a label in the XFDL namespace, set the bgcolor
      option to "green". */

   if ((XFDLNode == OK) && (cp_strcmp(localName, "label") == OK))
   {
      if ((error = UFLSetLiteralByRefEx(theNode, NULL, "bgcolor[0]", 0,
         NULL, NULL, "green")) != OK)
      {
         fprintf(stderr, "Could not set node color to green.\n");
         return(error);
      }
   }

   /* Free memory. */

   cp_free(localName);

   return(OK);
}
```

UFLGetNamespaceURI

Description

This function returns the *namespace URI* for the node.

Each namespace is defined in the form by a namespace declaration, as shown:

```
xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
xmlns:custom="http://www.ibm.com/xmlns/prod/XFDL/Custom"
```

Each namespace declaration defines both a prefix and a URI for the namespace. In this sample, the prefix for the XFDL namespace is *xfdl* and the URI is *http://www.ibm.com/xmlns/prod/XFDL/7.0*.

Tags within the form are assigned specific namespaces by using the defined prefix. For example, to declare that an option was in the custom namespace you would use the prefix *custom* as shown:

```
<field sid="testField">
   <custom:custom_option>value</custom:custom_option>
</field>
```

Function

```
r_error UFLGetNamespaceURI(
   formNodeP theFormNode,
   r_charP *theURI);
```

Parameters

Expression	Type	Description
theFormNode	**formNodeP**	The **formNodeP** to retrieve the namespace URI for.
theURI	**r_charP***	A pointer that will contain the URI. This string must be freed by the caller.

Returns

OK on success or an error code on failure.

Example

The following function uses recursion to traverse the entire node structure and destroys all nodes that are in the *custom* namespace identified by the following URI: *http://www.ibm.com/xmlns/prod/XFDL/Custom*. This function assumes that you are passing in the root node of the form.

```
r_error deleteCustomInfo(formNodeP theNode)
{
formNodeP tempNode, tempNode2;
r_charP theURI;

   /* Use recursion to step through each node of the form. */

   if ((tempNode = UFLGetChildren(theNode)) == NULL)
   {
      fprintf(stderr, "Could not locate child node.");
      return(NOTOK);
   }
   while(tempNode != null)
   {
      if ((tempNode2 = UFLGetNext(tempNode)) == NULL)
      {
         fprintf(stderr, "Could not locate sibling.");
         return(NOTOK);
      }
      if ((error = deleteCustomInfo(tempNode)) != OK)
      {
         fprintf(stderr, "deleteCustomInfo returned an error.");
         return(NOTOK);
```

```
      }
      tempNode = tempNode2;
   }

   /* Get the namespace URI for the node. */

   if ((UFLGetNamespaceURI(theNode, &theURI)) != OK)
   {
      fprintf(stderr, "Could not get namespace URI.");
      return(NOTOK);
   }

   /* If the node belongs to the custom namespace, delete it. */

   if (cp_strcmp(theURI, "http://www.ibm.com/xmlns/prod/XFDL/Custom") == OK)
   {
      if (UFLDestroy(theNode) != OK)
      {
         fprintf(stderr, "Could not delete node.");
         return(NOTOK);
      }
   }

   /* Free memory. */

   cp_free(theURI);
   return(OK);
}
```

UFLGetNamespaceURIFromPrefix

Description

This function returns the *namespace URI* that corresponds to a specific prefix.

Each namespace is defined in the form by a namespace declaration, as shown:

```
xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
xmlns:custom="http://www.ibm.com/xmlns/prod/XFDL/Custom"
```

Each namespace declaration defines both a prefix and a URI for the namespace. In this sample, the prefix for the XFDL namespace is *xfdl* and the URI is *http://www.ibm.com/xmlns/prod/XFDL/7.0*.

Tags within the form are assigned specific namespaces by using the defined prefix. For example, to declare that an option was in the custom namespace you would use the prefix *custom* as shown:

```
<field sid="testField">
   <custom:custom_option>value</custom:custom_option>
</field>
```

Function

```
r_error UFLGetNamespaceURIFromPrefix(
   formNodeP theFormNode,
   r_charP thePrefix,
   r_charP *theURI);
```

Parameters

Expression	Type	Description
theFormNode	**formNodeP**	Any node in the form that either declares or inherits the namespace in question.

Expression	Type	Description
thePrefix	**r_charP**	The namespace prefix. For example, *xfdl*.
theURI	**r_charP***	A pointer that will contain the namespace URI. For example: `http://www.ibm.com/xmlns/prod/XFDL/7.0` If the namespace URI is not declared, the pointer is set to **NULL**. Note that this string must be freed by the caller.

Returns

OK on success or an error code on failure.

Example

The following function copies a custom option from one form to another. The function assumes that you know the prefix for the custom namespace, but not the URI. First, the function uses **UFLGetNamespaceURIFromPrefix** to get the URI for the custom namespace in the first form. Next, it adds that namespace to the second form as a globally available namespace. It then locates the custom node in the first form and the global item node in the second form. Finally, it copies the custom node to the second form as a child of the global item node.

```
r_error copyCustomInfo(formNodeP form1, formNodeP form2)
{
   formNodeP tempNode, duplicateNode, globalNode;
   r_charP theURI;

   /* Get the URI for the custom namespace in form 1. */

   if (UFLGetNamespaceURIFromPrefix(form1, "custom", &theURI) != OK)
   {
      fprintf(stderr, "Could not get URI.");
      return(NOTOK);
   }

   /* Check to see if the URI is NULL. */

   if (theURI == NULL)
   {
      fprintf(stderr, "Could not get namespace URI.");
      return(NOTOK);
   }

   /* Create a custom namespace in form 2 using that URI. */

   if (UFLAddNamespace(form2, theURI, "custom") != OK)
   {
      fprintf(stderr, "Could not add namespace to second form.");
      return(NOTOK);
   }

   /* Locate the custom Status node in form 1. */

   if ((tempNode = UFLDereferenceEx(form1, NULL,
      "global.global.custom:Status", 0, UFL_OPTION_REFERENCE |
      UFL_SEARCH, NULL)) == NULL)
   {
      fprintf(stderr, "Could not locate custom Status node.");
      return(NOTOK);
   }
```

```
   /* Locate the global item in form 2. */

   if ((globalNode = UFLDereferenceEx(form2, NULL, "global.global", 0,
      UFL_ITEM_REFERENCE | UFL_SEARCH, NULL)) == NULL)
   {
      fprintf(stderr, "Could not locate global item in second form.");
      return(NOTOK);
   }

   /* Copy the custom node from form 1 and insert it as a child of
      the global item in form 2. */

   if ((duplicateNode = UFLDuplicate(tempNode, globalNode,
      UFL_APPEND_CHILD, NULL)) == NULL)
   {
      fprintf(stderr, "Could not duplicate node.");
      return(NOTOK);
   }

   /* Free memory. */

   cp_free(theURI);

   return(OK);
}
```

UFLGetNext

Description

This function, along with **UFLGetPrevious**, is used to traverse horizontally along the form hierarchy. **UFLGetNext** returns the next node in the tree. For instance, the page node corresponding to the first page of your form can be reached by calling **UFLGetNext** on the global page node.

Function

```
formNodeP UFLGetNext(
   formNodeP theFormNode);
```

Parameters

Expression	Type	Description
theFormNode	**formNodeP**	The **formNodeP** to retrieve the next node from.

Returns

The formNodeP that represents the next node or NULL if no such node exists.

Example

The following example uses **UFLGetNext** in a while loop to horizontally traverse across the form hierarchy until the last node on the branch is reached. The last node is returned.

```
formNodeP getLastNode(formNodeP *theNode)
{
formNodeP theLastNode = NULL;
formNodeP theNextNode;

   if(*theNode != NULL)
```

```
    {
       theNextNode = UFLGetNext(*theNode);
    }
    while (theNextNode != NULL)
    {
       reportNodeInfo(&theNextNode);
       theLastNode = theNextNode;
       theNextNode = UFLGetNext(theNextNode);
    }
    return theLastNode;
}
```

UFLGetNodeType

Description

This function returns the type for a node (for example, page, item, option, and so on). This allows you to quickly determine the type of node you are working with and what depth you are at in the node hierarchy.

Function

```
r_error UFLGetNodeType(
   formNodeP theFormNode,
   r_u_long *theType);
```

Parameters

Expression	Type	Description
theFormNode	**formNodeP**	The node to get the type for.
theType	**r_u_long***	A pointer that will contain the node's type. This will be one of the following constants: **UFL_FORM** — The root node of the form. **UFL_PAGE** — A page level node. **UFL_ITEM** — An item level node. **UFL_OPTION** — An option level node. **UFL_ARRAY** — An argument level node, such as an array element.

Returns

OK on success or an error code on failure.

This method throws a generic exception (**UWIException**) if an error occurs.

This function throws an exception if an error occurs.

Example

The following function recieves a node below the page level and uses **UFLGetParent** to ascend the hierarchy until it reaches a page node, as detected by **UFLGetNodeType**.

```
formNodeP ascendToPage(formNodeP theNode)
{
r_u_long theType;
```

```
   /* Get the type of the node. */

   if (UFLGetNodeType(theNode, &theType) != OK)
   {
      fprintf(stderr, "Could not get node type.");
      return(NULL);
   }

   /* While the node is not NULL and the node is not a page node,
      get the parent of the node. */

   while ((theNode != NULL) && (theType != UFL_PAGE))
   {

      /* Get the parent node. */

      theNode = UFLGetParent(theNode);

      /* Get the type of the node. */

      if (theNode != NULL)
      {
         if (UFLGetNodeType(theNode, &theType) != OK)
         {
            fprintf(stderr, "Could not get node type.");
            return(NULL);
         }
      }
   }
   return(theNode);
}
```

UFLGetParent

Description

This function, along with **UFLGetChildren**, is used to traverse vertically along the form hierarchy. **UFLGetParent** returns the parent of a node. If the node has no parent, NULL is returned. A form's structure can be traversed up to the root node using an iterator such as a while loop.

Function

```
formNodeP UFLGetParent(
   formNodeP theChildNode);
```

Parameters

Expression	Type	Description
theChildNode	**formNodeP**	The **formNodeP** to retrieve its parent from.

Returns

The formNodeP that represents the parent node or NULL if no such parent exists.

Example

The following example uses **UFLGetParent** in a while loop to vertically traverse up the form hierarchy until the top node on the branch is reached. The top node is returned.

getParent returns the parent node of *PAGE1.AGEFIELD.size*, that is, *PAGE1.AGEFIELD*.

```
formNodeP getTopNode(formNodeP *theNode)
{
formNodeP theTopNode = NULL;
formNodeP theParent;

   if(*theNode != NULL)
   {
      theParent = UFLGetParent(*theNode);
   }
   while (theParent != NULL)
   {
      reportNodeInfo(&theParent);
      theTopNode = theParent;
      theParent = UFLGetParent(theParent);
   }
   return theTopNode;
}
```

UFLGetPrefix

Description

This function returns the namespace *prefix* for the node.

Each namespace is defined in the form by a namespace declaration, as shown:

```
xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
xmlns:custom="http://www.ibm.com/xmlns/prod/XFDL/Custom"
```

Each namespace declaration defines both a prefix and a URI for the namespace. In this sample, the prefix for the XFDL namespace is *xfdl* and the URI is *http://www.ibm.com/xmlns/prod/XFDL/7.0*.

Tags within the form are assigned specific namespaces by using the defined prefix. For example, to declare that an option was in the custom namespace you would use the prefix *custom* as shown:

```
<field sid="testField">
   <custom:custom_option>value</custom:custom_option>
</field>
```

Note: A given prefix may not always resolve to the same namespace. Different portions of the form may define the prefix differently. For example, the custom prefix may resolve to a different namespace on the first page of a form than it does on the following pages.

Function

```
r_error UFLGetPrefix(
   formNodeP theFormNode,
   r_charP *thePrefix);
```

Parameters

Expression	Type	Description
theFormNode	**formNodeP**	The **formNodeP** to get the namespace prefix for.
thePrefix	**r_charP***	A pointer that will contain the prefix. This string must be freed by the caller.

Returns

OK on success or an error code on failure.

Example

The following function removes all nodes from the form that have a namespace prefix of "custom". The function walks through the form using **UFLGetChildren**and **UFLGetNext** in a recursive loop. While walking the form, it uses **UFLGetPrefix** to locate nodes in the custom namespace and deletes them using **UFLDestroy**. This function assumes that you are passing it the root node of the form.

```
r_error deleteCustomInfo(formNodeP theNode)
{
formNodeP tempNode, tempNode2;
r_charP thePrefix;

   /* Use recursion to step through each node of the form. */

   tempNode = UFLGetChildren(theNode);
   while(tempNode != null)
   {
      tempNode2 = UFLGetNext(tempNode);
      if (deleteCustomInfo(tempNode) != OK)
      {
         fprintf(stderr, "deleteCustomInfo returned an error.");
         return(NOTOK);
      }
      tempNode = tempNode2;
   }

   /* Get the namespace prefix for the node. */

   if (UFLGetPrefix(theNode, &thePrefix) != OK)
   {
      fprintf(stderr, "Could not get namespace URI.");
      return(NOTOK);
   }

   /* If the node belongs to the custom namespace, delete it. */

   if (cp_strcmp(thePrefix, "custom") == 0)
   {
      if (UFLDestroy(theNode) != OK)
      {
         fprintf(stderr, "Could not delete node.");
         return(NOTOK);
      }
   }

   /* Free memory. */

   cp_free(thePrefix);

   return(OK);
}
```

UFLGetPrefixFromNamespaceURI

Description

This function returns the namespace *prefix* for a specific namespace URI

Each namespace is defined in the form by a namespace declaration, as shown:

```
xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
xmlns:custom="http://www.ibm.com/xmlns/prod/XFDL/Custom"
```

Each namespace declaration defines both a prefix and a URI for the namespace. In this sample, the prefix for the XFDL namespace is *xfdl* and the URI is *http://www.ibm.com/xmlns/prod/XFDL/7.0*.

Tags within the form are assigned specific namespaces by using the defined prefix. For example, to declare that an option was in the custom namespace you would use the prefix *custom* as shown:

```
<field sid="testField">
   <custom:custom_option>value</custom:custom_option>
</field>
```

Function

```
r_error UFLGetPrefixFromNamespaceURI(
   formNodeP theFormNode,
   r_charP theURI,
   r_charP *thePrefix);
```

Parameters

Expression	Type	Description
theFormNode	**formNodeP**	Any node in the form that either declares or inherits the namespace in question.
theURI	**r_charP**	The namespace URI. For example: `http://www.ibm.com/xmlns/prod/XFDL/7.0`
thePrefix	**r_charP***	A pointer that will contain the prefix. If the namespace URI is not declared, the pointer is set to **NULL**. Note that this string must be freed by the caller.

Returns

OK on success or an error code on failure.

Example

The following function adds custom information to a form and assumes that the namespace URI for the custom information is known but that the prefix used to represent that namespace in the form is not known. First, the function uses **UFLGetPrefixFromNamespaceURI** to get the prefix in use. The function then concatenates the prefix with the name for the new node, "Status". Finally, the function locates the global item in the global page and creates a new option node.

```
r_error addStatus(formNodeP theNode)
{
r_charP thePrefix;
r_charP theNodeName;
```

```
   /* Get the prefix for the custom namespace. */

   if ((UFLGetPrefixFromNamespaceURI(theNode,
      "http://www.ibm.com/xmlns/prod/XFDL/Custom", &thePrefix)) != OK)
   {
      fprintf(stderr, "Could not get namespace prefix.");
      return(NOTOK);
   }

   /* Check whether the prefix is NULL. */

   if (thePrefix == NULL)
   {
      fprintf(stderr, "The namespace URI does not exist.");
      return(NOTOK);
   }

   /* Allocate memory to hold a name for the new node. */

   theNodeName=cp_malloc(cp_strlen(thePrefix) + 8);

   /* Create a name for a new node by concatenating the prefix with
      "Status". */

   cp_strcpy(theNodeName, thePrefix);
   cp_strcat(theNodeName, ":Status");

   /* Locate the global item in the global page so we can add a global
      option. */

   if ((theNode = UFLDereferenceEx(theNode, NULL, "global.global", 0,
      UFL_ITEM_REFERENCE | UFL_SEARCH, NULL)) == NULL)
   {
      fprintf(stderr, "Could not find global.global node.");
      return(NOTOK);
   }

   /* Create a new node in the custom namespace and give it a value
      of "Processed". */

   if ((theNode = UFLCreate(theNode, UFL_APPEND_CHILD, NULL,
      "Processed", NULL, theNodeName)) == NULL)
   {
      fprintf(stderr, "Could not create node.");
      return(NOTOK);
   }

   /* Free memory. */

   cp_free(thePrefix);
   cp_free(theNodeName);

   return(OK);
}
```

UFLGetPrevious

Description

This function, along with **UFLGetNext**, is used to traverse horizontally along the form hierarchy. **UFLGetPrevious** returns the previous node in the tree. For instance, if you call **UFLGetPrevious** one the Page1 node in your form, it will return the global page node.

Function

```
formNodeP UFLGetPrevious(
   formNodeP theFormNode);
```

Parameters

Expression	Type	Description
theFormNode	**formNodeP**	The **formNodeP** to retrieve the previous node from.

Returns

The formNodeP that represents the previous node or NULL if no such node exists.

Example

The following example uses **UFLGetPrevious** in a while loop to horizontally traverse across the form hierarchy until the first node on the branch is reached. The first node is returned.

```
formNodeP getFirstNode(formNodeP *theNode)
{
formNodeP theFirstNode = NULL;
formNodeP thePreviousNode;

   if(*theNode != NULL)
   {
      thePreviousNode = UFLGetPrevious(*theNode);
   }
   while (thePreviousNode != NULL)
   {
      reportNodeInfo(&thePreviousNode);
      theFirstNode = thePreviousNode;
      thePreviousNode = UFLGetPrevious(thePreviousNode);
   }
   return theFirstNode;
}
```

UFLGetReferenceEx

Description

This function returns the reference string that identifies the node. For example, a value node might return a reference of *Page1.Field1.value*. The reference will either begin at the page level of the form or at a level specified by the caller.

Function

```
r_short UFLGetReferenceEx(
   formNodeP aNode,
   r_charP theScheme,
   formNodeP theNSNode,
   formNodeP theStartPoint,
   r_short addNamespaces,
   r_charP *theReference);
```

Parameters

Expression	Type	Description
aNode	**formNodeP**	The node to get a reference for.

Expression	Type	Description
theScheme	**r_charP**	Reserved. This must be NULL.
theNSNode	**formNodeP**	A node that defines which namespace prefixes are used when constructing the reference. Only namespace prefixes that this node inherits are used. Use NULL if the node that this function is operating on has inherited the necessary namespaces.
theStartPoint	**formNodeP**	A node that determines the starting point of the reference. This node must be a parent of the *aNode* parameter. The reference will begin one level below the start point node. For example, if you provide a page node the reference will begin at the item level. Use NULL to start the reference at the page level.
addNamespaces	**r_short**	Use OK to add declarations for unknown namespaces to the namespace node (*theNSNode*). Otherwise, use NOTOK.
theReference	**r_charP***	A pointer that will store the reference. This string must be freed by the caller.

Returns

OK on success or an error code on failure.

Notes

Creating a Reference String

For more information about creating a reference, see "References" on page 8.

Working with Namespace Prefixes

In some cases, you may want to use the **UFLGetReferenceEx** function to get the reference to a node that uses a different prefix for a known namespace. For example, consider the following form:

```
<label sid="Label1" xmlns:data="URI">
   <value></value>
</label>
<field sid="Field1" xmlns:processing="URI">
   <value></value>
   <processing:myValue>10<processing:myValue>
</field>
```

In this form, *processing* and *data* are prefixes for the same namespace, since they both refer to the same URI. However, both namespaces have limited scope since they are declared at the item level. This means that *Label1* node does not understand the *processing* prefix, and that the *Field1* node does not understand the *data* prefix.

This becomes a problem if you want to refer to a namespace from a location that does not understand that namespace. For example, suppose you wanted to set the value of *Label1* to be a reference to the *myValue* node in *Field1*. Normally, you would locate the *myValue* node and use **getReferenceEx** as shown:

```
error = UFLGetReferenceEx(myValueNode, NULL, NULL, NULL, NOTOK,
   &theReference);
```

In this case, **UFLGetReferenceEx** would return the following reference: *Page1.Field1.processing:myValue*. However, because the *processing* namespace is not defined for *Label1*, a reference to the *processing* namespace is not understood. This means that you cannot set the value of *Label1* to equal this reference, since the node would not understand that content.

Instead, you must generate a reference that includes a known namespace prefix, such as the *data* namespace. You can do this by including a second node in the **UFLGetReferenceEx** function. The second node must understand the appropriate namespace. For example, you could include the *Label1* node in the function, as shown:

```
error = UFLGetReferenceEx(myValueNode, NULL, Label1Node, NULL, NOTOK,
   &theReference);
```

In this case, the function will substitute the *data* prefix for the *processing* prefix, since they both resolve to the same namespace. As a result, the function will return: *Page1.Field1.data:myValue*. Since the *data* prefix is defined within *Label1*, you can use this reference to set *Label1*'s value node.

Working with Unknown Namespaces

In some cases, you may want to use the **UFLGetReferenceEx** function to get the reference to a node that uses an unknown namespace. For example, consider the following form:

```
<page sid="Page1" xmlns:processing="URI1">
   <global sid="global">
      <processing:info></processing:info>
   </global>
   <field sid="Field1" xmlns:data="URI2">
      <value></value>
      <data:info>data</data:info>
   </field>
```

In this example, you might want to store a reference to the <data:info> element in the <processing:info> element. **UFLGetReferenceEx** would return the following reference for the <data:info> element: *Page1.Field1.data:info*. However, this reference includes the *data* namespace, which is not defined for the page global. This means that you could not store this reference in the <processing:info> element, because it would not understand the reference.

To solve this problem, you can use the *addNamespaces* flag in the **UFLGetReferenceEx** function. When this flag is set to OK, the function will add unknown namespaces to the *theNSNode*.

For example, if you set *theNSNode* to be the global item node for *Page1*, and set the *addNamespace* flag to OK, as shown:

```
error = UFLGetReferenceEx(dataNode, NULL, pageGlobalNode, NULL, OK,
   &theReference);
```

The function would return the reference to the <data:info> element, but would also modify the global item node to include the unknown *data* namespaces, as shown:

```
<global sid="global" xmlns:data="URI2">
```

You could then store the reference in that global item or any of its descendants, since the namespace is now properly defined.

Example

In the following example, a page node is passed to the function. The function then uses **UFLGetChildren** and **UFLGetNext** to locate the last item node in the page. **UFLGetReferenceEx** is then called to get the reference to that node, which is returned to the caller.

```
r_charP getLastItemReference(formNodeP pageNode)
{
formNodeP itemNode, tempNode;
r_charP theReference;

   /* Get the first item node in the page. */

   if ((itemNode = UFLGetChildren(pageNode)) == NULL)
   {
      fprintf(stderr, "Could not locate child node.");
      return(NOTOK);
   }

   /* Cycle through to the last item node in the page. */

   while ((tempNode = UFLGetNext(itemNode)) != NULL)
   {
      itemNode = tempNode;
   }

   /* Get the reference to the node and return it. */

   if (UFLGetReferenceEx(itemNode, NULL, NULL, NULL, NOTOK,
      &theReference) != OK)
   {
      fprintf(stderr, "Could not get reference to node.");
      return(NOTOK);
   }
   return(theReference);
}
```

UFLGetSecurityEngineName

Description

This function returns the name of the appropriate security engine for a given button or signature node. This is useful for determining which validation call you need to make to validate the signature.

Function

```
r_error UFLGetSecurityEngineName(
   formNodeP sigNode,
   r_u_long theOperation,
   r_charP *theName);
```

Parameters

Expression	Type	Description
sigNode	**formNodeP**	The button or signature node to find a security engine for.

Expression	Type	Description
theOperation	**r_u_long**	The operation you want the security engine for. Possible values are: **SEOPERATION_SIGN** — the engine is needed to sign the form. **SEOPERATION_VERIFY** — the engine is needed to verify the signature. **SEOPERATION_LISTIDENTITIES** — the engine is needed to generate a list of valid certificates for signing.
theName	**r_charP***	The name of the appropriate security engine. The possible names are: • CryptoAPI • Netscape • ClickWrap • HMAC-ClickWrap • PenOp Note that this string must be freed.

Returns

OK on success or an error code on failure.

Example

The following example uses **UFLGetSecurityEngineName** to get the appropriate engine for a signature verification. If the engine is *HMAC-ClickWrap*, the example calls a function that will verify an HMAC signature. Otherwise, the example calls a function that verifies other types of signatures.

```
r_error validateSignature(formNodeP sigNode)
{
r_charP engineName;
r_error error;

   if ((error = UFLGetSecurityEngineName(sigNode, SEOPERATION_VERIFY,
      &engineName)) == OK)
   {
      fprintf (stderr, "UFLGetSecurityEngineName error %hd.\n", error);
      return(NOTOK);
   }
   if (cp_strcmp(engineName, "HMAC-ClickWrap") == 0)
   {
      if ((error = validateAuthenticatedClickwrapSignature(sigNode))
         != OK)
      {
         fprintf (stderr, "validateAuthenticatedClickwrapSignature error
            %hd./n", error);
         return(NOTOK);
      }
   }
   else
   {
      if ((error = validateNormalSignature(sigNode)) != OK)
      {
         fprintf (stderr, "validateNormalSignature error %hd./n",
```

```
            error);
         return(NOTOK);
      }
   }
   cp_free(engineName);
}
```

UFLGetSigLockCount

Description

This function returns the signature lock count of a node. If 0 is returned, the node is not signed by any digital signature, but it may have descendants that are signed.

Function

```
r_short UFLGetSigLockCount(
   formNodeP theNode);
```

Parameters

Expression	Type	Description
theNode	**formNodeP**	The node to query. This can be **NULL**, in which case 0 is returned.

Returns

The number of locks on the given node or NOTOK on error.

Example

In the following example, **UFLDereferenceEx** is used to locate the name field node. **UFLGetSigLockCount** is then used to verify that the node is signed.

```
r_error checkSigLocks(formNodeP form)
{
formNodeP tempNode;
r_error numLocks;
   if ((tempNode = UFLDereferenceEx(form, NULL, "PAGE1.NAMEFIELD", 0,
      UFL_ITEM_REFERENCE, NULL)) == NULL)
   {
      fprintf(stderr, "Could not locate NAMEFIELD node.\n");
      return(NOTOK);
   }
   numLocks = UFLGetSigLockCount(tempNode);
   if (numLocks == NOTOK)
   {
      fprintf(stderr, "UFLGetSigLock error %hd.\n", numLocks);
      return(NOTOK);
   }
   if (numLocks != 1)
   {
      fprintf(stderr, "NAMEFIELD not locked.");
      return(NOTOK);
   }
   return(OK);
}
```

UFLGetSignature

Description

This function returns signature object for a given *button* or *signature* item.

Function

```
r_error  UFLGetSignature(
   formNodeP theItem,
   Signature **theSignature);
```

Parameters

Expression	Type	Description
theItem	**formNodeP**	The node that represents the button or signature item.
theSignature	**Signature****	The signature object that the function locates. Note that this object is tracked by reference counts, and must be released.

Returns

OK on success or an error code on failure.

Example

The following example uses **UFLGetSignature** to get the signature object from the signature node, and uses **Signature_GetDataByPath** to get the signer's identity from the signature object. It then calls **UFLValidateHMACWithSecret** to validate the signature. Finally, it releases the signature object.

```
r_error checkSignature(formNodeP theSignatureNode, Certificate *theServerCert,
   r_short *validation)
{
Signature *theSignatureObject;
r_charP theSecret;
r_charP signerCommonName;
r_boolean encodedData;
SecurityUserStatusType theStatus;
r_error error;

   if ((error = UFLGetSignature(theSignatureNode, &theSignatureObject))
      != OK)
   {
      fprintf(stderr, "UFLGetSignature error %ld.\n", error);
      return(error);
   }
   if ((error = Signature_GetDataByPath(theSignatureObject,
      "SigningCert: Subject: CN", NOTOK, &encodedData,
      &signerCommonName)) != OK)
   {
      fprintf(stderr, "Signature_GetDataByPath error %ld./n", error);
      return(error);
   }

   /* Include external code that matches the signer's identity to a shared
      secret, and sets theSecret to match.  This is most likely a
      database lookup. */

   if ((error = UFLValidateHMACWithSecret(theSignatureNode, theSecret,
      theServerCert, &theStatus, validation)) != OK)
```

```
    {
       fprintf(stderr, "UFLValidateHMACWithSecret error %ld.\n", error);
       return(error);
    }
    /* Check the status in case the process required user input. */

    if (theStatus != SUSTATUS_OK)
    {
       fprintf(stderr, "User input required to sign form./n");
       return(NOTOK);
    }
    cp_free(signerCommonName);

    /* Release the reference to the signature object. */

    IFSObject_ReleaseRef((IFSObject*)theSignature);

    return(OK);
}
```

UFLGetSignatureVerificationStatus

Description

When called, this function checks to see if the digital signatures in a given form are valid.

Function

```
r_short UFLGetSignatureVerificationStatus(
   formNodeP theForm,
   r_short *validSigsFlagPtr);
```

Parameters

Expression	Type	Description
theForm	**formNodeP**	The form in which to check the signatures.
validSigsFlagPtr	**r_short***	A pointer to a location storing the results of whether the signatures are valid. It will be set to OK if the signatures are valid or to NOTOK if there is at least one signature that is not valid.

Returns

OK on success or NOTOK on failure.

Additionally, the validSigsFlagPtr will contain one of the following values:

Code	Status
UFL_SIGS_OK	The signatures are valid.
UFL_SIGS_NOTOK	One or more signatures are broken.
UFL_SIGS_UNVERIFIED	One or more signatures are unverifiable.

Example

The following example uses **UFLGetSignatureVerificationStatus** to check if the signatures in a loaded form are valid. If any of the signatures are not valid, an error message is reported.

```
r_error verifyForm(formNodeP form)
{
r_error error;
r_short validFlag;

   error = UFLGetSignatureVerificationStatus(form, &validFlag);
   if (error != OK)
   {
      fprintf(stderr, "UFLGetSignatureVerificationStatus error %hd.\n",
         error);
      return(NOTOK);
   }
   if (validFlag != UFL_SIGS_OK)
   {
      fprintf(stderr, "At least one signature in form not valid");
   }
   return(OK);
}
```

UFLIsSigned

Description

This function determines whether a node is signed.

Function

```
r_error UFLIsSigned(
   formNodeP theFormNode,
   r_boolean excludeSelf,
   r_boolean *theStatus);
```

Parameters

Expression	Type	Description
theFormNode	**formNodeP**	Any form node.
excludeSelf	**r_boolean**	A signature node is always self-signed. To determine whether a second signature has been applied to that node, you must exclude the self-signing from this check. To exclude the self-signing from the signature check, set this to OK. To include the self-signing, set this to NOTOK.
returnPtr	**r_boolean***	A pointer that will contain the node's status. **OK** indicates that it is signed, **NOTOK** indicates that it is not signed.

Returns

OK on success or an error code on failure.

Example

The following function locates the value node for a Date field, checks to see if it is signed, and sets the value if the node is not signed.

```
r_error setDateValue(r_charP date, formNodeP theForm);
{
formNodeP tempNode;
r_boolean theStatus;
```

```
   /* Locate the value option for the Date field */

   if ((tempNode = UFLDereferenceEx(theForm, NULL,
      "PAGE1.Date.value", 0, UFL_OPTION_REFERENCE, NULL))
      == NULL)
   {
      fprintf(stderr, "Could not locate value node for Date.\n");
      return(NOTOK);
   }

   /* Check the value node to see if it is signed. */

   if (UFLIsSigned(tempNode, NOTOK, &theStatus) != OK)
   {
      fprintf(stderr,
         "Could not determine whether Date's value is signed.\n");
      return(NOTOK);
   }

   /* If the value node is signed, return an error. Otherwise, set it to
      the value passed into the function. */

   if (theStatus == OK)
   {
      fprintf(stderr, "Date's value is signed.\n");
      return(NOTOK);
   }
   else
   {
      if (UFLSetLiteralEx(tempNode, NULL, date) != OK)
      {
         fprintf(stderr, "Could not set literal for Date.\n");
         return(NOTOK);
      }
   }
   return(OK);
}
```

UFLIsValidFormat

Description

This function returns the boolean result of whether a string is valid according to the setting of the node's *format* option.

This function does not support XForms nodes.

Function

```
r_error UFLIsValidFormat(
   formNodeP theFormNode,
   r_charP theString,
   r_boolean *returnPtr);
```

Parameters

Expression	Type	Description
theFormNode	**formNodeP**	Any form node that is or contains a *format* option.
theString	**r_charP**	A string to be checked against the format. For example, to check 23.2 against a specific format, the string would be "23.2".

Expression	Type	Description
returnPtr	**r_boolean***	A pointer that will contain the node's status. **OK** indicates that the format is valid, **NOTOK** indicates that it is not valid.

Returns

OK on success or an error code on failure.

Example

The following function locates the Currency field and checks to see if "23.2" conforms to the format required by the field's *format* option.

```
r_error checkFormat(formNodeP theForm);
{
formNodeP theItem;
r_boolean isValid;

/* Locate the Currency field */

if ((theItem = UFLDereferenceEx(theForm, NULL, "PAGE1.Currency", 0,
   UFL_ITEM_REFERENCE, NULL)) == NULL)
{
   fprintf(stderr, "Could not locate the Currency node.\n");
   return(NOTOK);
}

/* Check the string to see if it is valid*/

UFLIsValidFormat(theItem, "23.2", &isValid);
   if (isValid == OK)
   {
      printf("The value was valid according to the format\n");
   }
   else
   {
      printf("The value was invalid according to the format\n");
   }
   return(OK);
}
```

UFLIsXFDL

Description

This function determines whether a node belongs to the XFDL namespace.

Each namespace is defined in the form by a namespace declaration, as shown:

```
xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
xmlns:custom="http://www.ibm.com/xmlns/prod/XFDL/Custom"
```

Each namespace declaration defines both a prefix and a URI for the namespace. In this sample, the prefix for the XFDL namespace is *xfdl* and the URI is *http://www.ibm.com/xmlns/prod/XFDL/7.0*.

Tags within the form are assigned specific namespaces by using the defined prefix. For example, to declare that an option was in the custom namespace you would use the prefix *custom* as shown:

```
<field sid="testField">
   <custom:custom_option>value</custom:custom_option>
</field>
```

Function

```
r_error UFLIsXFDL(
   formNodeP theFormNode,
   r_boolean *theStatus);
```

Parameters

Expression	Type	Description
theFormNode	**formNodeP**	The node to check.
theStatus	**r_boolean***	A pointer that will contain the node's status. OK indicates that it is in the XFDL namespace, NOTOK indicates that it is not.

Returns

OK on success or an error code on failure.

Example

The following function uses recursion to traverse the entire node structure and destroys all nodes that are not in the XFDL namespace. This function assumes that you are passing in the root node of the form.

```
r_error deleteCustomInfo(formNodeP theNode)
{
r_boolean theStatus;
formNodeP tempNode, tempNode2;

   /* Use recursion to step through each node in the form. */

   tempNode = UFLGetChildren(theNode);
   while(tempNode != null)
   {
      tempNode2 = UFLGetNext(tempNode);
      if (deleteCustomInfo(tempNode) != OK)
      {
         fprintf(stderr, "deleteCustomInfo returned an error.");
         return(NOTOK);
      }
      tempNode = tempNode2;
   }

   /* Determine whether the node is in the XFDL namespace. */

   if (UFLIsXFDL(theNode, &theStatus) != OK)
   {
      fprintf(stderr, "Could not determine if node is XFDL.");
      return(NOTOK);
   }
   /* If the node is not in the XFDL namespace, delete it. */
   if (*theStatus != OK)
   {
      if (   UFLDestroy(theNode) != OK)
      {
         fprintf(stderr, "Could not delete node.");
         return(NOTOK);
```

```
        }
    }
    return(OK);
}
```

UFLRemoveAttribute

Description

This function removes a specific attribute from a node. For example, the following XFDL represents a value node:

```
<value custom:myAtt="x"></value>
```

To remove the custom attribute from this node, you would use *removeAttribute*.

Function

```
r_error UFLRemoveAttribute(
  formNodeP theObject,
  r_charP theNamespaceURI,
  r_charP theName)
```

Parameters

Expression	Type	Description
theNode	**formNodeP**	The node containing the attribute you want to remove.
theNamespaceURI	**r_charP**	The namespace URI for the attribute. For example: `http://www.ibm.com/xmlns/prod/XFDL/7.0`
theAttribute	**r_charP**	The local name of the attribute. For example, *compute*, *encoding*, and so on.

Returns

OK on success or an error code on failure.

Notes

Attributes and the Null Namespace

If an attribute is on a node in a non-XFDL namespace, and that attribute has no namespace prefix, then the attribute is in the *null* namespace. For example, the following node is the custom namespace, as is the first attribute, but since the second attribute does not have a namespace prefix, it is in the null namespace:

```
<custom:processing custom:stage="2" user="tjones">
```

When an attribute is the null namespace, you may either provide a NULL value for the namespace URI or use the namespace URI for the containing element.

For example, to indicate *user* attribute on the *processing* node, you could use the null namespace or the custom namespace URI.

Attributes and Namespace Prefixes

If you refer to an attribute with a namespace prefix, **UFLRemoveAttribute** first looks for a complete match, including both prefix and attribute name. If it does not find such a match, it will look for a matching attribute name that has no prefix but whose containing element has the same namespace.

For example, assume that the *custom* namespace and the *test* namespace both resolve to the same URI. In the following case, looking for the *id* attribute would locate the second attribute (test:id), since it has an explicit namespace declaration:

```
<a xmlns:custom="ABC" xmlns:test="ABC">
   <custom:myElement id="1" test:id="2">
</a>
```

However, in the next case, the *id* attribute does not have an explicit namespace declaration. Instead, it inherits the custom namespace. However, since the inherited namespace resolves to the same URI, the *id* attribute is still located:

```
<custom:myElement id="1">
```

Example

The following function uses **UFLDereferenceEx** to locate a custom node in the form. It then uses **UFLRemoveAttribute** to delete the ″stage″ attribute from the node, and calls **UFLSetAttribute** to update the value of the status attribute to ″completed″.

```
r_error CompletedProcessing(form)
{
   formNodeP tempNode;

   /* Locate the custom processing node in the global item */

   if ((tempNode = UFLDereferenceEx(form, NULL,
      "global.global.custom:processing", 0, UFL_OPTION_REFERENCE, NULL))
      == NULL)
   {
      fprintf(stderr, "Could not locate custom node.\n");
      return(NOTOK);
   }

   /* Remove the "stage" attribute from the node */

   if (UFLRemoveAttribute(tempNode, NULL, "stage") != OK)
   {
      fprintf(stderr, "Could not remove attribute.\n");
      return(NOTOK);
   }

   /* Update the status attribute to "completed" */

   if (UFLSetAttribute(tempNode, NULL, "status", "completed") != OK)
   {
      fprintf(stderr, "Could not remove attribute.\n");
      return(NOTOK);
   }
}
```

UFLRemoveEnclosure

Description

This function will either remove an enclosure from a specific datagroup or delete the enclosure from the form.

Function

```
r_short UFLRemoveEnclosure(
   formNodeP aNode,
   r_charP theDataGroup);
```

Parameters

Expression	Type	Description
aNode	**formNodeP**	The item **formNodeP** that contains the enclosure to remove.
theDataGroup	**r_charP**	The datagroup that contains the enclosed item. If NULL, the item will be removed from all datagroups. If an item no longer belongs to any datagroups, it is deleted from the form.

Returns

OK on success or an error code on failure.

Example

The following example uses **UFLDereferenceEx** to locate a specific data node. **UFLRemoveEnclosure** is then used to remove the node from the form.

```
r_error deleteLogo(formNodeP form)
{
formNodeP tempNode=NULL;
r_error error=OK;

   if ((tempNode = UFLDereferenceEx(form, NULL, "PAGE1.LOGODATA", 0,
      UFL_ITEM_REFERENCE, NULL)== NULL)
   {
      fprintf(stderr, "Could not locate LOGODATA node.\n");
      return(NOTOK);
   }
   error = UFLRemoveEnclosure(tempNode, NULL);
   if (error != OK)
   {
      fprint(stderr, "UFLRemoveEnclosure error %hd.\n);
      return(NOTOK);
   }
   return(OK);
}
```

UFLReplaceXFormsInstance

Description

This function either inserts or replaces an XForms instance in a form's data model. The instance can come from either from either a file or a memory block.

Use caution when calling this function. It can be used to overwrite signed instance data.

Note: This function automatically updates the XForms data model.

Function

```
r_short UFLReplaceXFormsInstance(
   formNodeP theObject,
   r_charP theModelID,
   r_charP theNodeRef,
   formNodeP *theNSNode,
   r_charP theFilename,
   r_byte *theMemoryBlock,
   r_long *theMemoryBlockSize
   r_short replaceRef);
```

Parameters

Expression	Type	Description
theObject	**formNodeP**	The formNodeP object.
theModelID	**r_charP**	The ID of the affected model. You must use NULL to use the default model.
theNodeRef	**r_charP**	An XPath reference to the instance (or portion of an instance) you want to replace. An empty string indicates the default instance of the selected model.
theNSNode	**formNodeP**	A node that inherits the namespaces used in the reference. This node defines the namespaces for the function. Use NULL if the node that this function is operating on has inherited the necessary namespaces.
theFilename	**r_charP**	The file to read the instance from.
theMemoryBlock	**r_byte**	The memory block that contains the instance if you are not reading from a file, input stream, or Reader. Use **NULL** if *theFilename* is used.
theMemoryBlock Size	**r_long**	The size of *theMemoryBlock*.
replaceRef	**r_short**	If OK, the node specified by *theNodeRef* is replaced with data. If NOTOK, the data is appended as the last child of the instance node.

Returns

OK on success or an error code on failure.

Example

The following example shows a function that replaces an XForms instance.

```
r_error updateDataInstance(formNodeP theForm)
{
   if (UFLReplaceXFormsInstance(theForm, "model1",
      "instance('instance1')/loanrecord/user_personal_info", NULL,
      "c:\\InstanceData.xml", NULL, 0, OK)
      != OK)
   {
      fprintf(stderr, "Could not replace data instance.");
      return(NOTOK);
   }
   return(OK);
}
```

UFLSetActiveForComputationalSystem

Description

This function sets whether the computational system is active. When active, all computes in the form are evaluated on an on-going basis. When inactive, no computes are evaluated.

Note that turning the computational system on causes all computes in the form to be re-evaluated, which can be time consuming.

Function

```
r_short UFLSetActiveForComputationalSystem(
   formNodeP theForm,
   r_short activeFlag);
```

Parameters

Expression	Type	Description
theForm	**formNodeP**	Any node in the form.
activeFlag	**r_short**	Set to OK to activate the compute system or NOTOK to deactivate the compute system.

Returns

OK on success or an error code on failure.

Example

The following example reads a form into memory with the computational system turned off. The example then calls a processing function that adds a large amount of information to the form. Next, **UFLSetActiveForComputationalSystem** is called to turn the computational system on and evaluate all of the computes. Finally, the updated form is written to disk.

```
r_error processForm()
{
formNodeP theForm;
r_error error;

   /* Read the form into memory with the computes turned off */

   if ((theForm = UFLReadForm("input.xfd", UFL_AUTOCOMPUTE_OFF)) == NULL)
   {
      fprintf(stderr, "Could not read form.\n");
      return(NOTOK);
   }

   /* Call a function that adds information to the form from a database */

   addInformation(theForm);

   /* Activate the computational system. This will re-evaluate all
      computes with the new information in the form. */

   if ((error = UFLSetActiveForComputationalSystem(theForm, OK)) != OK)
   {
      fprintf(stderr, "Could not activate compute system.\n");
      return(error);
   }
```

```
   /* Write the updated form to disk */

   if ((error = UFLWriteForm("output.xfd", NULL, 0)) != OK)
   {
      fprintf(stderr, "Could not write form to disk.\n");
      return(error);
   }
}
```

UFLSetAttribute

Description

This function sets the value of a specific attribute for a node. For example, the following XFDL represents a value node:

```
<value custom:myAtt="x"></value>
```

To change the custom attribute, you would use **setAttribute**. If the attribute does not already exist, **setAttribute** will create it and assign the appropriate value.

Note: Do not use **UFLSetAttribute** to set the compute attribute. Instead, use **UFLSetFormula**.

Function

```
r_error UFLSetAttribute(
   formNodeP theFormNode;
   r_charP theNamespaceURI,
   r_charP theAttribute,
   r_charP theValue
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theFormNode	**formNodeP**	The node to set the attribute for.
theNamespaceURI	**r_charP**	The namespace URI for the attribute. For example: `http://www.ibm.com/xmlns/prod/XFDL/7.0`
theAttribute	**r_charP**	The local name of the attribute. For example, *encoding*.
theValue	**r_charP**	The value to assign to the attribute.

Returns

OK on success or an error code on failure.

Notes

Attributes and the Null Namespace

If an attribute is on a node in a non-XFDL namespace, and that attribute has no namespace prefix, then the attribute is in the *null* namespace. For example, the following node is the custom namespace, as is the first attribute, but since the second attribute does not have a namespace prefix, it is in the null namespace:

```
<custom:processing custom:stage="2" user="tjones">
```

When an attribute is the null namespace, you may either provide a NULL value for the namespace URI or use the namespace URI for the containing element.

For example, to indicate *user* attribute on the *processing* node, you could use the null namespace or the custom namespace URI.

Attributes and Namespace Prefixes

If you refer to an attribute with a namespace prefix, **UFLSetAttribute** first looks for a complete match, including both prefix and attribute name. If it does not find such a match, it will look for a matching attribute name that has no prefix but whose containing element has the same namespace.

For example, assume that the *custom* namespace and the *test* namespace both resolve to the same URI. In the following case, looking for the *id* attribute would locate the second attribute (test:id), since it has an explicit namespace declaration:

```
<a xmlns:custom="ABC" xmlns:test="ABC">
   <custom:myElement id="1" test:id="2">
</a>
```

However, in the next case, the *id* attribute does not have an explicit namespace declaration. Instead, it inherits the custom namespace. However, since the inherited namespace resolves to the same URI, the *id* attribute is still located:

```
<custom:myElement id="1">
```

Example

The following example shows a shortcut method that sets a custom data attribute for a specific node. A node and a string containing the contents of the attribute are passed to the function, which then uses **UFLSetAttribute** to set the attribute for the node.

```
r_error setCustomAttribute(formNodeP theNode, r_charP theContents)
{
   if (UFLSetAttribute(theNode,
      "http://www.ibm.com/xmlns/prod/XFDL/Custom",
      "Data", theContents) != OK)
   {
      fprintf(stderr, "Could not set attribute.");
      return(NOTOK);
   }
}
```

UFLSetFormula

Description

This function sets the formula for a node.

Function

```
r_short UFLSetFormula(
   formNodeP aNode,
   r_charP theFormula);
```

Parameters

Expression	Type	Description
anode	**formNodeP**	The **formNodeP** to set the formula for.

Expression	Type	Description
theFormula	**r_charP**	The formula to assign to the *aNode*. If NULL, the formula is assigned as NULL.

Returns

OK on success or an error code on failure.

Example

In this example, **UFLDereferenceEx** is used to locate an age field. **UFLSetFormula** is then used to set the appropriate formula for the field.

```
r_error setFormula(formNodeP form, int curMonth, int curDay,
   int birMonth, int birDay)
{
formNodeP tempNode=NULL;
r_error error=OK;

   if ((tempNode = UFLDereferenceEx(form, NULL, "PAGE1.AGEFIELD.value",
      0, UFL_OPTION_REFERENCE | UFL_SEARCH_AND_CREATE, NULL)) == NULL)
   {
      fprintf(stderr, "Could not locate AGEFIELD node.\n");
      return(NOTOK);
   }

/* The following logic simply identifies how the formula should be set. If
   the current date is later in the year than the birth date, then the age
   is: current year - birth year. If the current date is earlier in the
   year then the birth date, then the age is: current year - birth year -
   1. */

   if ((curMonth > birMonth) || ((curMonth == birMonth)&&;
      (curDay > birDay)))
   {
      error = UFLSetFormula(tempNode, "PAGE1.CURRENTYEAR.value -
         PAGE1.BIRTHYEAR.value");
      if (error != OK)
      {
         fprintf(stderr, "UFLSetFormula error %hd.\n", error);
         return(NOTOK);
      }
   }
   else
   {
      error = UFLSetFormula(tempNode, "PAGE1.CURRENTYEAR.value -
         PAGE1.BIRTHYEAR.value - \"1\"");
      if (error != OK)
      {
         fprintf(stderr, "UFLSetFormula error %hd.\n", error);
         return(NOTOK);
      }
   }
   return(OK);
}
```

UFLSetLiteralByRefEx

Description

This function finds a particular formNodeP as specified by a reference string. Once the formNodeP is found, its literal will be set as specified. If the formNodeP does not exist, this function will create it, but only if the formNodeP would be an option or argument node.

If necessary, this function can create several nodes at once. For example, if you set the literal for the second argument of an *itemlocation*, this function will create the *itemlocation* option node and the two argument nodes and then set the literal for the second argument node.

This function cannot create a formNodeP at the form, page, or item level; to do so, use **UFLCreate**.

Note: It is not necessary to call this function when you are using XForms. The UFLReplaceXFormsInstance and UFLExtractXFormsInstancefunctions perform this task automatically.

Function

```
r_short UFLSetLiteralByRefEx(
   formNodeP aNode,
   r_charP theScheme,
   r_charP theReference,
   r_u_long theReferenceCode,
   r_charP theCharSet,
   formNodeP theNSNode,
   r_charP theLiteral);
```

Parameters

Expression	Type	Description
aNode	**formNodeP**	A **formNodeP** to use as a starting point for the search (unless an absolute reference is used).
theScheme	**r_charP**	Reserved. This must be NULL.
theReference	**r_charP**	A string that contains the reference.
theReferenceCode	**r_u_long**	Reserved. Must be 0.
theCharSet	**r_charP**	The character set in which *theLiteral* parameter is written. Use **NULL** or **ANSI** for ANSI. . Use **Symbol** for Symbol.
theNSNode	**formNodeP**	A node that is used to resolve the namespaces in *theReference* parameter (see "Determining Namespace" on page 103). Use NULL if the node that you are calling this function on has inherited the necessary namespaces.
theLiteral	**r_charP**	The string that will be assigned to the literal. If NULL, any existing literal is removed.

Returns

OK on success or an error code on failure.

Notes

formNodeP

Before you decide which formNodeP to use this function on, be sure you understand the following:

1. The formNodeP you supply can never be more than one level in the hierarchy above the level at which your reference string starts. For example, if the reference string begins with an option, then the formNodeP can be no higher in the hierarchy than an item.
2. If the formNodeP is at the same level or lower in the hierarchy than the starting point of the reference string, the function will attempt to locate a common ancestor. The function will locate the ancestor of the formNodeP that is one level in the hierarchy above the starting point of the reference string. The function will then attempt to follow the reference string back down through the hierarchy. If the reference string cannot be followed from the located ancestor (for example, if the ancestor is not common to both the formNodeP and the reference string), the function will fail. For example, given a formNodeP that represents ″field_1″ and a reference of ″field_2″, the function will access the ″page″ node above ″field_1″, and will then try to locate ″field_2″ below that node. If the two fields were not on the same page, the function would fail.

Creating a Reference String

For more information about creating a reference, see “References” on page 8.

Digital Signatures

Do not set a node that is digitally signed. Doing so will break the digital signature and produce an error.

Determining Namespace

In some cases, you may want to use the **UFLSetLiteralByRefEx** function to set the value for a node that does not have a globally defined namespace. For example, consider the following form:

```
<label sid="Label1">
   <value>Field1.processing:myValue</value>
</label>
<field sid="Field1" xmlns:processing="URI">
   <value></value>
   <processing:myValue>10<processing:myValue>
</field>
```

In this form, the *processing* namespace is declared in the *Field1* node. Any elements within *Field1* will understand that namespace; however, elements outside of the scope of *Field1* will not.

In cases like this, you will often start your search at a node that does not understand the namespace of the node you are trying to locate. For example, you might want to locate the node referenced in the value of *Label1*. In this case, you would first locate the *Label1* value node and get its literal. Then, from the *Label1* value node, you would attempt to locate the *processing:myValue* node as shown:

```
error = UFLSetLiteralByRefEx(Label1Node, NULL,
   "Field1.processing:myValue", 0, NULL, NULL, "20")
```

In this example, the **UFLSetLiteralByRefEx** function would fail. The function cannot properly resolve the *processing* namespace because this namespace is not defined for the *Label1* value node. To correct this, you must also provide a node that understands the *processing* namespace (in this case, any node in the scope of *Field1*) as a parameter in the function:

```
error = UFLSetLiteralByRefEx(Label1Node, NULL,
   "Field1.processing:myValue", 0, NULL, Field1Node, "20")
```

Example

In the original form, the label for the **Age** field instructs the user to leave the field blank. However, now that the field has been filled in by a formula, this label needs to be changed. In the following example **UFLSetLiteralByRefEx** is used to change this value.

```
r_error setFormula(formNodeP form, int curMonth, int curDay,
int birMonth, int birDay)
{
formNodeP tempNode=NULL;
r_error error=OK;

   /* Additional code removed */

   error = UFLSetLiteralByRefEx(form, NULL, "PAGE1.AGELABEL.value", 0,
      NULL, NULL, "Age:");
   if (error != OK)
   {
      fprintf(stderr, "UFLSetLiteralByRefEx error %hd.\n", error);
      return(NOTOK);
   }
   return(OK);
}
```

UFLSetLiteralEx

Description

This function sets the literal of a node. You should only set the literal for option or argument nodes.

Note: It is not necessary to call this function when you are using XForms. The UFLReplaceXFormsInstance and UFLExtractXFormsInstancefunctions perform this task automatically.

Function

```
r_short UFLSetLiteralEx(
   formNodeP aNode,
   r_charP theCharSet,
   r_charP theLiteral);
```

Parameters

Expression	Type	Description
aNode	**formNodeP**	The **formNodeP** to set the literal for.
theCharSet	**r_charP**	The character set in which *theLiteral* parameter is written. Use **NULL** or **ANSI** for ANSI. . Use **Symbol** for Symbol.
theLiteral	**r_charP**	The literal to assign to the node. If NULL, any existing literal is removed.

Returns

OK on success or an error code on failure.

Notes

Digital Signatures

Do not set the literal of a node that has already been signed. Doing so will break the digital signature and produce an error.

Example

In the following example, **UFLDereferenceEx** is used to locate a specific node. **UFLSetLiteralEx** is then used to change the literal of that node.

```
r_error changeNameLabel(formNodeP form, r_charP newName)
{
formNodeP tempNode=NULL;
r_error error=OK;

   if ((tempNode = UFLDereferenceEx(form, NULL,
      "PAGE1.NAMELABEL.value", 0, UFL_OPTION_REFERENCE | UFL_SEARCH,
      NULL)) == NULL)
   {
      fprintf(stderr, "Could not locate value node for NAMELABEL.\n");
      return(NOTOK);
   }
   error = UFLSetLiteralEx(tempNode, NULL, newName);
   if (error != OK)
   {
      fprintf(stderr, "UFLSetLiteralEx error %hd.\n");
      return(NOTOK);
   }
   return(OK);
}
```

UFLSignForm

Description

This function takes a button node and creates a digital signature for that button. The signature is created using the signature filter in the button and the private key of the signer.

This function returns a signature object that is tracked by reference counts. If you do not need to use the object, remember to release the reference to it.

Function

```
r_error UFLSignForm(
   formNodeP theButton,
   Certificate *theSigner,
   StringDictionary theInfo,
   SecurityUserStatusType *theStatus,
   Signature **theSignature);
```

Parameters

Expression	Type	Description
theButton	**formNodeP**	The form node that represents the signature button.
theSigner	**Certificate***	The certificate to use to create the signature.
theInfo	**StringDictionary**	Always use a NULL value.
theStatus	**SecurityUserStatusType***	This is a status flag that reports whether the operation was successful. Possible values are: **SUSTATUS_OK** — the operation was successful. **SUSTATUS_CANCELLED** — the operation was cancelled by the user. **SUSTATUS_INPUT_REQUIRED** — the operation required user input, but could not receive it (for example, it was run on a server with no user).
theSignature	**Signature****	The signature object that the function creates. Note that this object is tracked by reference counts, and must be released.

Returns

OK on success or an error code on failure.

Example

In the following example, **UFLDereferenceEx** is used to retrieve the node of a specific signature button. **UFLSignForm** is then used to create a signature object. Since we do not need to use the signature object, we then release the reference count to that object.

```
r_error createSignature(formNodeP form, Certificate *theSigner)
{
Signature *theSignature;
SecurityUserStatusType theStatus;
formNodeP buttonNode;
r_error error;

   if ((buttonNode = UFLDereferenceEx(form, NULL, "PAGE1.SIGNBUTTON",
      0, UFL_ITEM_REFERENCE, NULL)) == NULL)
   {
      fprintf(stderr, "Could not locate SIGNBUTTON node.\n");
      return(NOTOK);
   }
   if ((error = UFLSignForm(buttonNode, theSigner, NULL, &theStatus,
      &theSignature)) != OK)
   {
      fprintf(stderr, "UFLSignForm error %hd.\n", error);
      return(NOTOK);
   }

   /* Check the status in case the process required user input. */

   if (theStatus != SUSTATUS_OK)
   {
      fprintf(stderr, "User input required to sign form./n");
      return(NOTOK);
   }
```

```
    /* Release the signature object. */

    IFSObject_ReleaseRef((IFSObject*)theSignature);
    return(OK);
}
```

UFLUpdateXFormsInstance

Description

This function supplants replaceXFormsInstance. It allows developers to insert data anywhere within the XForms instance data, or replace it entirely. The instance can come from either from either a file or a memory block. The UFLUpdateXFormsInstance function automatically updates the XForms data model.

Use caution when calling this function. It can be used to overwrite signed instance data.

Note: This function automatically updates the XForms data model.

Function

```
r_short UFLUpdateXFormsInstance(
    formNodeP theObject,
    r_charP theModelID,
    r_charP theNodeRef,
    formNodeP *theNSNode,
    r_charP theFilename,
    r_byte *theMemoryBlock,
    r_long *theMemoryBlockSize
    r_u_long updateType);
```

Parameters

Expression	Type	Description
theObject	**formNodeP**	The formNodeP object.
theModelID	**r_charP**	The ID of the affected model. You must use NULL to use the default model.
theNodeRef	**r_charP**	An XPath reference to the instance (or portion of an instance) you want to insert data into or replace. An empty string indicates the default instance of the selected model.
theNSNode	**formNodeP**	A node that inherits the namespaces used in the reference. This node defines the namespaces for the function. Use NULL if the node that this function is operating on has inherited the necessary namespaces.
theFilename	**r_charP**	The file to read the instance data from. Note that if both a file and a memory block are provided, the file will take precedence.

Expression	Type	Description
theMemoryBlock	**r_byte**	The memory block that contains the instance if you are not reading from a file, input stream, or Reader. Use **NULL** if *theFilename* is used.
theMemoryBlock Size	**r_long**	The size of *theMemoryBlock*.
updateType	**r_u_long**	Indicates which type of update to perform: **XFORMS_UPDATE_REPLACE** — Replaces the specified data element. **XFORMS_UPDATE_APPEND** — Adds the data to the end of the specified data instance or element as a child element. **XFORMS_UPDATE_INSERT_BEFORE** — Adds the data as a sibling of the specified element. This sibling is placed before the specified element.

Returns

OK on success or an error code on failure.

Example

The following example shows a function that replaces an XForms instance.

```
r_error updateDataInstance(formNodeP theForm)
{
   if (UFLUpdateXFormsInstance(theForm, "model1",
      "instance('instance1')/loanrecord/user_personal_info", NULL,
      "c:\\InstanceData.xml", NULL, 0, OK, XFORMS_UPDATE_REPLACE)
      != OK)
   {
      fprintf(stderr, "Could not replace data instance.");
      return(NOTOK);
   }
   return(OK);
}
```

UFLValidateHMACWithSecret

Description

This function determines whether an HMAC signature is valid. HMAC signatures include both Authenticated Clickwrap and Signature Pad signatures.

For Authenticated Clickwrap signatures, you must know the signer's shared secret to use this function. For Signature Pad signatures, you may use this function without the shared secret if the signature was created without one. In any case, the shared secret should be available from a corporate database or other system.

This function will also notarize (that is, digitally sign) a valid HMAC signature if you provide a digital certificate. However, notarization will not occur if the signature does not include a shared secret. Once notarized, you must use the **UFLVerifySignature** function to validate the signature.

Note: Authenticated Clickwrap is a separately licensed product. Please ensure that your company has the license to use Authenticated Clickwrap before you provide forms or functionality that rely on it.

Function

```
r_error  UFLValidateHMACWithSecret(
   formNodeP theObject,
   r_charP theSecret,
   Certificate *theServerCert,
   SecurityUserStatusType *functionStatus,
   r_short *validateStatus);
```

Parameters

Expression	Type	Description
theObject	**formNodeP**	The node that represents the signature item.
theSecret	**r_charP**	The shared secret that identifies the user. This should be available from a corporate database or other system. If there is more than one shared secret, you must concatenate the strings with no separating characters. For example, if the secrets were ″blue″ and ″red″, you would pass ″bluered″ to the function. If there is no shared secret pass an empty string.
theServerCert	**Certificate***	The server certificate. If the HMAC signature is valid, the function will use the private key of this certificate to digitally sign the HMAC signature. This signature is appended to the signature item, and can be verified using UFLVerifySignature. If you pass NULL, the function will simply validate the HMAC signature.
functionStatus	**SecurityUserStatusType***	This is a status flag that reports whether the operation was successful. Possible values are: **SUSTATUS_OK** — the operation was successful. **SUSTATUS_CANCELLED** — the operation was cancelled by the user. **SUSTATUS_INPUT_REQUIRED** — the operation required user input, but could not receive it (for example, it was run on a server with no user).
validateStatus	**r_short***	A constant that indicates whether the HMAC signature is valid. See the *Returns* section for a complete list.

Returns

OK on success or an error code on failure.

Additionally, the *validateStatus* pointer will contain one of the following codes, depending on the status of the signature::

Code	Numeric Value	Status
UFL_DS_OK	0	The signature is verified.
UFL_DS_ALGORITHM UNAVAILABLE	13590	The appropriate verification engine for the signature is not available.
UFL_DS_F2MATCHSIGNER	13529	The certificate does not match the signer's name.
UFL_DS_FAILED AUTHENTICATION	1272	The signature is invalid or the secret used is incorrect.
UFL_DS_HASHCOMPFAILED	13527	The document has been tampered with.
UFL_DS_NOSIGNATURE	13526	There is no signature.
UFL_DS_NOTAUTHENTICATED	1240	The signer cannot be authenticated.
UFL_DS_UNEXPECTED	13589	An unexpected error occurred.
UFL_DS_UNVERIFIABLE	859	The signature cannot be verified.

Example

The following example uses **UFLGetSignature** to get the signature object, and uses **Signature_GetDataByPath** to get the signer's identity from the signature object. It then calls **UFLValidateHMACWithSecret** to validate the signature. Finally, it releases the signature object.

```
r_error checkSignature(formNodeP theSignatureNode, Certificate *theServerCert,
   r_short *validation)
{
Signature *theSignatureObject;
r_charP theSecret;
r_charP signerCommonName;
r_boolean encodedData;
SecurityUserStatusType theStatus;
r_error error;

   if ((error = UFLGetSignature(theSignatureNode, &theSignatureObject))
      != OK)
   {
      fprintf(stderr, "UFLGetSignature error %ld.\n", error);
      return(error);
   }
   if ((error = Signature_GetDataByPath(theSignatureObject,
      "SigningCert: Subject: CN", NOTOK, &encodedData,
      &signerCommonName)) != OK)
   {
      fprintf(stderr, "Signature_GetDataByPath error %ld./n", error);
      return(error);
   }

   /* Include external code that matches the signer's identity to a shared
      secret, and sets theSecret to match.  This is most likely a
      database lookup. */

   if ((error = UFLValidateHMACWithSecret(theSignatureNode, theSecret,
      theServerCert, &theStatus, validation)) != OK)
```

```
    {
       fprintf(stderr, "UFLValidateHMACWithSecret error %ld.\n", error);
       return(error);
    }

    /* Check the status in case the process required user input. */

    if (theStatus != SUSTATUS_OK)
    {
       fprintf(stderr, "User input required to sign form./n");
       return(NOTOK);
    }

    cp_free(signerCommonName);

    /* Release the reference to the signature object. */

    IFSObject_ReleaseRef((IFSObject*)theSignature);

    return(OK);
}
```

UFLValidateHMACWithHashedSecret

Description

This function determines whether an HMAC signature is valid. HMAC signatures include both Authenticated Clickwrap and Signature Pad signatures.

For Authenticated Clickwrap signatures, you must know the hash of the signer's shared secret to use this function. For Signature Pad signatures, you may use this function without the shared secret if the signature was created without one. In any case, the shared secret should be available from a corporate database or other system.

This function will also notarize (that is, digitally sign) a valid HMAC signature if you provide a digital certificate. However, notarization will not occur if the signature does not include a shared secret. Once notarized, you must use the **UFLVerifySignature** function to validate the signature.

Note: Authenticated Clickwrap is a separately licensed product. Please ensure that your company has the license to use Authenticated Clickwrap before you provide forms or functionality that rely on it.

Function

```
r_error  UFLValidateHMACWithHashedSecret(
   formNodeP theObject,
   r_byte *hashedSecret,
   r_long secretSize,
   Certificate *theCertificate,
   SecurityUserStatusType *theStatus,
   r_short *validateStatus);
```

Parameters

Expression	Type	Description
theObject	**formNodeP**	The node that represents the signature item.

Expression	Type	Description
hashedSecret	**r_byte***	The hash of the shared secret that identifies the user. This should be available from a corporate database or other system. If there is more than one shared secret, you must concatenate the strings with no separating characters and then hash the combined secret. For example, if the secrets were ″blue″ and ″red″, you would pass the hash of ″bluered″ to the function. If there is no shared secret, pass and empty string. You must encode the byte array as follows: **Authenticated Clickwrap (HMAC)** UTF-8 **Signature Pad** UTF-16LE The method for doing this depends on the C library you are using to interface with the API.
secretSize	**r_long**	The size of the hashed secret, measured in bytes.
theCertificate	**Certificate***	The server certificate. If the HMAC signature is valid, the function will use the private key of this certificate to digitally sign the HMAC signature. This signature is appended to the signature item, and can be verified using UFLVerifySignature. If you pass NULL, the function will simply validate the HMAC signature.
theStatus	**SecurityUserStatusType***	This is a status flag that reports whether the operation was successful. Possible values are: **SUSTATUS_OK** — the operation was successful. **SUSTATUS_CANCELLED** — the operation was cancelled by the user. **SUSTATUS_INPUT_REQUIRED** — the operation required user input, but could not receive it (for example, it was run on a server with no user).
validateStatus	**r_short***	A constant that indicates whether the HMAC signature is valid. See the *Returns* section for a complete list.

Returns

OK on success or an error code on failure.

Additionally, the *validateStatus* pointer will contain one of the following codes, depending on the status of the signature:

Code	Numeric Value	Status
UFL_DS_OK	0	The signature is verified.
UFL_DS_ALGORITHM UNAVAILABLE	13590	The appropriate verification engine for the signature is not available.
UFL_DS_F2MATCHSIGNER	13529	The certificate does not match the signer's name.
UFL_DS_FAILED AUTHENTICATION	1272	The signature is invalid or the secret used is incorrect.
UFL_DS_HASHCOMPFAILED	13527	The document has been tampered with.
UFL_DS_NOSIGNATURE	13526	There is no signature.
UFL_DS_NOT AUTHENTICATED	1240	The signer cannot be authenticated.
UFL_DS_UNEXPECTED	13589	An unexpected error occurred.
UFL_DS_UNVERIFIABLE	859	The signature cannot be verified.

Example

The following example uses **UFLGetSignature** to get a signature object, and uses **Signature_GetDataByPath** to get the signer's identity from the signature object. Next, it calls **UFLValidateHMACWithHashedSecret** to validate the signature. Finally, it releases the signature object.

```
r_error checkSignature(formNodeP theSignatureNode, Certificate *theServerCert,
   r_short *validation)
{
Signature *theSignatureObject;
r_byte *hashedSecret;
r_long secretSize;
r_boolean encodedData;
r_charP signerCommonName;
SecurityUserStatusType theStatus;
r_error error;

   if ((error = UFLGetSignature(theSignatureNode, &theSignatureObject))
      != OK)
   {
      fprintf(stderr, "UFLGetSignature error %ld.\n", error);
      return(error);
   }
   if ((error = Signature_GetDataByPath(theSignatureObject,
      "SigningCert: Subject: CN", NOTOK, &encodedData,
      &signerCommonName)) != OK)
   {
      fprintf(stderr, "Signature_GetDataByPath error %ld./n", error);
      return(error);
   }

   /* Include external code that matches the signer's identity to a hashed
      shared secret, sets *hashedSecret to match, and sets secretSize to
      the size of the hashed secret.  This is most likely a database
      lookup. */

   if ((error = UFLValidateHMACWithHashedSecret(theSignatureNode,
      hashedSecret, secretSize, theServerCert, &theStatus, validation))
```

```
      != OK)
   {
      fprintf(stderr, "UFLValidateHMACWithHashedSecret error %ld.\n",
         error);
      return(error);
   }

   /* Check the status in case the process required user input. */

   if (theStatus != SUSTATUS_OK)
   {
      fprintf(stderr, "User input required to sign form./n");
      return(NOTOK);
   }

   cp_free(signerCommonName);

   /* Release the reference to the signature object. */

   IFSObject_ReleaseRef((IFSObject*)theSignature);
   return(OK);
}
```

UFLVerifyAllSignatures

Description

This function verifies the correctness of all digital signatures in a given form whose root node is provided. It finds all items of type signature and calls **UFLVerifySignature** for each signature. Errors are logged for all invalid signatures.

This function checks the following conditions for each signature:

- The signature item contains mimedata.
- The mimedata contains a hash value and signer certificate.
- The signer certificate contains the same ID as that recorded in the signature item's *signer* option.
- The signer certificate has not expired.

Function

```
r_short UFLVerifyAllSignatures(
   formNodeP theForm,
   r_short reportAsErrorsFlag,
   r_short *validSigsFlagPtr);
```

Parameters

Expression	Type	Description
theForm	**formNodeP**	The form containing the signatures to verify.
reportAsErrorsFlag	**r_short**	Set to OK if you want errors about the signatures to be reported using the Error system, or NOTOK if you want the error code to be only returned through the *validSigsFlagPtr*.
validSigsFlagPtr	**r_short***	A pointer to a location that stores the result of the signature check. It will be set either to OK if all the signatures are valid or to NOTOK if at least one signature is invalid.

Returns

OK on success or NOTOK on failure.

Additionally, the *validSigsFlagPtr* will contain one of the following values:

Code	Status
UFL_SIGS_OK	The signatures are valid.
UFL_SIGS_NOTOK	One or more signatures are broken.
UFL_SIGS_UNVERIFIED	One or more signatures are unverifiable.

Example

In the following example, **UFLVerifyAllSignatures** determines whether or not all the signatures in the form are valid. If any one of the digital signatures is not valid, an error message is displayed.

```
r_error checkSignatures(formNodeP form)
{
r_error error;
r_short validFlag;

   error = UFLVerifyAllSignatures(form, OK, &validFlag);
   if (error != OK)
   {
      fprintf(stderr, "UFLVerifyAllSignatures error %hd.\n", error);
      return(NOTOK);
   }

/* Report an additional error if not all the signatures are valid. */

   if (validFlag != UFL_SIGS_OK)
   {
      fprintf(stderr, "Not all signatures are valid.\n");
   }
   return(OK);
}
```

UFLVerifySignature

Description

This function verifies the correctness of the given digital signature. You supply the root of the form that contains the signature you want to verify. This function checks the following conditions:

- The signature item contains mimedata.
- The mimedata contains a hash value and signer certificate.
- The signer certificate contains the same ID as that recorded in the signature item's *signer* option.
- The signer certificate has not expired.

A plain text representation of the form (filtered by the signature item's filter) is constructed and the result is hashed. This hash value must match the hash value stored in the signature.

Function

```
r_short UFLVerifySignature(
   formNodeP theForm,
   formNodeP signatureItem,
   r_charP *theCertChain,
   r_short reportAsErrorsFlag,
   r_short *validSigStatusPtr);
```

Parameters

Expression	Type	Description
theForm	**formNodeP**	The form containing the signature to verify.
signatureItem	**formNodeP**	The signature to verify.
theCertChain	**r_charP***	Reserved. Must be NULL.
reportAsErrorsFlag	**r_short**	Set to OK if you want errors about the signatures to be reported using the Error systemor NOTOK if you want the error code to be returned through the *validSigStatusPtr*.
validSigStatusPtr	**r_short***	A pointer to where to store whether the signature was valid. It will be set either to OK if the signature is valid or to an error code if the signature is invalid.

Returns

OK on success or NOTOK on failure.

Additionally, the *validSigStatusPtr* will contain one of the following values, depending on the status of the signature:

Code	Status
UFL_DS_OK	The signature is verified.
UFL_DS_ALGORITHMUNAVAILABLE	The appropriate verification engine for the signature is not available.
UFL_DS_CERTEXPIRED	The certificate has expired.
UFL_DS_CERTNOTFOUND	The certificate cannot be located.
UFL_DS_CERTNOTTRUSTED	The certificate is not trusted.
UFL_DS_CERTREVOKED	The certificate has been revoked.
UFL_DS_CRLINVALID	The certificate revocation list is invalid.
UFL_DS_F2MATCHSIGNER	The certificate does not match the signer's name.
UFL_DS_HASHCOMPFAILED	The document has been tampered with.
UFL_DS_ISSUERCERTEXPIRED	The issuer's certificate has expired.
UFL_DS_ISSUERINVALID	The issuer is invalid for the certificate used to sign.

Code	Status
UFL_DS_ISSUERKEYUSAGE UNACCEPTABLE	The issuer certificate's key usage extension does not match what the key was used for.
UFL_DS_ISSUERNOTCA	The certificate's issuer is not a Certificate Authority.
UFL_DS_ISSUERNOTFOUND	The issuer's certificate was not located.
UFL_DS_ISSUERSIGFAILED	Verification of the issuer's certificate failed.
UFL_DS_KEYREVOKED	The key used to create the signature has been revoked.
UFL_DS_KEYUSAGEUNACCEPTABLE	The certificate's key usage extension does not match what the key was used for.
UFL_DS_KRLINVALID	The Key Revocation List is invalid.
UFL_DS_NOSIGNATURE	There is no signature.
UFL_DS_NOTAUTHENTICATED	The signer cannot be authenticated.
UFL_DS_POLICYUNACCEPTABLE	The certificate's policy extension does not match the acceptable policies.
UFL_DS_SIGNATUREALTERED	The signature has been tampered with.
UFL_DS_UNEXPECTED	An unexpected error occurred.
UFL_DS_UNVERIFIABLE	The signature cannot be verified.

Example

In the following example, **UFLDereferenceEx** is used to locate a signature node. **UFLVerifySignature** is then used to determine whether or not the signature is valid. If the signature is not valid, a message is printed.

```
r_error checkSignature(formNodeP form)
{
formNodeP tempNode;
r_error error;
r_charP certChain;
r_short validFlag;

   if ((tempNode = UFLDereferenceEx(form, NULL, "PAGE1.SIGNATURE", 0,
      UFL_ITEM_REFERENCE, NULL)) == NULL)
   {
      fprintf(stderr, "Could not locate SIGNATURE node.\n");
      return(NOTOK);
   }
   error = UFLVerifySignature(form, tempNode, &certChain, OK,
      &validFlag);
   if (error != OK)
   {
      fprintf(stderr, "UFLVerifySignature error %hd.\n", error);
      return(NOTOK);
   }

/* Report an additional error if the signature is not valid. */
```

```
      if (validFlag == NOTOK)
      {
         fprintf(stderr, "Not all signatures are valid.\n");
      }

/* Free the memory associated with the chain of issuance. */

      cp_free(certChain);

      return(OK);
   }
```

UFLWriteForm

Description

This function will write a form to the specified file . The version number of the form determines the format of the output file. You can specify whether to compress the output file and whether to observe the *transmit* and *save* settings in the form.

If no format is specified, the default is to write the form in the same format in which it was read. If the form in question was created dynamically by your application, **UFLWriteForm** will, by default, write it as an XFDL form in uncompressed format.

Function

```
r_short UFLWriteForm(
   formNodeP theForm,
   r_charP theFilePath,
   formNodeP triggerItem,
   r_u_long flags);
```

Parameters

Expression	Type	Description
theForm	**formNodeP**	This is the root node of the form that should be written.
theFilePath	**r_charP**	This is the path to the file on the local disk to which the form will be written.
triggerItem	**formNodeP**	This is the item that caused the form to be submitted. Set to NULL if the API receives the form in a manner other than transmission.

Expression	Type	Description
flags	**r_u_long**	The following flags are valid: **UFL_TRANSMIT_ALLOW** allows the transmit options (that is, *transmitdatagroups*, *transmitgroups*, *transmititemrefs*, *transmititems*, *transmitoptionrefs*, *transmitpagerefs* and *transmitoptions*) to control which portions of the form are sent. Without this flag, the entire form will be sent regardless of the *transmit* options in the form. **UFL_SAVE_ALLOW** allows the *saveformat* option to specify what format the form should be saved in. If no format is specified then the form will be saved in the same format that it is read. **Note:** Specify 0 if you do not want to enable any of the transmit options.

Returns

OK on success or an error code on failure.

Example

The following example uses **UFLWriteForm** to write the form in memory to a file on the local drive.

```
error = UFLWriteForm(form, "output.xfd", NULL, 0);
   if (error != OK)
   {
      fprintf(stderr, "UFLWriteForm error %hd.\n", error);
      return(NOTOK);
   }
   return(OK);
}
```

UFLXMLModelUpdate

Description

This function updates the XML data model in the form. This is necessary if computes have changed the structure of the data model in some way, such as changing or adding bindings. These sorts of changes do not take effect until the **UFLXMLModelUpdate** function is called.

Function

```
r_short UFLXMLModelUpdate(
   formNodeP aNode);
```

Parameters

Expression	Type	Description
aNode	**formNodeP**	Any node in the form.

Returns

OK on success or an error code on failure.

Example

The following example uses **UFLSetLiteralByRefEx** to change a binding in the form, so that it binds to a different option. It then calls **UFLXMLModelUpdate** so that the data model reflects the change.

```
r_error setBinding(theForm)
{
   if (UFLSetLiteralByRefEx(theForm, NULL,
      "global.global.xmlmodel.bindings[0][boundoption]", 0, NULL, NULL,
      "PAGE1.FIELD5.value") == NOTOK)
   {
      fprintf(stderr, "Could not set literal value in binding.");
      return(NOTOK);
   }
   if (UFLXMLModelUpdate(theForm) == NOTOK)
   {
      fprintf(stderr, "Could not update XML model.");
      return(NOTOK)
   }
   return(OK)
}
```

The Hash Functions

The **Hash** functions allow you to hash messages.

- You must include the following header file in any .c source file that calls a Hash function:

```
#include "Hash.h"
```

Hash_Hash

Description

This function hashes a message using the hashing algorithm of your choice.

Function

```
r_error Hash_Hash(
   Hash *theHashObject,
   r_byte *theMessage,
   r_long messageSize,
   r_byte **hashedMessage,
   r_long *hashedSize);
```

Parameters

Expression	Type	Description
theHashObject	**Hash***	The hash object you are using to hash the message.
theMessage	**r_byte***	The message you want to hash.
messageSize	**r_long**	The size of the message you want to hash, in bytes.
hashedMessage	**r_byte****	The hashed message that is generated by the function. Note that this string must be freed.
hashedSize	**r_long***	The size of the hashed message, in bytes.

Returns

OK on success or an error code on failure.

Example

The following example uses **UFLGetSignature** and **Signature_GetDataByPath** to get the signer's identity from the signature object. It then retrieves the signer's shared secret from a database, and hashes that secret using the **Hash_Hash** function. Next, it calls **UFLValidateHMACWithHashedSecret** to validate the signature. Finally, it releases the signature object.

```
r_error checkSignature(formNodeP theSignatureNode, Certificate *theServerCert,
   Hash *theHashObject,
r_short *validation)
{
Signature *theSignatureObject;
r_charP theSecret;
r_charP signerCommonName;
r_boolean encodedData;
```

```
   r_byte *theSecret;
   r_long secretSize;
   r_byte *hashedSecret;
   r_long hashedSize;
   SecurityUserStatusType theStatus;
   r_error error;

'      if ((error = UFLGetSignature(theSignatureNode, &theSignatureObject))
          != OK)
       {
          fprintf(stderr, "UFLGetSignature error %ld.\n", error);
          return(error);
       }
       if ((error = Signature_GetDataByPath(theSignatureObject,
          "SigningCert: Subject: CN", NOTOK, &encodedData,
          &signerCommonName)) != OK)
       {
          fprintf(stderr, "Signature_GetDataByPath error %ld./n", error);
          return(error);
       }

       /* Include external code that matches the signer's identity to a
          shared secret, sets *theSecret to match, and sets secretSize to
          the size of the secret.  This is most likely a database
          lookup. */

       if ((error = Hash_Hash(theHashObject, theSecret, secretSize,
          &hashedSecret, &hashedSize)) != OK)
       {
          frprintf(stderr, "Hash_Hash error %hd./n", error);
          return(NOTOK);
       }

       if ((error = UFLValidateHMACWithHashedSecret(theSignature,
          hashedSecret, hashedSize, theServerCert, &theStatus, validation))
          != OK)
       {
          fprintf(stderr, "UFLValidateHMACWithHashedSecret error %hd.\n",
             error);
          return(NOTOK);
       }

       /* Check the status in case the process required user input. */

       if (theStatus != SUSTATUS_OK)
       {
          fprintf(stderr, "User input required to sign form./n");
          return(NOTOK);
       }

       cp_free(signerCommonName);
       cp_free(hashedSecret);

       /* Release the reference to the signature object. */

       IFSObject_ReleaseRef((IFSObject*)theSignature);

       return(OK);
    }
```

The Initialization Functions

The initialization functions provide an easy method for initializing the API.

- You must include the following header file in any .c source file that calls an Initialization Function.

  ```
  #include "xfdllib.h"
  ```

- Before using any of the Form functions you must first initialize the API using the **IFSInitialize** function.

Note: If you are using the Form functions from within the FCI functions then the API will have already been initialized.

IFSGetGlobalIFX

Description

This function retrieves the IFX Manager from the Forms System. Use this function to initialize your applications to work with extensions.

For more information about extensions refer to ″Introduction to the FCI Library″ .

Function

```
IFX *IFSGetGlobalIFX(void);
```

Parameters

There are no parameters for this function.

Returns

Returns a pointer to the Internet Forms Extension Manager referred to in this manual as the IFX Manager. Returns NULL on failure.

Example

```
IFX *getIFXManager(void *theInstance)
{
IFX *theIFX;
r_error theError;
      error = IFSInitialize("SampleAp", "1.0.0", "4.5.0");
      if (error != OK)
      {
         fprintf(stderr, "Could not retrieve IFX");
         return(NULL);
      }
      else
      {
         theIFX = IFSGetGlobalIFX();
         return(theIFX);
      }
}
```

IFSInitialize

Description

This function initializes the API. The parameters specify which version of the API your application should bind with (see the Notes below for more details).

You must call this function before calling any of the other functions in the API.

Function

```
r_error IFSInitialize(
   r_charP progName,
   r_charP progVer,
   r_charP apiVer);
```

Parameters

Expression	Type	Description
progName	**r_charP**	The name of the application calling **IFSInitialize**. This name is used to identify the application within the .ini file. It also sets the name that is returned by the XFDL **applicationName** function.
progVer	**r_charP**	The version number of the application calling **IFSInitialize**. If the .ini file has an entry for this version of the applicaton, the application will bind to the version of the API listed in that entry.
apiVer	**r_charP**	The version number of the API the application should use by default. If the .ini file does not contain an entry for the specific application, the application will bind to the API specified by this parameter.

Returns

OK on success or an error code on failure.

Notes

About Binding Your Applications to the API

When you initialize the API, the **IFSInitialize** function determines which version of the API to use based on the parameters you pass it. This allows you to exercise a great deal of control over which version of the API is used by your applications, and prevents the problems normally associated with common DLL files (often referred to as ″DLL hell″).

IFSInitialize uses a configuration file to determine which version of the API will bind to any application. This allows multiple versions of the API to co-exist on your computer, and ensures that your applications use the correct version of the API.

The configuration file is called PureEdgeAPI.ini and is installed with the API. Refer to the *IBM Workplace Forms Server - API Installation and Setup Guide* for the exact location of the file.

Note: You should redistribute the PureEdgeAPI.ini file with any applications that use the API. See the IBM Workplace Forms Server - API Installation and Setup Guide for more information about redistributing applications.

The configuration file contains a section for each application that might call the API, plus a default ″API″ section. Each section contains a list of version numbers in the following format:

```
<version of application> = <folder containing appropriate version of API>
```

For example, the configuration file might look like this:

```
[API]
5.1.0 = 51
5.0.0 = 50
[CustomApplication]
1.1.0 = 51
1.0.0 = 50
```

In this case, the folder indicated on the right hand side of each statement is part of the relative path to the API, and assumes the API was installed in the default folder. For example, under Windows ″50″ would resolve to:

```
c:\WinNT\System32\PureEdge\50
```

You can also specify an absolute path by placing a drive letter before the path. For example, ″c:\50″ would resolve to:

```
c:\50\
```

When you initialize the API, you include three parameters in the initialization call:

- The name of your application (as it would appear in the configuration file).
- The version of your application.
- The version of the API that your application should bind to by default.

The initialization call will first check the configuration file to see if your application is listed. For example, using the configuration file above, if you make an initialization call for ″CustomApplication″ version ″1.1.0″, then the application binds to the API in the ″51″ folder.

If your application is not listed in the configuration file, the initialization call will use the default version of the API. For example, using the configuration file above, if you declare ″5.1.0″ as the default API, then your application binds to the API in the ″51″ folder.

You can add your own entries to the configuration file before distributing it to your customers, or you can rely on the default API entries.

Note: **IFSInitialize** was introduced for version 4.5.0 of the API. Binding does not work in this manner for earlier versions of the API. Do not include earlier versions of the API in the configuration file.

Example

```
r_error initialize(void *theInstance)
{
r_error error;
      error = IFSInitialize("SampleAp", "1.0.0", "5.1.0");
      if (error != OK)
      {
         fprintf(stderr, "IFSInitialize error %hd.\n", error);
```

```
        return(NOTOK);
    }
    return(OK);
}
```

IFSInitializeWithLocale

Description

This function initializes the API. The parameters specify the default locale, and which version of the API your application should bind with (see the Notes below for more details).

You must call this function before calling any of the other functions in the API.

Function

```
r_error IFSInitializeWithLocale(
   r_charP progName,
   r_charP progVer,
   r_charP apiVer
   r_charP theLocale);
```

Parameters

Expression	Type	Description
progName	**r_charP**	The name of the application calling **IFSInitializeWithLocale**. This name is used to identify the application within the .ini file. It also sets the name that is returned by the XFDL **applicationName** function.
progVer	**r_charP**	The version number of the application calling **IFSInitializeWithLocale**. If the .ini file has an entry for this version of the applicaton, the application will bind to the version of the API listed in that entry.
apiVer	**r_charP**	The version number of the API the application should use by default. If the .ini file does not contain an entry for the specific application, the application will bind to the API specified by this parameter.
theLocale	**r_charP**	The default locale of the application.

Returns

OK on success or an error code on failure.

Notes

About Binding Your Applications to the API

When you initialize the API, the **IFSInitializeWithLocale** function determines which version of the API to use based on the parameters you pass it. This allows you to exercise a great deal of control over which version of the API is used by your applications, and prevents the problems normally associated with common DLL files (often referred to as “DLL hell”).

IFSInitializeWithLocale uses a configuration file to determine which version of the API will bind to any application. This allows multiple versions of the API to co-exist on your computer, and ensures that your applications use the correct version of the API.

The configuration file is called PureEdgeAPI.ini and is installed with the API. Refer to the *IBM Workplace Forms Server - API Installation and Setup Guide* for the exact location of the file.

Note: You should redistribute the PureEdgeAPI.ini file with any applications that use the API. See the *IBM Workplace Forms Server - API Installation and Setup Guid*e for more information about redistributing applications.

The configuration file contains a section for each application that might call the API, plus a default ″API″ section. Each section contains a list of version numbers in the following format:

```
<version of application> = <folder containing appropriate version of API>
```

For example, the configuration file might look like this:

```
[API]
2.6.1 = 70
2.6.0 = 70
[CustomApplication]
1.1.0 = 70
1.0.0 = 70
```

In this case, the folder indicated on the right hand side of each statement is part of the relative path to the API, and assumes the API was installed in the default folder. For example, under Windows “70” would resolve to:

```
C:\Program Files\IBM\Workplace Forms\Server\26\API\redist
   \msc32\PureEdge\26
```

You can also specify an absolute path by placing a drive letter before the path. For example, “c:\70” would resolve to:

```
c:\70\
```

When you initialize the API, you include three parameters in the initialization call:

- The name of your application (as it would appear in the configuration file).
- The version of your application.
- The version of the API that your application should bind to by default.

The initialization call will first check the configuration file to see if your application is listed. For example, using the configuration file above, if you make an initialization call for “CustomApplication” version “1.1.0”, then the application binds to the API in the “70” folder.

If your application is not listed in the configuration file, the initialization call will use the default version of the API. For example, using the configuration file above, if you declare “2.6.1” as the default API, then your application binds to the API in the “70” folder.

You can add your own entries to the configuration file before distributing it to your customers, or you can rely on the default API entries.

Example

```
   r_error initialize(void *theInstance)
   {
   r_error error;
      error = IFSInitializeWithLocale("SampleAp", "1.0 .0",
         "5.1.0", "fr-FR");
         if (error != OK)
         {
            fprintf(stderr, "IFSInitializeWithLocale error %hd.\n", error);
            return(NOTOK);
         }
      return(OK);
}
```

The LocalizationManager Functions

The **LocalizationManager** functions control which language the API uses to report errors.

- You must include the following header file in any .c source file that calls a LocalizationManager function:

```
#include "LocalizationManager.h"
```

LocalizationManager_GetCurrentThreadLocale

Description

This function returns which *locale* is in use for the current thread. This determines what language the API uses when reporting errors. By default, the API uses the default locale.

The API supports the following locales:

Language	Locale	Locale Name
Chinese	Simplified Han, China	zh-Hans-CN
	Simplified Han, Singapore	zh-Hans-SG
	Traditional Han, Hong Kong S.A.R., China	zh-Hant-HK
	Traditional Han, Taiwan	zh-Hant-TW
Croatian	Croatia	hr-HT
Czech	Czech Republic	cs-CZ
Danish	Denmark	da-DK
Dutch	Belgium	nl-BE
	The Netherlands	nl-NL
English	Australia	en-AU
	Belgium	en-BE
	Canada	en-Ca
	Hong Kong S.A.R., China	en-HK
	India	en-IN
	Ireland	en-IE
	New Zealand	en-NZ
	Philippines	en-PH
	Singapore	en-SG
	South Africa	en-ZA
	United Kingdom	en-GB
	United States	en-US
Finnish	Finland	fi-FI
French	Belgium	fr-BE
	Canada	fr-CA
	France	fr-FR

Language	Locale	Locale Name
	Luxembourg	fr-LU
	Switzerland	fr-CH
German	Austria	de-AT
	Germany	de-DE
	Luxembourg	de-LU
	Switzerland	de-CH
Greek	Greece	el-GR
Hungarian	Hungary	hu-HU
Italian	Italy	it-IT
	Switzerland	it-CH
Japanese	Japan	ja-JP
Korean	South Korea	ko-KR
Norwegian Bokmål	Norway	nb-NO
Polish	Poland	pl-PL
Portuguese	Brazil	pt-BR
	Portugal	pt-PT
Romanian	Romania	ro-RO
Russian	Russian	ru-RU
Slovak	Slovakia	sk-SK
Slovene	Slovenia	sl_SI
Spanish	Argentina	es-AR
	Bolivia	es-BO
	Chile	es-CL
	Colombia	es-CO
	Costa Rica	es-CR
	Dominican Republic	es-DO
	Ecuador	es-EC
	El Salvador	es-SV
	Guatemala	es-GT
	Honduras	es-HN
	Mexico	es-MX
	Nicaragua	es-NI
	Panama	es-PA
	Paraguay	es-PY
	Peru	es-PE
	Puerto Rico	es-PR
	Spain	es-ES
	United States	es-US
	Uruguay	es-UY
	Venezuela	es-VE
Swedish	Sweden	sv-SE

Language	Locale	Locale Name
Turkish	Turkey	tr-TR

The locale name consists of two parts: the language code and the country code, as shown

```
<language code>_<COUNTRY CODE>
```

For example, to specify the Japanese locale, you would type:

```
jp_JP
```

If you need a more specific locale, you may add additional codes after the country code. For example, to indicate French in France with a Euro dialect, you would type:

```
fr_FR_EU
```

Function

```
r_error LocalizationManager_GetCurrentThreadLocale(
   LocalizationManager *theObject,
   r_charP *returnPtr);
```

Parameters

Expression	Type	Description
theObject	**LocalizationManager**	The Localization Manager object.
returnPtr	**r_charP**	A pointer that will contain the returned value.

Returns

OK on success or an error code on failure.

Example

The following function calls **Localization Manager_GetCurrentThreadLocale** to get the locale.

```
r_charP getThreadLocale()
{
   r_error theError;
   r_charP theLocale;
   theLocale = NULL;
   if ((theError =
      LocalizationManager_GetCurrentThreadLocale(&theLocale)) != OK)
   {
      return NULL;
   }
   return theLocale;
}
```

LocalizationManager_GetDefaultLocale

Description

This function returns the default *locale* the API uses when reporting errors.

The API supports the following locales:

Language	Locale	Locale Name
Chinese	Simplified Han, China	zh-Hans-CN
	Simplified Han, Singapore	zh-Hans-SG
	Traditional Han, Hong Kong S.A.R., China	zh-Hant-HK
	Traditional Han, Taiwan	zh-Hant-TW
Croatian	Croatia	hr-HT
Czech	Czech Republic	cs-CZ
Danish	Denmark	da-DK
Dutch	Belgium	nl-BE
	The Netherlands	nl-NL
English	Australia	en-AU
	Belgium	en-BE
	Canada	en-Ca
	Hong Kong S.A.R., China	en-HK
	India	en-IN
	Ireland	en-IE
	New Zealand	en-NZ
	Philippines	en-PH
	Singapore	en-SG
	South Africa	en-ZA
	United Kingdom	en-GB
	United States	en-US
Finnish	Finland	fi-FI
French	Belgium	fr-BE
	Canada	fr-CA
	France	fr-FR
	Luxembourg	fr-LU
	Switzerland	fr-CH
German	Austria	de-AT
	Germany	de-DE
	Luxembourg	de-LU
	Switzerland	de-CH
Greek	Greece	el-GR
Hungarian	Hungary	hu-HU
Italian	Italy	it-IT
	Switzerland	it-CH
Japanese	Japan	ja-JP
Korean	South Korea	ko-KR
Norwegian Bokmål	Norway	nb-NO
Polish	Poland	pl-PL
Portuguese	Brazil	pt-BR

Language	Locale	Locale Name
	Portugal	pt-PT
Romanian	Romania	ro-RO
Russian	Russian	ru-RU
Slovak	Slovakia	sk-SK
Slovene	Slovenia	sl_SI
Spanish	Argentina	es-AR
	Bolivia	es-BO
	Chile	es-CL
	Colombia	es-CO
	Costa Rica	es-CR
	Dominican Republic	es-DO
	Ecuador	es-EC
	El Salvador	es-SV
	Guatemala	es-GT
	Honduras	es-HN
	Mexico	es-MX
	Nicaragua	es-NI
	Panama	es-PA
	Paraguay	es-PY
	Peru	es-PE
	Puerto Rico	es-PR
	Spain	es-ES
	United States	es-US
	Uruguay	es-UY
	Venezuela	es-VE
Swedish	Sweden	sv-SE
Turkish	Turkey	tr-TR

The locale name consists of two parts: the language code and the country code, as shown

```
<language code>_<COUNTRY CODE>
```

For example, to specify the Japanese locale, you would type:

```
jp_JP
```

If you need a more specific locale, you may add additional codes after the country code. For example, to indicate French in France with a Euro dialect, you would type:

```
fr_FR_EU
```

Function

```
r_error LocalizationManager_GetDefaultLocale(
   LocalizationManager *theObject,
   r_charP *returnPtr);
```

Parameters

Expression	Type	Description
theObject	**LocalizationManager**	The Localization Manager object.
returnPtr	**r_charP**	A pointer that will contain the returned value.

Returns

OK on success or an error code on failure.

Example

The following function calls **Localization Manger_GetDefaultLocale** to get the locale.

```
   r_charP getDefaultLocale()
{
      r_error theError;
      r_charP theLocale;
      theLocale = NULL;
      if ((theError =
         LocalizationManager_GetDefaultLocale(&theLocale)) != OK)
      {
         return NULL;
      }
      return theLocale;
   }
```

LocalizationManager_SetCurrentThreadLocale

Description

This function sets which *locale* the API uses when reporting errors. By default, the API uses the application's default locale.

The API supports the following locales:

Language	Locale	Locale Name
Chinese	Simplified Han, China	zh-Hans-CN
	Simplified Han, Singapore	zh-Hans-SG
	Traditional Han, Hong Kong S.A.R., China	zh-Hant-HK
	Traditional Han, Taiwan	zh-Hant-TW
Croatian	Croatia	hr-HT
Czech	Czech Republic	cs-CZ
Danish	Denmark	da-DK
Dutch	Belgium	nl-BE
	The Netherlands	nl-NL
English	Australia	en-AU
	Belgium	en-BE
	Canada	en-Ca
	Hong Kong S.A.R., China	en-HK
	India	en-IN

Language	Locale	Locale Name
	Ireland	en-IE
	New Zealand	en-NZ
	Philippines	en-PH
	Singapore	en-SG
	South Africa	en-ZA
	United Kingdom	en-GB
	United States	en-US
Finnish	Finland	fi-FI
French	Belgium	fr-BE
	Canada	fr-CA
	France	fr-FR
	Luxembourg	fr-LU
	Switzerland	fr-CH
German	Austria	de-AT
	Germany	de-DE
	Luxembourg	de-LU
	Switzerland	de-CH
Greek	Greece	el-GR
Hungarian	Hungary	hu-HU
Italian	Italy	it-IT
	Switzerland	it-CH
Japanese	Japan	ja-JP
Korean	South Korea	ko-KR
Norwegian Bokmål	Norway	nb-NO
Polish	Poland	pl-PL
Portuguese	Brazil	pt-BR
	Portugal	pt-PT
Romanian	Romania	ro-RO
Russian	Russian	ru-RU
Slovak	Slovakia	sk-SK
Slovene	Slovenia	sl_SI
Spanish	Argentina	es-AR
	Bolivia	es-BO
	Chile	es-CL
	Colombia	es-CO
	Costa Rica	es-CR
	Dominican Republic	es-DO
	Ecuador	es-EC
	El Salvador	es-SV
	Guatemala	es-GT
	Honduras	es-HN

Language	Locale	Locale Name
	Mexico	es-MX
	Nicaragua	es-NI
	Panama	es-PA
	Paraguay	es-PY
	Peru	es-PE
	Puerto Rico	es-PR
	Spain	es-ES
	United States	es-US
	Uruguay	es-UY
	Venezuela	es-VE
Swedish	Sweden	sv-SE
Turkish	Turkey	tr-TR

The locale name consists of two parts: the language code and the country code, as shown

```
<language code>_<COUNTRY CODE>
```

For example, to specify the Japanese locale, you would type:

```
jp_JP
```

If you need a more specific locale, you may add additional codes after the country code. For example, to indicate French in France with a Euro dialect, you would type:

```
fr_FR_EU
```

Function

```
r_error LocalizationManager_SetCurrentThreadLocale(
   LocalizationManager *theObject,
   r_charP theLocale);
```

Parameters

Expression	Type	Description
theObject	**r_charP**	The LocalizationManager object
theLocale	**r_charP**	The name of the locale.

Returns

OK on success or an error code on failure.

Example

The following function checks the *language* string to determine which locale to use. It then calls **Localization Manger_SetCurrentThreadLocale** to set the appropriate locale.

```
r_error setCurrentLanguage(r_charP language)
{
r_charP locale;
r_error theError;
```

```
   if (cp_strcmp(language, "english") == OK)
   {
      if ((theError = LocalizationManager_SetCurrentThreadLocale(
         "en_US")) != OK)
      {
         fprintf(stderr, "LocalizationManager_SetCurrentThreadLocale
            error %hd./n", error);
         return(NOTOK);
      }
   else
   {
      if ((theError = LocalizationManager_SetCurrentThreadLocale(
         "fr_CA")) != OK)
      {
         fprintf(stderr, "LocalizationManager_SetCurrentThreadLocale
            error %hd./n", error);
         return(NOTOK);
      }
   }
}
```

LocalizationManager_SetDefaultLocale

Description

This function sets the default *locale* the API uses when reporting errors, if no other locale is specified. By default, the API uses the locale specified by the operating system.

The API supports the following locales:

Language	Locale	Locale Name
Chinese	Simplified Han, China	zh-Hans-CN
	Simplified Han, Singapore	zh-Hans-SG
	Traditional Han, Hong Kong S.A.R., China	zh-Hant-HK
	Traditional Han, Taiwan	zh-Hant-TW
Croatian	Croatia	hr-HT
Czech	Czech Republic	cs-CZ
Danish	Denmark	da-DK
Dutch	Belgium	nl-BE
	The Netherlands	nl-NL
English	Australia	en-AU
	Belgium	en-BE
	Canada	en-Ca
	Hong Kong S.A.R., China	en-HK
	India	en-IN
	Ireland	en-IE
	New Zealand	en-NZ
	Philippines	en-PH
	Singapore	en-SG
	South Africa	en-ZA
	United Kingdom	en-GB

Language	Locale	Locale Name
	United States	en-US
Finnish	Finland	fi-FI
French	Belgium	fr-BE
	Canada	fr-CA
	France	fr-FR
	Luxembourg	fr-LU
	Switzerland	fr-CH
German	Austria	de-AT
	Germany	de-DE
	Luxembourg	de-LU
	Switzerland	de-CH
Greek	Greece	el-GR
Hungarian	Hungary	hu-HU
Italian	Italy	it-IT
	Switzerland	it-CH
Japanese	Japan	ja-JP
Korean	South Korea	ko-KR
Norwegian Bokmål	Norway	nb-NO
Polish	Poland	pl-PL
Portuguese	Brazil	pt-BR
	Portugal	pt-PT
Romanian	Romania	ro-RO
Russian	Russian	ru-RU
Slovak	Slovakia	sk-SK
Slovene	Slovenia	sl_SI
Spanish	Argentina	es-AR
	Bolivia	es-BO
	Chile	es-CL
	Colombia	es-CO
	Costa Rica	es-CR
	Dominican Republic	es-DO
	Ecuador	es-EC
	El Salvador	es-SV
	Guatemala	es-GT
	Honduras	es-HN
	Mexico	es-MX
	Nicaragua	es-NI
	Panama	es-PA
	Paraguay	es-PY
	Peru	es-PE
	Puerto Rico	es-PR

Language	Locale	Locale Name
	Spain	es-ES
	United States	es-US
	Uruguay	es-UY
	Venezuela	es-VE
Swedish	Sweden	sv-SE
Turkish	Turkey	tr-TR

The locale name consists of two parts: the language code and the country code, as shown

```
<language code>_<COUNTRY CODE>
```

For example, to specify the Japanese locale, you would type:

```
jp_JP
```

If you need a more specific locale, you may add additional codes after the country code. For example, to indicate French in France with a Euro dialect, you would type:

```
fr_FR_EU
```

Function

```
r_error LocalizationManager_SetDefaultLocale(
   r_charP theLocale);
```

Parameters

Expression	Type	Description
theLocale	**r_charP**	The name of the locale.

Returns

OK on success or an error code on failure.

Example

The following function checks the *language* string to determine which locale to use. It then calls **Localization Manger_SetDefaultLocale** to set the appropriate locale.

```
r_error setDefaultLanguage(r_charP language)
{
r_charP locale;
r_error theError;

   if (cp_strcmp(language, "english") == OK)
   {
      if ((theError = LocalizationManager_SetDefaultLocale("en_US")) !=
         OK)
      {
         fprintf(stderr, "LocalizationManager_SetDefaultLocale
            error %hd./n", error);
         return(NOTOK);
      }
   }
   else
   {
```

```
      if ((theError = LocalizationManager_SetDefaultLocale("fr_CA")) !=
         OK)
      {
         fprintf(stderr, "LocalizationManager_SetDefaultLocale
            error %hd./n", error);
         return(NOTOK);
      }
   }
}
```

The SecurityManager Functions

The SecurityManager functions allow you to retrieve the Security Manager and obtain a hashing algorithm.

- You must include the following header file in any .c source file that calls a Security function:

```
#include "SecurityManager.h"
```

SecurityManager_GetSingleton

Description

This function retrieves the *Security Manager* object. Use the *Security Manager* object to retrieve the available hash algorithms.

Note that the *Security Manager* object is tracked by reference counts, and must be released.

Function

```
r_error SecurityManager_GetSingleton(
   SecurityManager **theSecurityManager);
```

Parameters

Expression	Type	Description
theSecurityManager	**SecurityManager****	The Security Manager object that the function retrieves. Note that this object is tracked by reference counts, and must be released.

Returns

OK on success or an error code on failure.

Example

The following example uses **SecurityManager_GetSingleton** to get the *Security Manager* object. The example then calls **SecurityManager_LookupHashAlgorithm** to get the *sha1* hash algorithm. Finally, the *SecurityManager* object is released. (Note that the hash object is not released, as it is passed back to the calling function.)

```
r_error getHashAlgorithm(Hash **theHashObject);
{
SecurityManger *theSecurityManager;
Hash *tempHashObject;
r_error error;
      *theHashObject = NULL;
      if((error = SecurityManager_GetSingleton(&theSecurityManager))
         != OK)
      {
         fprintf(stderr, "SecurityManager_GetSingleton error %hd./n",
            error);
         return(NOTOK);
      }
```

```
      if((error = SecurityManager_LookupHashAlgorithm(theSecurityManager,
         "sha1", &tempHashObject)) != OK)
      {
         fprintf(stderr, "SecurityManager_LookupHashAlgorithm error %hd./n),
            error);
         return(NOTOK);
      }
      *theHashObject = tempHashObject;
      /* Relase the Security Manager object. */
      IFSObject_ReleaseRef((IFSObject*)theSecurityManager);
      return(OK);
}
```

SecurityManager_LookupHashAlgorithm

Description

This function retrieves a hash object. Use the hash object to hash shared secrets for the **UFLValidateHMACWithHashedSecret** function.

Note that the hash object is tracked by reference counts, and must be released.

Function

```
r_error SecurityManager_LookupHashAlgorithm(
   SecurityManager *theSecurityManager,
   r_charP algorithmName,
   Hash **theHash);
```

Parameters

Expression	Type	Description
theSecurityManager	**SecurityManager***	The Security Manager object.
algorithmName	**r_charP**	The name of the hash algorithm you want to retrieve. The available hash algorithms are *sha1* and *md5*.
theHashObject	**Hash****	The hash algorithm object that the function retrieves. Note that this object is tracked by reference counts, and must be released.

Returns

OK on success or an error code on failure.

Example

The following example uses **SecurityManager_GetSingleton** to get the *Security Manager* object. The example then calls **SecurityManager_LookupHashAlgorithm** to get the *sha1* hash algorithm. Finally, the *SecurityManager* object is released. (Note that the hash object is not released as it is passed back to the calling function.)

```
r_error getHashAlgorithm(Hash **theHashObject);
{
SecurityManager *theSecurityManager;
Hash *tempHashObject;
r_error error;

*theHashObject = NULL;
   if((error = SecurityManager_GetSingleton(&theSecurityManager))
      != OK)
```

```
   {
      fprintf(stderr, "SecurityManager_GetSingleton error %hd./n",
         error);
      return(NOTOK);
   }
   if((error = SecurityManager_LookupHashAlgorithm(theSecurityManager,
      "sha1", &tempHashObject)) != OK)
   {
      fprintf(stderr, "SecurityManager_LookupHashAlgorithm error %hd.
         /n"), error);
      return(NOTOK);
   }

   *theHashObject = tempHashObject;

   /* Relase the Security Manager object. */

   IFSObject_ReleaseRef((IFSObject*)theSecurityManager);
   return(OK);
}
```

The Signature Functions

The **Signature** functions allow you to work with signature objects.

- You must include the following header file in any .c source file that calls a Signature function:

```
#include "Signature.h"
```

Signature_GetDataByPath

Description

This function retrieves a piece of data from a signature object.

Function

```
r_error Signature_GetDataByPath(
   Signature *theSignature,
   r_charP thePath,
   r_boolean tagData,
   r_boolean *encoded,
   r_charP *theData);
```

Parameters

Expression	Type	Description
theSignature	**Signature***	The signature object you want to query.
thePath	**r_charP**	The path to the data you want to retrieve. See the Notes section below for more information on data paths.
tagData	**r_boolean**	OK if the path should be prepended to the data, or NOTOK if not. If the path is prepended, an equals sign (=) is used as a separator. For example, suppose the path is ″Signing Cert: Issuer: CN″ and the data is ″IBM″. If OK, the path will be prepended, producing ″CN=IBM″. If NOTOK, the path will not be prepended, and the result will be ″IBM″.
encoded	**r_boolean***	OK if the return data is base 64 encoded, or NOTOK if not. The function returns binary data in base 64 encoding.
theData	**r_charP***	The data that the function locates. This is **NULL** if no data is found. Note that this string must be freed.

Notes

About Data Paths

Data paths describe the location of information within a signature, just like file paths describe the location of files on a disk. You describe the path with a series of

colon separated tags. Each tag represents either a piece of data, or an object that contains further pieces of data (just like directories can contain files and subdirectories).

For example, to retrieve the version of a signature, you would use the following data path:

```
Demographics
```

However, to retrieve the signer's common name, you first need to locate the signing certificate, then the subject, then finally the common name within the subject, as follows:

```
SigningCert: Subject: CN
```

Some tags may contain more than one piece of information. For example, the issuer's organizational unit may contain a number of entries. You can either retrieve all of the entries as a comma separated list, or you can specify a specific entry by using a zero-indexed element number.

For example, the following path would retrieve a comma separated list:

```
SigningCert: Issuer: OU
```

Adding an element number of 0 would retrieve the first organizational unit in the list, as shown:

```
SingingCert: Issuer: OU: 0
```

Signature Tags

The following table lists the tags available in a signature object. Note that Clickwrap and HMAC Clickwrap signatures have additional tags (detailed in Clickwrap Signature Tags and HMAC Clickwrap Tags).

Tag	Description
Engine	The security engine used to create the signature. This is an object that contains further information, as detailed in *Security Engine Tags*.
SigningCert	The certificate used to create the signature. This is an object that contains further information, as detailed in *Certificate Tags*. Note that this object does not exist for Clickwrap or HMAC Clickwrap signatures.
HashAlg	The hash algorithm used to create the signature.
CreateDate	The date on which the signature was created.
Demographics	A string describing the signature.
LastVerificationStatus	A short representing the verification status of the signature. This is updated whenever the signature is verified. See "UFLVerifySignature" on page 115 for a complete list of the possible values.

Clickwrap Signature Tags

The following table lists additional tags available in both Clickwrap and HMAC Clickwrap signatures. Note that HMAC Clickwrap signatures have further tags (detailed in HMAC Clickwrap Tags).

Tag	Description
TitleText	The text for the Windows title bar of the signature dialog box.
MainPrompt	The text for the title portion of the signature dialog box.
MainText	The text for the text portion of the signature dialog box.
Question1Text	The first question in the signature dialog box.
Answer1Text	The signer's answer.
Question2Text	The second question in the signature dialog box.
Answer2Text	The signer's answer.
Question3Text	The third question in the signature dialog box.
Answer3Text	The signer's answer.
Question4Text	The fourth question in the signature dialog box.
Answer4Text	The signer's answer.
Question5Text	The fifth question in the signature dialog box.
Answer5Text	The signer's answer.
EchoPrompt	Text that the signer must echo to create a signature.
EchoText	The signer's response to the echo text.
ButtonPrompt	The text that provides instructions for the Clickwrap signature buttons.
AcceptText	The text for the accept signature button.
RejectText	The text for the reject signature button.

Certificate Tags

The following table lists the tags available in a certificate object. Note that Clickwrap and HMAC Clickwrap signatures do not contain these tags.

Tag	Description
Subject	The subject's distinguished name. This is an object that contains further information, as detailed in Distinguished Name Tags.
Issuer	The issuer's distinguished name. This is an object that contains further information, as detailed in Distinguished Name Tags.
IssuerCert	The issuer's certificate. This is an object that contains the complete list of certificate tags.
Engine	The security engine that generated the certificate. This is an object that contains further information, as detailed in Security Engine Tags.
Version	The certificate version.
BeginDate	The date on which the certificate became valid.
EndDate	The date on which the certificate expires.
Serial	The certificate's serial number.
SignatureAlg	The signature algorithm used to sign the certificate.
PublicKey	The certificate's public key.
FriendlyName	The certificate's friendly name.

Distinguished Name Tags

The following table lists the tags available in a distinguished name object. Note that Clickwrap and HMAC Clickwrap signatures do not contain these tags.

Tag	Description
CN	The common name.
E	The e-mail address.
T	The title.
O	The organization.
OU	The organizational unit.
C	The country.
L	The locality.
ST	The state.
All	The entire distinguished name.

HMAC Clickwrap Tags

The following table lists the tags available in HMAC Clickwrap signature. Note that these tags are in addition to both the regular Signature Tags and the Clickwrap Signature Tags.

Tag	Description
HMACSigner	A string indicating which answers store the signer's ID.
HMACSecret	A string indicating which answers store the signer's secret.
Notarization	The notarizing signatures. This is one or more signature objects that contain further information, as detailed in Signature Tags . There can be any number of notarizing signatures. Use an element number to retrieve a specific signature. For example, to get the first notarizing signature use: `Notarization: 0` If no element number is provided, the data will be retrieved from the first valid notarizing signature found. If no valid notarizing signatures are found, the function will return NULL.

Security Engine Tags

The following table lists the tags available in the security engine object:

Tag	Description
Name	The name of the security engine used by the server.
Help	The help text for the security engine.
HashAlg	A hash algorithm supported by the security engine.

Returns

OK on success or an error code on failure.

Example

The following example uses **UFLGetSignature** to get the signature object from the signature node, and uses **Signature_GetDataByPath** to get the signer's identity from the signature object. It then calls **UFLValidateHMACWithSecret** to validate the signature. Finally, it releases the signature object.

```
r_error checkSignature(formNodeP theSignatureNode, Certificate *theServerCert,
   r_short *validation)
{
Signature *theSignatureObject;
r_charP theSecret;
r_charP signerCommonName;
r_boolean encodedData;
SecurityUserStatusType theStatus;
r_error error;

   if ((error = UFLGetSignature(theSignatureNode, &theSignatureObject))
      != OK)
   {
      fprintf(stderr, "UFLGetSignature error %ld.\n", error);
      return(error);
   }
   if ((error = Signature_GetDataByPath(theSignatureObject,
      "SigningCert: Subject: CN", NOTOK, &encodedData,
      &signerCommonName)) != OK)
   {
      fprintf(stderr, "Signature_GetDataByPath error %ld./n", error);
      return(error);
   }
   /* Include external code that matches the signer's identity to a shared
      secret, and sets theSecret to match.  This is most likely a
      database lookup. */

   if ((error = UFLValidateHMACWithSecret(theSignatureNode, theSecret,
      theServerCert, &theStatus, validation)) != OK)
   {
      fprintf(stderr, "UFLValidateHMACWithSecret error %ld.\n", error);
      return(error);
   }

   /* Check the status in case the process required user input. */

   if (theStatus != SUSTATUS_OK)
   {
      fprintf(stderr, "User input required to sign form./n");
      return(NOTOK);
   }

   cp_free(signerCommonName);

   /* Release the reference to the signature object. */

   IFSObject_ReleaseRef((IFSObject*)theSignature);

   return(OK);
}
```

Signature_GetSigningCert

Description

This function retrieves the signing certificate from a signature object.

Function

```
r_error Signature_GetSigningCert(
   Signature *theSignature,
   Certificate **theCertificate);
```

Parameters

Expression	Type	Description
theSignature	**Signature***	The signature from which you want to extract the signing certificate.
theCertificate	**Certificate****	A pointer that is set to the signing certificate.

Returns

OK on success or an error code on failure.

Example

The following example gets the signing certificate from a signature object, then iterates through the certificate issuers until it reaches the end of the chain. During the iteration, each certificate is passed to a function that processes them.

```
r_error processCertChain(Signature *theSig)
{
Certificate *theCert, *issuerCert;
SecurityUserStatusType theStatus;

   /* Get the signing certificate from the signature. */

   if (Signature_GetSigningCert(theSig, &theCert) != OK)
   {
      fprintf(stderr, "Could not get signing certificate.\n");
      return(NOTOK);
   }

   /* Loop through the certificate chain, passing each certificate to the
      ProcessCert function. The loop ends when the issuer certificate is
      NULL. */

   while (theCert != NULL)
   {

      /* Pass the certificate to the processCert function.  Note that
         this is not an API function, but rather a function you would
         write to process the certificate in some way. */

      if (ProcessCert(theCert) != OK)
      {
         fprintf(stderr, "Could not process certificate.\n");
         return(NOTOK);
      }

      /* Get the issuer certificate from theCert. */

      if (Certificate_GetIssuer(theCert, &theStatus, &issuerCert) != OK)
```

```
      {
         fprintf(stderr, "Could not get issuer certificate.\n");
         return(NOTOK);
      }

      /* Check to ensure the function exited with the correct status. */

      if (theStatus != SUSTATUS_OK)
      {
         fprintf(stderr, "GetIssuer exited with wrong status.\n");
         return(NOTOK);
      }

      /* Free theCert object. */

      IFSObject_ReleaseRef(theCert);

      /* Assign theCert to equal the issuerCert for next iteration of the
         loop. */

      theCert = issuerCert;
   }
   return(OK);
}
```

The Utility Functions

The API includes some utility functions that perform a variety of tasks. These include the string and memory functions detailed earlier in this manual, as well as the functions listed in this section.

- To use the functions listed in this section in an application, include the following header file as the first include in any .c source file that calls these functions.

```
#include "masqutil.h"
```

Refer to ″The API Memory and String Functions″ for a complete description of the string and memory functions.

MUGetThreadSafeFlag

Description

This function will query the current application to determine whether it is thread safe. This is particularly useful if you are developing an extension, since some applications may not support multi-thread extensions.

Refer to ″Introduction to the FCI Library″ for more information on creating an extension.

Function

```
r_error MUGetThreadSafeFlag(
   r_boolean *theFlagPtr);
```

Parameters

Expression	Type	Description
*theFlagPtr	**r_boolean**	A pointer to the thread safe flag. A value of OK means the application is thread safe; NOTOK means the application is not thread safe.

Returns

OK on success or an error code on failure.

Example

In the following example, **MUGetThreadSafeFlag** is used to determine which function should be called, depending on whether the current application is thread safe.

```
r_error SelectRoutine();
{
r_boolean AppIsThreadSafe;
r_error theError;
      if((theError = MUGetThreadSafeFlag(&AppIsThreadSafe)) != OK)
      {
         fprintf(stderr, "MUGetThreadSafeFlag error %hd./n", error);
         return(NOTOK);
      }
```

```
      if (AppIsThreadSafe == OK)
         ThreadSafeRoutine();
      else
         NotThreadSafeRoutine();
}
```

The XFDL Functions

The **XFDL** functions create the root nodes of forms and handle administrative tasks related to the Form Library.

- You must include the following header file in any .c source file that calls an XFDL function:

```
#include "xfdllib.h"
```

UFLCreate

Description

This function creates a new formNodeP and attaches it to the form hierarchy at the indicated location. Once created, the type and identifier of a formNodeP cannot be changed.

Note that you can also use **UFLSetLiteralByRefEx** to create a **formNodeP** at the option level and below. Using **UFLSetLiteralByRefEx** is often easier and faster than using **UFLCreate**.

Function

```
formNodeP UFLCreate(
   formNodeP aNode,
   r_short where,
   r_charP theType,
   r_charP theLiteral,
   r_charP theFormula,
   r_charP theIdentifier);
```

Parameters

Expression	Type	Description
aNode	**formNodeP**	The new formNodeP is placed in the form hierarchy in relation to this node. If NULL, this creates a new formNodeP hierarchy (a new form)
where	r_short	A constant that describes the location, in relation to the parameter *aNode*, in which the new node should be placed: **UFL_APPEND_CHILD** — adds the new node as the last child of aNode. **UFL_AFTER_SIBLING** — adds the new node as a sibling of *aNode*, placing it immediately after that node. **UFL_BEFORE_SIBLING —** adds the new node as a sibling of *aNode*, placing it immediately before that node. Note: If the parameter *aNode* is NULL, then this parameter should be set to 0.

Expression	Type	Description

theType	**r_charP**	The type to assign to the formNodeP being created. This is only necessary for page and item nodes. Use NULL for all other nodes. The type cannot be changed after the node has been created.
		
		If you are creating a non-XFDL node, you must also include the namespace that the node should belong to, as shown:
		`<namespace prefix>:<type>`
		
		For example:
		`custom:myItem`
		
		If you do not provide a namespace, the function will assign the default namespace for the form.

theLiteral	**r_charP**	The literal to assign to this formNodeP. NULL is valid.
theFormula	**r_charP**	The formula to assign to this formNodeP. NULL is valid.

theIdentifier	**r_charP**	The identifier to assign to this formNodeP. The identifier cannot be changed after the node has been created. NULL is valid.
		
		If you are creating an option or argument level node, this must also include the namespace the node should belong to. Use the following format:
		`<namespace prefix>:<type>`
		
		For example:
		`custom:myOption`
		
		If you do not provide a namespace, the function will assign the default namespace for the form.

Returns

The new **formNodeP**.

Example

In the following example, **UFLDereferenceEx** is used to locate a specific node. **UFLCreate** is then used to create a sibling to that node and to place it directly after that node in the form structure.

```
r_error addPicLabel(formNodeP form)
{
formNodeP tempNode;

   if ((tempNode = UFLDereferenceEx(form, NULL, "PAGE1.GENDERLABEL", 0,
      UFL_ITEM_REFERENCE, NULL)) == NULL)
   {
      fprintf(stderr, "Could not locate GENDERLABEL node.\n");
      return(NOTOK);
   }
   if((tempNode = UFLCreate(tempNode, UFL_AFTER_SIBLING, "label", NULL,
      NULL, "PICLABEL")) == NULL)
   {
      fprintf(stderr, "Could not create PICLABEL node.\n");
```

```
        return(NOTOK);
    }
    return(OK);
}
```

UFLGetEngineCertificateList

Description

This function locates all available certificates for a particular signing engine.

Note that each certificate in the list is tracked by reference counts. Once you are done with the certificates, you must release the reference counts to the certificates and free the array (see the example for details).

Function

```
r_error UFLGetEngineCertificateList(
    r_charP engineName,
    SecurityUserStatusType *theStatus,
    Certificate ***theCertificate,
    r_long *certCount);
```

Parameters

Expression	Type	Description
engineName	**r_charP**	The name of the signing engine. Valid signing engines include: Generic RSA, CryptoAPI, Netscape, and Entrust. (Note that Generic RSA is the union of CryptoAPI and Netscape.)
theStatus	**SecurityUserStatusType***	This is a status flag that reports whether the operation was successful. Possible values are: **SUSTATUS_OK** — the operation was successful. **SUSTATUS_CANCELLED** — the operation was cancelled by the user. **SUSTATUS_INPUT_REQUIRED** — the operation required user input, but could not receive it (for example, it was run on a server with no user).
certList	**Certificate*** **	The list of certificates that the function locates. Note that each certificate object is tracked by reference counts, and must be released. Furthermore, the array must also be freed.
certCount	**r_long***	The number of certificates that the funtion located.

Returns

OK on success or an error code on failure.

Example

The following function uses **UFLGetEngineCertificateList** to get a list of valid certificates for the CryptoAPI signing engine. Next, the function cycles through the

returned certificates and uses **Certificate_GetDataByPath** to find the certificate with a common name of "Workplace Forms Server". **Signature_GetDataByPath** is then used to retrieve the common name from the existing signature, which is used to retrieve a shared secret from a database. The function then uses **UFLValidateHMACWithSecret** to validate the signature and notarize it using the server certificate. Finally, the example frees the memory.

```
r_error serverNotarize(formNodeP theSignatureNode, r_short *validation)
{
SecurityUserStatusType theStatus;
Certificate **certList;
Signature *theSignatureObject;
r_charP signerCommonName;
r_charP theSecret;
r_charP signerCommonName;
SecurityUserStatusType theStatus;
r_boolean encodedData;
r_long certCount;
r_long correctCert = -1;
r_error error;
r_long i;

   if ((error = UFLGetEngineCertificateList("CryptoAPI", &theStatus,
      &certList, &certCount)) != OK)
   {
      fprintf(stderr, "UFLGetCertificateList error %hd.\n", error);
      return(NOTOK);
   }

   /* Check the status, in case the process required user input. */

   if (theStatus != SUSTATUS_OK)
   {
      fprintf(stderr, "User input required to retrieve certificate list.
         /n");
      return(NOTOK);
   }

   /* Loop through the certificates to find the Workplace Forms Server
      certificate */

   for (i=0; i<certCount; i++)
   {
      if ((error = Certificate_GetDataByPath(certList[i],
         "SigningCert: Subject: CN", NOTOK, &encodedData,
         &signerCommonName)) != OK)
      {
         fprintf(stderr, "Certificate_GetDataByPath error %hd./n",
            error);
         return(NOTOK);
      }
      if (cp_strcmp(signerCommonName, "Workplace Forms Server") == OK)
      {
         correctCert = i;
         cp_free(signerCommonName);
         break;
      }
      cp_free(signerCommonName);
   }
   if (correctCert == -1)
   {
      fprintf(stderr, "Could not locate required certificate.");
      return(NOTOK);
   }

   /* Get the signature object. */
```

```
   if ((error = UFLGetSignature(theSignatureNode, &theSignatureObject))
      != OK)
   {
      fprintf(stderr, "UFLGetSignature error %ld.\n", error);
      return(error);
   }

   /* Get the signer's common name from the signature object */

   if ((error = Signature_GetDataByPath(theSignatureObject,
      "SigningCert: Subject: CN", NOTOK, &encodedData,
      &signerCommonName)) != OK)
   {
      fprintf(stderr, "Signature_GetDataByPath error %ld./n", error);
      return(error);
   }

   /* Include external code that matches the signer's identity to a shared
      secret, and sets theSecret to match.  This is most likely a
      database lookup. */

   /* Validate the signature and notarize using the server certificate */

   if ((error = UFLValidateHMACWithSecret(theSignatureNode, theSecret,
      certList[correctCert], &theStatus, validation)) != OK)
   {
      fprintf(stderr, "UFLValidateHMACWithSecret error %ld.\n", error);
      return(error);
   }

   /* Check the status in case the process required user input. */

   if (theStatus != SUSTATUS_OK)
   {
      fprintf(stderr, "User input required to sign form./n");
      return(NOTOK);
   }
   cp_free(signerCommonName);

   /* Release each certificate object in the array */

   for(i=0; i<certCount; i++)
   {
      IFSObject_ReleaseRef((IFSObject*)certList[i]);
   }

   /* Free the array */

   pe_free(certList);

   /* Release the signature object */

   IFSObject_ReleaseRef((IFSObject*)theSignature);

   return(OK);
}
```

UFLIsDigitalSignaturesAvailable

Description

This function determines whether digital signatures are available on the current computer.

Function

```
r_short UFLIsDigitalSignaturesAvailable(
r_short *availableFlagPtr);
```

Parameters

Expression	Type	Description
availableFlagPtr	**r_short***	A pointer that contains **OK** if digital signatures are available and **NOTOK** if digital signatures are not available.

Returns

OK on success or an error code on failure.

Example

In the following example, **UFLDereferenceEx** is used to locate the digital signature button in the form. **UFLIsDigitalSignaturesAvailable** is then used to determine whether or not digital signatures are supported; if they aren't, the signature button is removed.

```
r_error checkSignatureButton(formNodeP form)
{
r_error error;
r_short sigsAvail;
formNodeP tempNode;

   error = UFLIsDigitalSignaturesAvailable(&sigsAvail);
   if (error != OK)
   {
      fprintf(stderr, "UFLIsDigitalSignaturesAvailable error %hd.\n",
         error);
      return(NOTOK);
   }
   if (sigsAvail == NOTOK)
   {
      if ((tempNode = UFLDereferenceEx(form, NULL, "PAGE1.SIGNBUTTON",
         0, UFL_ITEM_REFERENCE, NULL)) == NULL)
      {
         fprintf(stderr, "Could not locate SIGNBUTTON node.\n");
         return(NOTOK);
      }
      error = UFLDestroy(tempNode);
      if (error != OK)
      {
         fprintf(stderr, "UFLDestroy error %hd.\n", error);
         return(NOTOK);
      }
   }
   return(OK);
}
```

UFLReadForm

Description

This function will read a form into memory from a specified file .

Function

```
formNodeP UFLReadForm(
   r_charP theFilePath,
   r_u_long flags);
```

Parameters

Expression	Type	Description
theFilePath	**r_charP**	The path to the source file on the local disk.
flags	**r_u_long**	The following flags cause special behaviors. If using multiple flags, combine them using a bitwise OR. For example: `UFL_AUTOCOMPUTE_OFF | UFL_AUTOCREATE_FORMATS_OFF` **0** — no special behavior. **UFL_AUTOCOMPUTE_OFF** — Reads the form into memory, but disables the compute system so that no computes are evaluated. **UFL_AUTOCREATE_CONTROLLED_OFF** — Reads the form into memory, but disables the creation of all options that are maintained only in memory (for example, itemnext, itemprevious, pagenext, pageprevious, and so on). **UFL_AUTOCREATE_FORMATS_OFF** — Reads the form into memory, but disables the evaluation of all format options. **UFL_SERVER_SPEED_FLAGS** — Turns off the following features: computes, automatic formatting, duplicate sid detection, the event model, and relative page and item tags (for example, itemprevious, itemnext, and so on). This setting significantly improves server processing times. **UFL_XFORMS_INITIALIZE_ONLY** — Turns off the following features: controlled item construction, UI connection to the XForms model, action handling set up, and the rebuild/recalculate/revalidate/refresh sequence after instance replacements.

Returns

A new **formNodeP**.

Notes

Duplicate Scope IDs

If a form contains duplicate scope IDs (for example, two items on the same page with the same SID), **UFLReadForm** will fail to read the form and will return an error. This enforces correct XFDL syntax, and eliminates certain security risks that exist when duplicate scope IDs appear in signed forms.

Digital Signatures

When a form containing one or more digital signatures is read, the signatures will be verified. The result of the verification is stored in a flag that can be checked by calling **UFLGetSignatureVerificationStatus**.

Note that this flag is only set by **UFLReadForm**, and its value will not be adjusted by changes made to the form after it has been read. This means that calls such as **UFLSetLiteralEx** may actually break a signature (by changing the value of a signed item), but that this will not adjust the flag's value. To verify a signature after changes have been made to a form, it is best to use **UFLVerifyAllSignatures**.

Note that when a form is signed, all signed computes are frozen at their start value (regardless of whether the compute system is disabled).

Server-Side Processing

Using the UFL_SERVER_SPEED_FLAGS setting significantly improves performance during server-side processing. We strongly recommend you use this flag if you do not require computes to update while processing the form.

Example

In the following example, **UFLReadForm** is used to load a form from a file on the local disk.

```
r_error loadForm(formNodeP *form)
{
   if ((*form = UFLReadForm("sample.xfd", 0)) == NULL)
   {
      fprintf(stderr, "Could not load form.\n");
      return(NOTOK);
   }
   return(OK);
}
```

Introduction to the FCI Library

The Function Call Interface (FCI) library provides a means for creating extremely powerful form applications in a simple and elegant manner.

The FCI Library is a collection of functions for developing custom-built functions that form developers may call from XFDL forms. By creating custom functions, you can extend the capabilities of forms without requiring an upgrade to either your forms software or the form description language (XFDL). Using the functions from the FCI Library, you can:

- Create packages of functions for forms.
- Set up the packages as extensions for Workplace Forms products such as Viewer or Designer.
- Determine how and when the functions are used. For example, you can specWorkplace Formsify that a function should run when a form opens, when it closes and so on.

About Functions, Packages, and Extensions

The purpose of the Function Call Interface is to make the functionality of forms extensible without requiring updates to your forms driver software. This library of functions allows you to create self-contained modules called *extensions* that provide *packages* of *functions* for use in XFDL forms.

Note: The forms driver software is any application that initializes and calls on the API.

Functions can be used almost anywhere in a form; the appropriateness of their use depends mainly on their behavior. For instance the *XFDL Specification* contains a default package of functions called system. Every application built with the API version 4.4 or greater can use these functions.

Functions are grouped together to form packages. When you call a function from a form, you must include the function's package name in the call. For example, the function **multiply** is part of the package called sample_package. To call the **multiply** function from a form and assign the result to the form option *do_multiply* you would type the following:

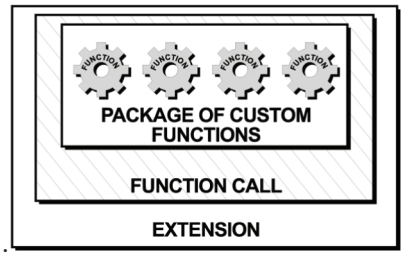
```
<label sid =  "do_multiply">
      <value compute = "sample_package.multiply(field1.value, &#xA;
         field2.value)">
      </value>
</label>
```

The most common use of a function is to return a value that is used to set a form option, such as the *value* of a field. For example, the **toupper** function in the system package, which converts a string to upper case and returns the result, might be used to set the *value* of a particular form field. This function could take as its sole argument the value of a label elsewhere on the form (or on another form) and convert it to upper case as follows:

```
<label sid = "SomeLabel">
      <value>"I am a label"</value>
</label>
<field sid = "SomeField">
```

```
      <size>
         <ae>20</ae>
         <ae>1</ae>
      </size>
      <value compute = "system.toupper(SomeLabel.value)"></value>
</field>
```

To create a package of functions you must create a *Forms extension* or *extension*. Refer to ″The FCI Extension Architecture″ for more information. Or, for a practical guide to building your own extensions and functions refer to ″Getting Started with the FCI Library″



Use the following rules to help you define your own packages and extensions:

- Each package can contain multiple functions.
- Each extension can contain multiple packages, however it is easier to define one package per extension.
- All package names must contain an underscore. IBM reserves all other package names. Refer to ″Package Naming Conventions″ for more information.
- The *XFDL Specifications* contains a default package of functions called **system**. Every application built with version 4.4 or greater of the API can use these functions.
- You cannot add to the system package of functions. For details on the system functions, see the *XFDL Specification*.

To distribute the functions, you compile the project as a DLL. This DLL is the extension. In order for your forms to access the functions in the extension, you need to distribute it with the Workplace Forms products. Refer to ″Distributing Extensions″ for more information.

About the Function Call Interface (FCI)

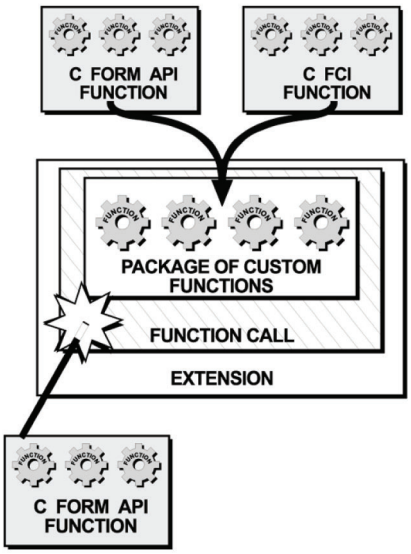
This FCI Library of functions allows you to create an extension structure that contains one or more packages of your custom functions.

How the Form and FCI Libraries Work Together

The Form Library provides developers with tools for accessing and manipulating XFDL forms as structured data types. For instance, functions in the Form Library will provide your applications with a means for reading and writing forms, retrieving information contained in form elements or assigning information to the elements of a form. For more information about the Form Library refer to “Introduction to the Form Library” on page 15.

The FCI Library of functions allows you to create an extension structure that contains one or more packages of your custom functions.

Once you have set up the framework for your custom functions you can use standard ANSI C functions, Form Library functions or even other FCI functions to implement the details of each function that you create.

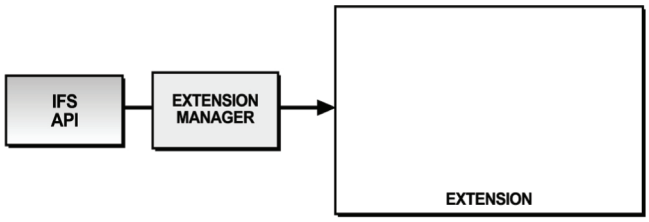


The FCI Extension Architecture

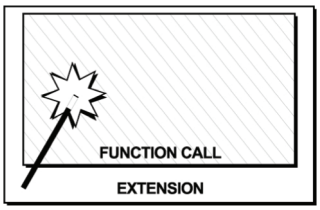
Extensions can exist in any of the following locations:

- The extensions folder of the Workplace Forms product that will use the extension (for example the Viewer or the Designer).
- The API extensions folder, <Windows System>\PureEdge\Extensions.

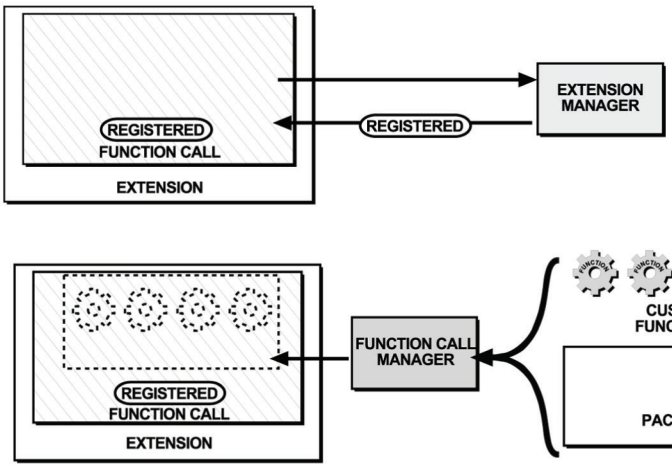
When the Forms System is initialized, the API checks for any existing extensions. If it finds any, it calls the initialization function for each extension and passes them a structure called the *IFX Manager*.



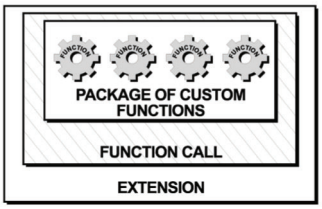
As part of the initialization, those extensions that provide a function call interface create one or more *Function Call objects*.



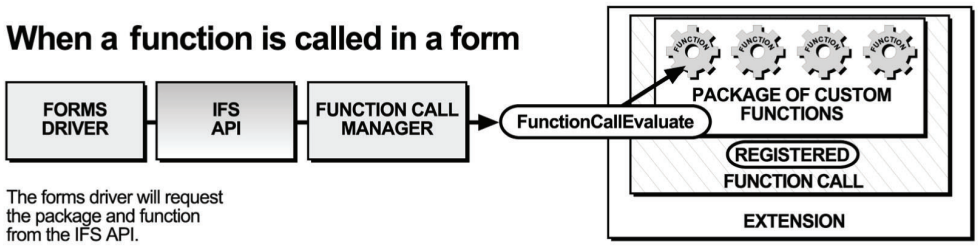
Each Function Call object registers itself with the IFX Manager as a function call and then registers your custom-built functions and corresponding packages with the *Function Call Manager*.



The final result is an extension containing a registered Function Call object. The registered Function Call object contains your package of custom functions.



When a function is called in a form, the forms driver requests the package and function from the API. The API will use the Function Call Manager to locate the Function Call object that contains the requested function and evaluate it.



Note: The forms driver software is any application that initializes and calls on the API.

Getting Started with the FCI Library

This section acts as both a reference and a tutorial on the Function Call Interface library. A series of practical examples is provided which you may work through to build an extension called **Sample Extension**. The extension contains a package of functions called **sample_package**, which contains one function called **multiply.** The **multiply** function multiplies the value of two fields together and stores the result in a third field. Try adding other functions to the package for more practice using the FCI Library.

Although the FCI Library contains many functions, you only need to use a few of them to create a simple package of functions. These are:

- C_ExtensionInit
- IFSObject_AllocateObject
- FunctionCall_SetObjectProc
- IFXRegisterInterface
- FCMRegisterFunctionCall
- FunctionCallEvaluate
- FunctionCallHelp

The remaining FCI functions allow you to customize the behavior of your functions and extensions. For example you can attach additional information to a particular extension, or get a list of currently registered extensions.

Refer to ″FCI Library Quick Reference Guide″ for a detailed description of the functions used in this API.

Note: Before you can build extensions using the FCI functions, you must set up your development environment. Refer to the IBM Workplace Forms Server - API Installation and Setup Guide for more information.

Creating Extensions with the FCI Functions

The following table is a guide for creating extensions using the FCI Library of functions. Refer to the corresponding page numbers for more details:

Procedure	Page
Install the API and related files, as outlined in the IBM Workplace Forms Server - API Installation and Se*tup Guide*.	N/A
Set up the extension.	Setting Up the Extension
Create the extension source file.	Creating the Extension Source File
Set up the extension initialization function.	Setting Up the Extension Initialization Function
Create **C_ExtensionInit**.	Creating C_ExtensionInit
Create a new Function Call structure.	Creating a New FunctionCall Structure
Define the services of the Function Call structure.	Defining Services Provided by the FunctionCall Structure

Procedure	Page
Register each Function Call structure with the IFX Manager.	Registering a FunctionCall with the IFX Manager
Register your package(s) of custom functions with the Forms System.	Registering your packages of custom functions using the Function Call Manager
Implement your custom functions.	Implementing your custom functions
Provide help information for each of your functions.	Providing help information for each of your functions
Compile the extension.	Compiling Your Extension
Test the extension.	Testing the Extension
Distribute the extension.	Distributing Extensions

Setting Up the Extension

A set of template project files for FCI applications have been provided for you. Copy the template files into the directory in which you will be building your application.

- Files that are compatible with the Visual C++ development environment can be found in the <*API Program folder*>\Samples\Msc32\Fci\Template.
- If you prefer to setup the FCI application development environment yourself, follow the procedure outlined in ″Appendix A: Setting Up Your Development Environment″ .

Creating the Extension Source File

When the Forms System is initialized, the API checks for any existing extensions and calls the initialization function (**C_ExtensionInit**). Your first step in creating a custom function is to set up and initialize **C_ExtensionInit**.

1. Create a new C source file called fciExtension.c.
2. Include the following preprocessor directives:

```
#include "masqutil.h"
#include "IFX.h"
#include "xfdllib.h"
#include "Extension.h"
#include "FunctionCall.h"
#include "FunctionCallManager.h"
```

Note: masqutil.h must always be the first include in any source file that uses the Form Library or the FCI Library.

Setting Up the Extension Initialization Function

Creating C_ExtensionInit

The API will initialize an extension by calling the function **C_ExtensionInit** and passing it a pointer to the extension and a pointer to the IFX Manager.

Set up the **C_ExtensionInit** function within your C source file.

- **C_ExtensionInit** is responsible for the registration of all the services that the extension provides.

- In the following example the **C_ExtensionInit** function for **fciExtension.c** is declared and passed two pointers and
- is the current extension you are initializing.
- is the IFX Manager structure. Through this structure all other objects and services can be reached.

```
#ifndef OLD_STYLE_PARAMS
PRE_FUNCTION_DECL r_short POST_FUNCTION_DECL C_ExtensionInit(
      Extension *theExtension, IFX *theIFXManager)
#else
PRE_FUNCTION_DECL r_short POST_FUNCTION_DECL C_ExtensionInit(
      theExtension, theIFXManager)
Extension *theExtension;
IFX *theIFXManager;
#endif
{
      /* Additional code removed */
}
```

Creating a New FunctionCall Structure

C_ExtensionInit will have to create a FunctionCall structure that contains your custom-built functions.

1. Declare a pointer to a new FunctionCall structure before you create it in the **C_ExtensionInit** function. Also, declare any additional variables that you will need in **C_ExtensionInit**.

   ```
   FunctionCall *theFunctionCall;
   r_short theError;
   IFXCriteriaMatchingHandler   *returnPtr;
   ```

2. Create a new FunctionCall structure inside **C_ExtensionInit** by calling the function, **IFSObject_AllocateObject**. A generic IFSObject will be created and a generic IFSObject pointer will be returned. Cast the returned pointer to a FunctionCall pointer.
   - In the following example, **C_ExtensionInit** creates a new **FunctionCall** structure by calling the **IFSObject_AllocateObject** function. The pointer returned from the call to **IFSObject_AllocateObject** is cast to a pointer to a **FunctionCall** structure.

     ```
     if ((theFunctionCall = (FunctionCall*)IFSObject_AllocateObject(
     FUNCTIONCALL_INTERFACE_NAME, FUNCTIONCALL_CURRENT_VERSION, NOTOK, 0))
     == NULL)
     return(NOTOK);
     ```

Note: For more about the **IFSObject_AllocateObject** function refer to "IFSObject_AllocateObject" on page 185.

Defining Services Provided by the FunctionCall Structure

The FunctionCall structure you have created will provide two services or functions.

The first service is the implementation of your custom functions. A forward declaration for this service was declared previously as the function FCISimpleFunctions. This function tells the forms driver what to do when custom functions are called from within forms.

The second service defines the help available to form designers when working with your custom functions in Workplace Forms compliant products, such as the Designer. A forward declaration for this service was declared previously as the function FCISimpleHelp.

Use the function called **FunctionCall_SetObjectProc** to define the services offered by the FunctionCall structure.

- This function is called once for each service being defined, and is passed the following parameters:
- The **FunctionCall** structure.
- The function that implements the service.
- The type of service being defined
- In the following example **FunctionCall_SetObjectProc** is called twice. The first call defines the services offered by **FCISimpleFunctions.** And the second call defines the services offered by **FCISimpleHelp.**

```
if ((theError = FunctionCall_SetObjectProc(theFunctionCall,
      (voidP)FCISimpleFunctions, FUNCTIONCALLEVALUATE)) != OK)
      return(theError);
if ((theError = FunctionCall_SetObjectProc(theFunctionCall,
      (voidP)FCISimpleHelp, FUNCTIONCALLHELP)) != OK)
      return(theError);
```

Note: For more information about the **FunctionCall_SetObjectProc** function refer to page FunctionCall_SetObjectProc.

Registering a FunctionCall with the IFX Manager

Each FunctionCall structure must be registered with the IFX Manager as an interface that provides function call support.

C_ExtensionInit, registers the FunctionCall structure with the IFX Manager using the function **IFXRegisterInterface**.

- The last parameter in the call to **IFXRegisterInterface** is a pointer to an object of type **IFXCriteriaMatchingHandler**. An object of this type can be generated by calling the function **FCMGetDefaultListener**.
- In the following example **theFunctionCall** is registered with the IFX Manager **theIFXManager**.

```
/*    Get the default listener */
if ((returnPtr = FCMGetDefaultListener()) == NULL)
      return(NOTOK);
/*    Register the interface with the Extention Manager.*/
if ((theError = IFXRegisterInterface(theIFXManager,
      (GenericInterface*)theFunctionCall, FUNCTIONCALL_INTERFACE_NAME,
      FUNCTIONCALL_CURRENT_VERSION, FUNCTIONCALL_MIN_VERSION_SUPPORTED,
      0X01000300L, 0, NULL, 0, returnPtr)) != OK)
      return(theError);
```

Note: For more information about the **IFXRegisterInterface** function refer to page IFXRegisterInterface.

Registering your packages of custom functions using the Function Call Manager

Call the Function Call Manager function **FCMRegisterFunctionCall** from **C_ExtensionInit** to register each of your custom functions and corresponding package(s) with the Forms System.

- The FCI allows you to assign a version number to each function that you create. This allows you to provide upgrades to single functions in extensions you have already distributed to users. For more information see 171.
- When registering your package(s) of functions with the Function Call Manager, be aware of the API package naming conventions. For more information see 171.
- You must register each of your custom functions separately. So, if you are registering three functions with the Function Call Manager, you must call **FCMRegisterFunctionCall** three times.
- In the following example **C_ExtensionInit** uses the **FCMRegisterFunctionCall** function to register the **multiply** function with the Function Call Manager:

```
if ((theError = FCMRegisterFunctionCall(theFunctionCall, PACKAGE_NAME,
      "multiply", FCI_MULTIPLY_ID, FCI_FOLLOWS_STRICT_CALLING_PARAMETERS,
      "S,S", FCI_MULTIPLY_VERSION, FCI_MULTIPLY_DESCRIPTION)) != OK)
      return(theError);
      /* Finished */
      return(OK);
}
```

Note: For more information about the **FCMRegisterFunctionCall** function refer to page FCMRegisterFunctionCall.

About Function Version Numbers

Along with registering your package(s) of custom functions with the Function Call Manager, the **FCMRegisterFunctionCall** function is also used to specify a version number for each function that you create. In the previous example the multiply function is registered with the version number 0x01000300.

Assigning a version number to each function allows you to provide upgrades to single functions in extensions you have already distributed to users.

For example, if you distributed an extension containing a package of 50 functions for your application and then wanted to change the behavior of one of the functions you could:

- Write a new extension containing just the upgraded function.
- Register the new function using **FCMRegisterFunctionCall**, with the same package name and function name as the original function but with a higher version number.
- Distribute the new extension to users.

When the API initialized all of the extensions it would find two functions with the same package name and function name. It would de-register the one with the lower version number thereby updating your application.

Note: For more information about using version numbers refer to ″Defining a Version Number″ .

Package Naming Conventions

The main purpose of package names is to distinguish the functions in a package from those in other packages that could potentially have the same names. All packages you create must contain an underscore in their names. For example the **beep** function belongs to a package called **my_funcs**.

- Choose a name that aptly describes the set of functions you are creating and that is distinct enough to be unique within its realm of usage.

- Since the package name is an internal logical element of the application, the 8-character naming restriction and case-insensitivity of Windows files do not exist.

Note: A group of functions is provided with the Forms System software as the system package. The system package is reserved for system functions that are defined in the XFDL Specification. You may not add to the system package or call your packages by the name system. Instead, ensure your package names contain an underscore.

Implementing your custom functions

Implement your custom functions as part of the extension.

- The function defined previously as **FCISimpleFunctions** implements the behavior of your custom functions.
- **FCISimpleFunctions** can be passed several commands by the Forms driver. When a function call in a form is to be evaluated, **FCISimpleFunctions** will be passed the command **FCICOMMAND_RUN** as a value for the parameter *theCommand*.
- This function must ensure that proper error checking is implemented. The function should check that the package and function names have been properly defined, and that the package name supplied by the Forms driver matches the package name specified in your extension. If you plan to write a multi-thread function, you should also check that the application using the is thread safe (using **MUGetThreadSafeFlag**).
- In the following example, the **Multiply** function is implemented as part of **FCISimpleFunctions** in **fciExtension.c**:

```
#ifndef OLD_STYLE_PARAMS
PRE_FUNCTION_DECL r_short POST_FUNCTION_DECL FCISimpleFunctions(FunctionCall
   *theObject,

r_charP thePackageName, r_charP theFunctionName, r_u_long theFunctionID,
   r_long theFunctionInstance,
r_short theCommand, formNodeP theForm, formNodeP theComputeNode, IFSUserData

**theFunctionDataPtr, IFSUserData **theFunctionInstanceDataPtr, formNodeP
   *theArgList, r_long
theArgListSize, formNodeP theResult)
#else
PRE_FUNCTION_DECL r_short POST_FUNCTION_DECL FCISimpleFunctions(theObject,
thePackageName, theFunctionName,theFunctionID, theFunctionInstance,theCommand,
   theForm,
theComputeNode, theArgList, theArgListSize, theResult)
FunctionCallManager *theObject;
r_charP thePackageName;
r_charP theFunctionName;
r_u_long theFunctionID;
r_long theFunctionInstance;
r_short theCommand;
formNodeP theForm;
formNodeP theComputeNode;
IFSUserData **theFunctionDataPtr;
IFSUserData **theFunctionInstanceDataPtr;
formNodeP *theArgList;
r_long *theArgListSize;
formNodeP theResult;
#endif
{
r_charP theFirstParam;
double firstNum;
r_charP theSecondParam;
```

```
double secondNum;
r_short theError;
double theMultiplyResult;
char theLiteralBuffer[50];
/*    Verify the input parameters */
      if ((thePackageName == NULL) || (theFunctionName == NULL))
      {
         MessageBox(NULL, "Invalid parameters", "Error", MB_OK);
         return(NOTOK);
      }
/*    Make sure that the package is the one we are expecting. */
      if (cp_strcmp(thePackageName, PACKAGE_NAME) != OK)
      {
         MessageBox(NULL, "This function was called with the wrong
            package name", "Error", MB_OK);
         return(NOTOK);
      }
/*    First switch based on the command */
      switch (theCommand)
      {
         case FCICOMMAND_RUN:
/*    Now switch based on the function name */
            switch (theFunctionID)
            {
/*    The "multiply" function. */
               case FCI_MULTIPLY_ID:

                  if ((theError = UFLGetLiteralEx(theArgList[0],
                     NULL, &theFirstParam)) != OK)
                  {
                     MessageBox(NULL, "Unable to get the first
                        parameter.", "Error", MB_OK);
                     return(theError);
                  }
                  if ((theError = UFLGetLiteralEx(theArgList[1],
                     NULL, &theSecondParam)) != OK)
                  {
                     MessageBox(NULL, "Unable to get the first
                        parameter.", "Error", MB_OK);
                     return(theError);
                  }

/*    Since the literal can be NULL, we make sure that it isn't */
                  if (theFirstParam == NULL)
                     theFirstParam = "";
                  if (theSecondParam == NULL)
                     theSecondParam = "";

/*    Convert the literal into a number */
                  firstNum = atof(theFirstParam);
                  secondNum = atof(theSecondParam);
/*    Now multiply the numbers */
                  theMultiplyResult = firstNum * secondNum;
                  sprintf(theLiteralBuffer, "%f", theMultiplyResult);
                  UFLSetLiteralEx(theResult, NULL, theLiteralBuffer);
                  break;
/*    Error if the function is unknown. */
               default:
                  MessageBox(NULL, "The function was called with an
                     unrecognized function", "Error", MB_OK);
               return(NOTOK);
            }
            break;
         case FCICOMMAND_INSTANCEREGISTER:
            break;
         case FCICOMMAND_INSTANCEDEREGISTER:
            break;
```

```
            case FCICOMMAND_REGISTER:
               break;
            case FCICOMMAND_DEREGISTER:
               break;
            default:
               break;
         }
/*       Finished */
         return(OK);
}
```

Note: For more information about the **FunctionCallEvaluate** template refer to "Function Call Evaluate Template" on page 187.

Providing help information for each of your functions

By using the function template FunctionCallHelp, you can provide help information to form designers within a development environment (for example the Designer). Use FunctionCallHelp to help form designers choose and use the correct functions.

Provide in depth help information for each of the functions you create by implementing the function FCISimpleHelp.

- **FCISimpleHelp** must be implemented since the service was defined previously with the call to **FunctionCall_SetObjectProc**.
- In the following example, **FCISimpleHelp** provides help information for the **multiply** function.

```
#ifndef OLD_STYLE_PARAMS
PRE_FUNCTION_DECL r_short POST_FUNCTION_DECL FCISimpleHelp(FunctionCall
   *theObject,
r_charP thePackageName, r_charP theFunctionName, r_u_long theFunctionID,
   IFSUserData
**theFunctionDataPtr, r_charP *theQuickDescPtr, r_charP *theFunctionDescPtr,
   r_charP
*theSampleCodePtr, r_charP **theArgsNameListPtr, r_long *theArgsNameListSizePtr,
   r_charP
**theArgsDescListPtr, r_long *theArgsDescListSizePtr, r_short
   **theArgsFlagListPtr, r_long
*theArgsFlagListSizePtr, r_charP *theRetValDescPtr, r_short
   *theRetValFlagPtr)
#else
PRE_FUNCTION_DECL r_short POST_FUNC_DECL FCISimpleHelp(theObject,
   thePackageName,
theFunctionName, theFunctionID, theFunctionDataPtr, theQuickDescPtr,
   theFunctionDescPtr,
theSampleCodePtr, theArgsNameListPtr, theArgsNameListSizePtr,
   theArgsDescListPtr,
theArgsDescListSizePtr, theArgsFlagListPtr, theArgsFlagListSizePtr,
   theRetValDescPtr, theRetValFlagPtr)
FunctionCall *theObject;
r_charP thePackageName;
r_charP theFunctionName;
r_u_long theFunctionID;
IFSUserData **theFunctionDataPtr;
r_charP *theQuickDescPtr;
r_charP *theFunctionDescPtr;
r_charP *theSampleCodePtr;
r_charP **theArgsNameListPtr;
r_long *theArgsNameListSizePtr;
r_charP **theArgsDescListPtr;
r_long *theArgsDescListSizePtr;
r_short **theArgsFlagListPtr;
```

```
r_long *theArgsFlagListSizePtr;
r_charP *theRetValDescPtr;
r_short *theRetValFlagPtr;
#endif
{
      switch(theFunctionID)
      {
         case FCI_MULTIPLY_ID:
            if (((*theQuickDescPtr) = cp_strdup("Multiplies two numbers
               together")) == NULL)
            {
               MessageBox(NULL, "Out of memory", "Error", MB_OK);
               return(NOTOK);
            }
            if (((*theFunctionDescPtr) = cp_strdup(
               "This function multiplies two numbers together."
               "/n It takes the values of two fields"
               "/n and finds their product with a precision"
               "/n of up to seven decimal places.")) == NULL)
            {
               MessageBox(NULL, "Out of memory", "Error", MB_OK);
               return(NOTOK);
            }
            if (((*theSampleCodePtr) = cp_strdup(
               "\t<field sid=\"PRODUCT_FIELD\">\n"
               "\t\t<value compute=\"test_Package.multiply"
               "(field1.value, field2.value)\"></value>\n"
               "\t\t<size>\n"
               "\t\t\t<ae>10</ae>\n"
               "\t\t\t<ae>1</ae>\n"
               "\t\t</size>\n"
               "\t</field>\n")) == NULL)
            {
               MessageBox(NULL, "Out of memory", "Error", MB_OK);
               return(NOTOK);
            }
            if (((*theRetValDescPtr) = cp_strdup(
               "The product of the two parameters given")) == NULL)
            {
               MessageBox(NULL, "Out of memory", "Error", MB_OK);
               return(NOTOK);
            }
            break;
         default:
            break;
      }
      return(OK);
}
```

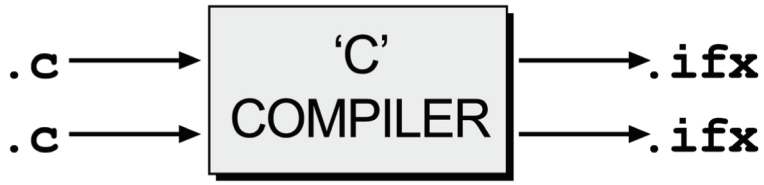
Note: For more information about the **FunctionCallHelp** template refer to page "Function Call Help Template" on page 190.

Compiling Your Extension

Once you have generated the C source file for your extension, you must compile the source code to create the extension.

- Use a compiler for C that is supported by the API to compile your extension. Refer to the *IBM Workplace Forms Server - API Installation and Setup Guide* for

more information on compatible development environments.



- Before building your extension you should have a .c source file that represents your extension. After compiling the .c file you will have a set of files with the same name as the .c files but with the extension .ifx.
- For example, after compiling the source code for the extension fciExtension.c, your C compiler will create the corresponding extension: fciExtension.ifx.
- The details of compiling your source code are not included in this manual. Consult your documentation for specific information on how to use your compiler for C.

Testing the Extension

1. To test the extension with an API application:
   - Copy the extension files that you just created to the following directory:

     ```
     <Windows System>\PureEdge\Extensions\
     ```

     To test the extension with Viewer:
   - Copy the extension files that you just created to the following directory:

     ```
     <Viewer Directory>\Extensions\
     ```
2. Use the sample form that accompanies the API to test the multiply function. Copy the file Multiply.xfd from <*API Program folder*>\Samples\Msc32\Fci\Tutorial\ to the folder containing your application.
3. Fill out the form and see if the product of **Field1** and **Field2** is returned.

Note: To view the forms provided with this API, you must have a licensed or evaluation copy of the Workplace Forms Viewer installed.

Distributing Extensions

Once you have created and tested your extensions you can distribute them as .ifx files

To distribute your extensions so that they may be used by a specific Workplace Forms product:

- Copy the extension file to the Extensions folder of the Workplace Forms product that will use the extension.
  - For example, in order for the **multiply** function to work in the Viewer, you would copy the file fciExtension.ifx to the following folder:

    ```
    <Viewer Program folder>\Extensions
    ```

To distribute your extensions so that they may be used by all Workplace Forms products:

- Copy the extension file to the Forms System Global Extensions folder.
  - For example in order for the **multiply** function to work in the Viewer you would copy the file fciExtension.ifx to the following folder:

    ```
    C:\<Windows System>\PureEdge\Extensions
    ```

Summary

By working through this section you have successfully built an extension. In the process you have learned how to initialize, compile and test FCI applications and use the following functions from the FCI Library:

- IFSObject_AllocateObject
- FunctionCall_SetObjectProc
- IFXRegisterInterface
- FCMRegisterFunctionCall

This FCI application has been included with this API and can be found in the folder <*API Program folder*>\Samples\Msc32\Fci\Demo\Multiply.

To view the forms provided with these sample applications, you must have a licensed or evaluation copy of the Workplace Forms Viewer installed.

Note: Files that are compatible with the Visual C++ development environment can be found in the <API Program folder>\Samples\Msc32. For more information about compatible development environments for the API refer to the IBM Workplace Forms Server - API Installation and Setup Guide.

FCI Library Quick Reference Guide

The following sections make up a quick reference guide to the structures, constants, and functions used in the FCI Library:

- ″The Extension Functions″ describes how to initialize an extension.
- ″The FunctionCall Structure and Functions″ describes the FunctionCall structure and lists the functions associated with the structure.
- ″The IFX Manager and Functions″ lists the FCI functions that handle the management of extensions.
- ″The Function Call Manager″ lists the FCI functions that handle the management of functions

About the Function Descriptions

The functions in this reference guide are listed according to the structures that they belong to and the functionality they provide. All functions are described using the following format:

- **Description**: Provides a general description of what the function does.
- **Function**: Lists the function's signature and type of value returned (if any).
- **Parameters**: Lists and describes each parameter in detail.
- **Returns**: Indicates what value is returned by the function.
- **Notes**: Provides additional information to help you use the function.
- **Example**: Provides sample code that uses the function in question.

The Extension Functions

The extension function **C_ExtensionInit** is used to initialize an extension and provide services for function calls through a FunctionCall structure.

When the Forms System is initialized, the API checks for existing extensions and calls **C_ExtensionInit** for each extension.

Your extension must implement the function **C_ExtensionInit** since it is defined as part of the extension interface.

Includes

You must include the following files as part of the extension source code:

```
#include "masqutil.h"
#include "IFX.h"
#include "Extension.h"
```

Example

For an example of how to set up an extension, refer to ″Getting Started with the FCI Library″ .

C_ExtensionInit

Description

This function is responsible for the registration of all the services that an extension provides. This function is not implemented in the API, so you must implement it yourself.

Function Template

```
PRE_FUNCTION_DECL r_short POST_FUNCTION_DECL C_ExtensionInit(
   Extension *theExtension,
   IFX *theIFXManager);
```

Parameters

Expression	Type	Description
theExtension	**Extension**	The extension that will be registered.
theIFXManager	**IFX**	The IFX Manager.

Returns

OK on success or an error code on failure.

Notes

- Use the **C_ExtensionInit** function to create a new function call structure that contains your custom-built functions.

- PRE_FUNCTION_DECL and POST_FUNCTION_DECL are used in function prototyping and function declaration to accurately define the function across all operating systems (for example Windows or UNIX) and compilers.

Example

```
PRE_FUNCTION_DECL r_short POST_FUNCTION_DECL C_ExtensionInit(Extension
   *theExtension,
IFX *theIFXManager)
{
FunctionCall *theFunctionCall;
      if((theFunctionCall = (FunctionCall*)IFSObject_AllocateObject(
         FUNCTIONCALL_INTERFACE_NAME, FUNCTIONCALL_CURRENT_VERSION, NOTOK,
         0))==NULL)
      {
         return (NOTOK);
      }
      /* Additional Code Removed */
}
```

The FunctionCall Structure and Functions

To define your custom functions for XFDL forms, you must first create a FunctionCall structure. The FunctionCall structure contains definitions for your custom functions.

To create a FunctionCall structure you must generate a generic IFSObject using the function **IFSObject_AllocateObject**. Then you must customize the IFSObject to provide Function Call services by calling the function **FunctionCall_SetObjectProc**.

The FunctionCall structure must provide services for the functions **FunctionCallEvaluate** and **FunctionCallHelp** since they are defined as part of the FunctionCall structure.

Remember that to make your functions available to the API you must register your FunctionCall with the IFX Manager using the function, **IFXRegisterInterface**.

You must also register each of your functions with the Function Call Manager using the function **FCMRegisterFunctionCall**.

For more information about the Function Call Manager functions refer to "The Function Call Manager" on page 197. For more information about the IFX Manager functions, refer to "The IFX Manager and Functions" on page 193.

Includes

You must include the following files as part of the extension source code:

```
#include "masqutil.h"
#include "IFX.h"
#include "xfdllib.h"
#include "FunctionCall.h"
```

Example

For an example of how to set up this class and the Extension class, refer "Getting Started with the FCI Library" on page 167.

FunctionCall Constants

The following table lists the constants that are used by the FunctionCall structure and functions along with a short description of each constant:

Named Constants	Description
FUNCTIONCALL_INTERFACE_NAME	Name of the Function Call Interface. Used with **IFXRegisterInterface** as a value for the parameter *theInterfaceName*.
FUNCTIONCALL_CURRENT_VERSION	Current version of the Function Call Interface. Used with **IFXRegisterInterface** as a value for the parameter *theInterfaceVersion*.

Named Constants	Description
FUNCTIONCALL_MIN_VERSION_ SUPPORTED	The minimum version of the Function Call Interface that is supported. Used with **IFXRegisterInterface** as a value for the parameter *theMinInterfaceVersion*.
FCICOMMAND_RUN	Used with **FunctionCallEvaluate** as a possible value for the parameter *theCommand*. This constant indicates that **FunctionCallEvaluate** should evaluate a given function.
FCICOMMAND_INSTANCEDEREGISTER	Used with **FunctionCallEvaluate** as a possible value for the parameter *theCommand*. This constant indicates that **FunctionCallEvaluate** should execute a procedure when an instance of the function has been de-registered.
FCICOMMAND_DEREGISTER	Used with **FunctionCallEvaluate** as a possible value for the parameter *theCommand*. This constant indicates that **FunctionCallEvaluate** should execute a procedure when the function has been de-registered.
FCICOMMAND_REGISTER	Used with **FunctionCallEvaluate** as a possible value for the parameter *theCommand*. This constant indicates that **FunctionCallEvaluate** should execute a procedure when the function is registered.
FCICOMMAND_INSTANCEREGISTER	Used with **FunctionCallEvaluate** as a possible value for the parameter *theCommand*. This constant indicates that **FunctionCallEvaluate** should execute a procedure when an instance of the function is registered.
FCI_WANTS_INSTANCE_DEREGISTER_ CALL	Used with **FCMRegisterFunctionCall** as a possible value for the parameter *theFlags*. Indicates that the Forms System should call **FunctionCallEvaluate** with *theCommand* set to **FCICOMMAND_ INSTANCEDEREGISTER** when an instance of the function is de-registered.
FCI_WANTS_REGISTER_CALL	Used with **FCMRegisterFunctionCall** as a possible value for the parameter *theFlags*. Indicates that the Forms System should call **FunctionCallEvaluate** with **theCommand** set to **FCICOMMAND_REGISTER** when the function is registered.
FCI_WANTS_INSTANCE_REGISTER_CALL	Used with **FCMRegisterFunctionCall** as a possible value for the parameter *theFlags*. Indicates that the Forms System should call **FunctionCallEvaluate** with *theCommand* set to **FCICOMMAND_INSTANCEREGISTER** when an instance of the function is registered.

Named Constants	Description
FCI_WANTS_INSTANCE_DATA	Used with **FCMRegisterFunctionCall** as a possible value for the parameter *theFlags*. Indicates that an **IFSUserData** will be passed as *theFunctionInstanceData* in the function **FunctionCallEvaluate**.
FCI_FOLLOWS_STRICT_CALLING_ PARAMETERS	Used with **FCMRegisterFunctionCall** as a possible value for the parameter *theFlags*. Indicates that the user of your custom function must provide the parameters you define in the **FCMRegisterFunctionCall** parameter *theCallingParams*.
FCIARGFLAG_STRING	Used as a possible value for the flag *theArgsFlagList* in the function **FunctionCallHelp**. This value represents a parameter of type **String**.
FCIARGFLAG_OPTIONAL	Used as a possible value for the flag *theArgsFlagList* in the function **FunctionCallHelp**. This value represents an optional parameter.
FCIARGFLAG_REPEATING	Used as a possible value for the flag *theArgsFlagList* in the function **FunctionCallHelp**. This value represents a repeating parameter.

IFSObject_AllocateObject

Description

This function creates a generic IFSObject and which is then cast to a FunctionCall structure. This enables you to call functions from the extension.

Function

```
IFSObject* IFSObject_AllocateObject(
   r_charP theInterfaceName,
   r_u_long theInterfaceVersion,
   r_short theFinalFlag,
   r_long theExtraDataSize);
```

Parameters

Expression	Type	Description
theInterfaceName	**r_charP**	Name of the Function Call Interface. Set to **FUNCTIONCALL_INTERFACE_NAME**.
theInterfaceVersion	**r_u_long**	The version of the Function Call Interface. Set to **FUNCTIONCALL_INTERFACE_VERSION.**
theFinalFlag	**r_short**	Flag which determines if the object can be modified. Always set to NOTOK.
theExtraDataSize	**r_long**	Always 0.

Returns

A pointer to a generic **IFSObject**. Normally, this will be cast to a **FunctionCall** pointer in an extension.

Notes

- **IFSObject_AllocateObject** creates a generic **IFSObject** that provides very few services. To customize the object you must define its behavior by making a call to **IFSObject_SetObjectProc**.

Example

```
C_ExtensionInit(Extension *theExtension, IFX *theIFXManager)
{
/*    Additional Code Removed */
      if ((theFunctionCall = (FunctionCall*)
         IFSObject_AllocateObject(FUNCTIONCALL_INTERFACE_NAME,
         FUNCTIONCALL_CURRENT_VERSION, NOTOK, 0)) == NULL)
      {
         return NOTOK;
      }
/* Additional Code Removed */
}
```

FunctionCall_SetObjectProc

Description

This function defines the services provided by a **FunctionCall** structure. Use this function to:

- Define the behavior of you custom function when called from a form.
- Implement the help provided to a developer when using your function in a compliant Workplace Forms product such as the Designer.

Function

```
r_short FunctionCall_SetObjectProc(
   FunctionCall *theFunctionCall,
   voidP theProcedure,
   r_long theProcID);
```

Parameters

Expression	Type	Description
theFunctionCall	**FunctionCall***	The Function Call whose procedures are to be set.
theProcedure	**voidP**	A pointer to the implementation of the procedure.
theProcID	**r_short**	This parameter defines the type of function being defined. Set to: **FUNCTIONCALLEVALUATE** if *theProcedure* contains the implementation details of your custom function. **FUNCTIONCALLHELP** if *theProcedure* contains the implementation details of your help function

Returns

OK on success or an error code on failure.

Notes

- In any extension, **FunctionCall_SetObjectProc** will be called twice. Once to define the behavior of your custom function and once to provide help information about each of your custom functions in the form development environment (for example the Designer).

Example

```
C_ExtensionInit(Extension *theExtension, IFX *theIFXManager)
{
      /* Additional Code Removed */
      if ((theError = FunctionCall_SetObjectProc(theFunctionCall,
         (voidP)FCISimpleFunctions, FUNCTIONCALLEVALUATE)) != OK)
      {
         return theError;
      }

      if ((theError = FunctionCall_SetObjectProc(theFunctionCall,
         (voidP)FCISimpleHelp, FUNCTIONCALLHELP)) != OK)
      {
         return theError;
      }
      /* Additional Code Removed */
}
```

Function Call Evaluate Template

Description

This function performs the necessary work for your custom-built function. You will have to insert the details of your custom functions within this function and pass a pointer for this function to IFSObject_SetObjectProc.

Function Template

```
PRE_FUNCTION_ DECL r_short POST_FUNCTION_DECL FCISimpleFunction(
   FunctionCall *theFunctionCall,
   r_charP thePackageName,
   r_charP theFunctionName,
   r_u_long theFunctionID,
   r_long theFunctionInstance,
   r_short theCommand,
   formNodeP theForm,
   formNodeP theComputeNode,
   IFSUserData **theFunctionDataPtr,
   IFSUserData **theFunctionInstanceDataPtr,
   formNodeP *theArgList,
   r_long *theArgListSizePtr,
   formNodeP theResult);
```

Parameters

Expression	Type	Description
theFunctionCall	**FunctionCall***	The **FunctionCall** structure.

Expression	Type	Description
thePackageName	**r_charP**	The name of the package that contains the function.
theFunctionName	**r_charP**	The name of the function.
theFunctionID	**r_u_long**	A unique number that can be used to identify the function.
theFunctionInstance	**r_long**	A unique number that differentiates one instance of the function from another instance. See **Notes** for more information.
theCommand	**r_short**	The name of the command for this function to perform. See **Notes** for more information. Other commands can be found within the manual.
theForm	**formNodeP**	The root node of the form that contains the function.
theComputeNode	**formNodeP**	The node within the form that the function belongs to. See **Notes** for more information.
theFunctionDataPtr	**IFSUserData****	Reserved. Although this expression is not used, it must be present.
theFunctionInstance DataPtr	**IFSUserData****	Reserved. Although this expression is not used, it must be present.
theArgList	**formNodeP***	The list of arguments. See **Notes** for more information.
theArgListSizePtr	**r_long***	A pointer to the number of arguments.
theResult	**formNodeP**	The **formNodeP** structure in which you can store the result. Simply use **UFLSetLiteralEx** on this to store the result.

Returns

OK on success or an error code on failure.

Notes

- PRE_FUNCTION_DECL and POST_FUNCTION_DECL are used to import and export the functions you implement in your .c source file to other code modules used by the Forms System.
- *theCommand* — the value of *theCommand* represents the command that your implementation of **FunctionCallEvaluate** will perform.
  - The value of *theCommand* depends on the value of the parameter called *theFlags* in the function called **FCMRegisterFunctionCall**.
  - Usually the value of *theCommand* will be set to **FCICOMMAND_RUN** which indicates that a function must be evaluated.
  - Other possible values for *theCommand* include:
  - **FCICOMMAND_INSTANCEDEREGISTER** — This constant indicates that **FunctionCallEvaluate** should execute a procedure when an instance of the function has been de-registered.

- **FCICOMMAND_DEREGISTER** — This constant indicates that **FunctionCallEvaluate** should execute a procedure when the function has been de-registered.
- **FCICOMMAND_REGISTER —** This constant indicates that **FunctionCallEvaluate** should execute a procedure when the function is registered.
- **FCICOMMAND_INSTANCEREGISTER** — This constant indicates that **FunctionCallEvaluate** should execute a procedure when an instance of the function is registered.

- *theFunctionInstance* — is a unique number that differentiates one instance of the function with another instance. For example if a form contains two calls to the function **test_Pkg.multiply** then two unique values for *theFunctionInstance* variable will exist.
- *theComputeNode* — is the node within the form that the function belongs to. For example, if you have an item like:

  ```
  <label sid="label1">
     <value compute="test_Pkg.multiply(f1.value, f2.value)"></value>
  </label>
  ```

  theComputeNode will point to the node that represents the value option.
- *theFunctionInstanceData* — is data specific to an instance of a function. It will always be returned when the instance of the function is called. This data is only provided when the **FCI_WANTS_INSTANCE_DATA** flag is set during the call to **FCMRegisterFunctionCall**.
- *theArgList* — Each argument's value is stored as a literal within a **formNodeP** structure. For example to get the value of the first argument, type the following:

  ```
  UFLGetLiteral(theArgList[0], NULL, &returnValue)
  ```

Example

```
PRE_FUNCTION_DECL r_short POST_FUNCTION_DECL FCISimpleFunctions(FunctionCall
   *theObject,
r_charP thePackageName, r_charP theFunctionName, r_u_long theFunctionID, r_long
   theFunctionInstance,
r_short theCommand, formNodeP theForm, formNodeP theComputeNode, IFSUserData
**theFunctionDataPtr, IFSUserData **theFunctionInstanceDataPtr, formNodeP
     *theArgList, r_long
theArgListSize, formNodeP theResult)
{
r_charP theFirstParam;
double firstNum;
r_charP theSecondParam;
double secondNum;
double theMultiplyResult;
char theLiteralBuffer[50];
/*    Verify the input parameters */
      if ((thePackageName == NULL) || (theFunctionName == NULL))
      {
         MessageBox(NULL, "Invalid parameters", "Error", MB_OK);
         return(NOTOK);
      }
/* Make sure that the package is the one we are expecting. */
      if (cp_strcmp(thePackageName, PACKAGE_NAME) != OK)
      {
         MessageBox(NULL,"Function called with the wrong package",
            "Error", MB_OK);
         return(NOTOK);
      }
/* The first switch in this function is based on theCommand.  The only
      case that we are interested in handling is FCICOMMAND_RUN that
      indicates that we should evaluate a function. */
```

```
      switch (theCommand)
      {
         case FCICOMMAND_RUN:
/* The second switch is based on theFunctionID that you set for each of
      your custom functions. This makes it easy for a single Function Call
      object to support multiple functions. */
            switch (theFunctionID)
            {
               case FCI_MULTIPLY_ID:
                  /* Insert Implementation Details Here */

                  break;

               default:
                  break;
            }
            break;
            case FCICOMMAND_INSTANCEREGISTER:
               break;

            case FCICOMMAND_INSTANCEDEREGISTER:
               break;
            case FCICOMMAND_REGISTER:
               break;
            case FCICOMMAND_DEREGISTER:
               break;
         default:
            break;
      }
      return(OK);
}
```

Function Call Help Template

Description

Provides help information about each of your custom functions. This help can be accesses from a form development environment. For example, the Designer can display this help information.

Function Template

```
PRE_FUNCTION_DECL r_short POST_FUNCTION_DECL FCISimpleHelp(
   FunctionCall *theFunctionCall,
   r_charP thePackageName,
   r_charP theFunctionName,
   r_u_long theFunctionID,
   IFSUserData **theFunctionDataPtr,
   r_charP *theQuickDescPtr,
   r_charP *theFunctionDescPtr,
   r_charP *theSampleCodePtr,
   r_charP **theArgsNameListPtr,
   r_long *theArgsNameListSizePtr,
   r_charP *theArgsDescListPtr,
   r_long *theArgsDescListSizePtr,
   r_short **theArgsFlagListPtr,
   r_long *theArgsFlagsListSizePtr,
   r_charP *theRetValDescPtr,
   r_short *theRetValFlagPtr);
```

Parameters

Expression	Type	Description
theFunctionCall	**FunctionCall***	The **FunctionCall** structure for your application.
thePackageName	**r_charP**	The name of the package that contains the function.
theFunctionName	**r_charP**	The name of the function.
theFunctionID	**r_u_long**	A unique number that can be used to identify the function.
theFunctionDataPtr	**IFSUserData****	Reserved. Although this expression is not used, it must be present.
theQuickDescPtr	**r_charP***	The function will set a quick one-line description of what the function does.
theFunctionDescPtr	**r_charP***	The function will set a longer more detailed description of the function.
theSampleCodePtr	**r_charP***	The function will set an example of the XFDL code used to call your function, including an example of the function parameters.
theArgsNameListPtr	**r_charP****	The function will set a list of arguments that your function takes. See **Notes** for more information.
theArgsNameListSizePtr	**r_long***	The pointer to the size of *theArgsNameList*.
theArgsDescListPtr	**r_charP***	The function will set a description of each of the arguments in the *theArgsNameList*. See Notes for more information.
theArgsDescListSizePtr	**r_long***	The pointer to the size of *theArgsDescList*.
theArgsFlagListPtr	**r_short***	The function will set a list of bit flags representing the type of each argument that the function takes. See **Notes** for more information.
theArgsFlagsListSizePtr	**r_long***	A pointer to the size of *theArgsFlagList*
theRetValDesc	**r_charP***	The function will set a description of function custom function's return value.
theRetValFlag	**r_charP***	The function will set a bit flag representing the type of the return value. See **Notes** for more information. Simply use **UFLSetLiteralEx** on this object to store the result.

Returns

OK on success or an error code on failure.

Notes

- PRE_FUNCTION_DECL and POST_FUNCTION_DECL are used to import and export the functions you implement in your .c source file to other code modules used by the Forms System.
- Refer to the table of "FunctionCall Constants" on page 183 for possible values for:
  - theArgsFlagList
  - theRetValFlag
- For both *theArgsDescListPtr* and *theArgsNameListPtr,* if you have no arguments you must dereference the pointer and set it to **NULL**. For example:

```
*theArgsNameListPtr = NULL;
```

Example

```
PRE_FUNCTION_DECL r_short POST_FUNCTION_DECL FCISimpleHelp(FunctionCall *theObject,
r_charP thePackageName, r_charP theFunctionName, r_u_long theFunctionID, IFSUserData
**theFunctionDataPtr, r_charP *theQuickDescPtr, r_charP *theFunctionDescPtr, r_charP
*theSampleCodePtr, r_charP **theArgsNameListPtr, r_long *theArgsNameListSizePtr,
   r_charP
**theArgsDescListPtr, r_long *theArgsDescListSizePtr, r_short **theArgsFlagListPtr,
   r_long
*theArgsFlagListSizePtr, r_charP *theRetValDescPtr, r_short *theRetValFlagPtr)
{
/* Switch on theFunctionID. This makes it easy for a single FunctionCall
      structure to support multiple functions. */

      switch(theFunctionID)
      {
/* Remember that you must define an ID number for each custom function
      that you create. In the example below the constant FCI_MULTIPLY_ID
      represents the identification number for the multiply function. */
         case FCI_MULTIPLY_ID:
            *theQuickDescPtr =
               cp_strdup("Multiplies two numbers together");

            *theFunctionDescPtr =cp_strdup(
               "Function multiplies two numbers.\n"
               "It takes the values of two fields\n"
               "and finds their product with precision\n"
               "of up to seven decimal places.");
            *theSampleCodePtr = cp_strdup(
               "\t<field sid=\"PRODUCT_FIELD\">\n"
               "\t\t<value compute=\"test_Pkg.multiply"
               "(field1.value, field2.value)\"></value>\n"
               "\t\t<size>\n"
               "\t\t\t<ae>10</ae>\n"
               "\t\t\t<ae>1</ae>\n"
               "\t\t</size>\n"
               "\t</field>\n");
            *theRetValDescPtr = cp_strdup(
               "The product of the two parameters given");
            break;
         default:
            break;
      }
      return(OK);
}
```

The IFX Manager and Functions

The IFX Manager consists of the **IFX structure** and several functions that manage extensions. To create a simple package of functions that can be called from within an XFDL form, you need to use the IFXRegisterInterface function.

For information on the FCI functions used to create a simple package, refer to "The Function Call Manager" on page 197.

Includes

You must include the following files as part of the extension source code:

```
#include "masqutil.h"
#include "IFX.h"
```

Example

For an example of how to use this function, refer to ″Getting Started with the FCI Library″ .

IFXDeregisterInterface

Description

De-registers a **FunctionCall** from the IFX Manager.

Function

```
r_short IFXDeregisterInterface(
   IFX *theIFXManger,
   GenericInterface *theInterface);
```

Parameters

Expression	Type	Description
theIFXManager	**IFX***	The IFX Manager.
theInterface	**GenericInterface***	The **FunctionCall** structure you are de-registering with the IFX Manager. See **Notes** for more information.

Returns

OK on success or an error code on failure.

Example

The following example registers a FunctionCall with the IFX Manager by calling **IFXRegisterInterface**. The interface is the immediately de-registered with a call to IFXDeregisterInterface.

```
if ((theError = IFXRegisterInterface(theIFXManager,
      (GenericInterface*)theFunctionCall, IFX_INTERFACE_NAME,
      IFX_CURRENT_VERSION, IFX_MIN_VERSION_SUPPORTED, 0X01000300L, 0, NULL,
      0, FCMGetDefaultListener() ))!= OK)
{
      return theError;
}
if ((theError = IFXDeregisterInterface(theIFXManager,
      (GenericInterface*)theFunctionCall) ) != OK)
{
      return theError;
}
```

IFXGetInterfaceInstances

Description

Returns a list of **FunctionCall** structures that are currently registered with the IFX Manager.

Function

```
r_short IFXGetInterfaceInstances(
   IFX *theIFXManager,
   r_charP theInterfaceName,
   r_u_long theInterfaceVersion,
   GenericInterface ***theInterfaceListPtr,
   r_long *theInterfaceListSizePtr);
```

Parameters

Expression	Type	Description
theIFXManager	**IFX***	The IFX Manager for your application.
theInterfaceName	**r_charP**	The name of the interface that you are looking for. In this case the Function Call Interface.
theInterfaceVersion	**r_u_long**	The Function Call Interface version.
theInterfaceListPtr	**GenericInterface*****	A pointer to the Generic Interface List that contains all of the instances of the interface specified in **theInterfaceName**. **Note** Typically the Generic Interface instances in the list must be typecast to type **FunctionCall**.
theInterfaceListSizePtr	**r_long***	Points to the size of the Generic Interface List.

Returns

OK on success or an error code on failure.

Example

```
GenericInterface **theFunctionCallList;
r_long *theNumOfFunctionCalls;
FunctionCall theFunctionCall;
      if (theError = (IFXGetInterfaceInstances(theIFXMangager,
         FUNCTIONCALL_INTERFACE_NAME, FUNCTIONCALL_CURRENT_VERSION,
         &theFunctionCallList,theNumOfFunctionCalls)) != OK)
      {
         return theError;
      }
      for(int i = 0 ; i < *theNumberOfFunctionCalls; i++)
      {
         theFunctionCall = (FunctionCall*)theFunctionCallList[i];
      }
```

IFXRegisterInterface

Description

Registers a **FunctionCall** structure with the IFX Manager.

Function

```
r_short IFXRegisterInterface(
   IFX *theIFXManager,
   GenericInterface *theInterface,
   r_charP theInterfaceName,
   r_u_long theInterfaceVersion,
   r_u_long theMinInterfaceVersion,
   r_u_long theImplementationVersion,
   r_u_long theFlags,
   r_charP *theCriteriaList,
   r_long theCriteriaListSize,
   IFXCriteriaMatchingHandler *theCriteriaHandler);
```

Parameters

Expression	Type	Description
theIFXManager	**IFX***	The IFX Manager.
theInterface	**GenericInterface***	The structure that you are registering with the IFX Manager. In this case, a **FunctionCall** Interface will be registered.
theInterfaceName	**r_charP**	The name of the Interface that you are registering. In this case a Function Call Interface. Typical setting: **FUNCTIONCALL_INTERFACE_NAME**
theInterfaceVersion	**r_u_long**	The function call interface version. Typical setting: **FUNCTIONCALL_CURRENT_VERSION**
theMinInterfaceVersion	**r_u_long**	The minimum version that the interface will support. Typical setting: **FUNCTIONCALL_MIN_VERSION_ SUPPORTED**
theImplementation Version	**r_u_long**	The **FunctionCall** implementation version.
TheFlags	**r_u_long**	Reserved. Setting: **0**
theCriteriaList	**r_charP**	Reserved. Setting: **null**

Expression	Type	Description
theCriteriaListSize	**r_long**	A pointer to the size of **theCriteriaList**. Reserved. Setting: **0**
theCriteriaHandler	**IFXCriteria MatchingHandler***	Reserved. Setting: **FCMGetDefaultListener()**

Returns

OK on success or an error code on failure.

Example

```
r_short C_ExtensionInit(Extension *theExtension, IFX *theIFXManager)
{
FunctionCall *theFunctionCall;
r_short theError;
      /* Additional code removed */
      if ((theError = IFXRegisterInterface(theIFXManager,
         (GenericInterface*)theFunctionCall,FUNCTIONCALL_INTERFACE_NAME,
         FUNCTIONCALL_CURRENT_VERSION, FUNCTIONCALL_MIN_VERSION_SUPPORTED,
         0X01000300L, 0, NULL, 0, FCMGetDefaultListener() )) != OK)
      {
         return(theError);
      }
      /* Additional code removed */
}
```

The Function Call Manager

The Function Call Manager is a collection of functions that handle the management of FunctionCall structures. To create a simple package of functions for calling from within XFDL forms, you need to use the following Function Call Manager functions:

- FCMRegisterFunctionCall
- FCMEvaluateFunctionCall
- FCMGetFunctionCallHelp

Includes

You must include the following files as part of the IFX extension source code:

```
#include "masqutil.h"
#include "IFX.h"
#include "FunctionCall.h"
#include "FunctionCallManager.h"
```

Example

For an example of how to use these functions to create a package of functions, refer to ″Getting Started with the FCI Library″ .

FCMDeregisterFunctionCall

Description

De-registers a particular function from a **FunctionCall** structure.

Function

```
r_short FCMDeregisterFunctionCall(
   FunctionCall *theFunctionCallObject,
   r_charP thePackageName,
   r_charP theFunctionName);
```

Parameters

Expression	Type	Description
theFunctionCallObject	**FunctionCall***	The Function Call structure that contains the function that you are de-registering.
thePackageName	**r_charP**	The name of the package that the function belongs to. See **Notes** for more information.
theFunctionName	**r_charP**	The name of the function to be de-registered. See **Notes** for more information.

Returns

OK on success or an error code on failure.

Notes

- *thePackageName* — Use the package name that you created when you registered the function using **FCMRegisterFunctionCall**.
- *theFunctionName* — Use the function name that you created when you registered the function using **FCMRegisterFunctionCall**.

Example

The following example de-registers the multiply function from the package called test_Package.

```
if ( theError = FCMDeregisterFunctionCall(theFunctionCall,
      "test_Package", "multiply") == NOTOK )
{
         return (theError);
}
```

FCMGetDefaultListener

Description

Helps the IFX Manager determine which **FunctionCall** implements a specific function.

Function

```
IFXCriteriaMatchingHandler* FCMGetDefaultListener();
```

Parameters

There are no parameters for this function.

Returns

Returns a pointer to an **IFXCriteriaMatchingHandler** that can be used to locate the **FunctionCall** that contains the specific function.

Notes

Typically this function is used when calling the IFX function **IFXRegisterInterface**. Refer to the function description for **IFXRegisterInterface** for more information.

FCMGetFunctionCallHelp

Description

This function is used by the API to call the **FunctionCall Manager** function **FCMFunctionCallHelp**.

Function

```
r_short FCMGetFunctionCallHelp(
   r_charP thePackageName,
   r_charP theFunctionName,
   r_u_long *theVersionPtr,
   r_charP *theQuickDescPtr,
   r_charP *theFunctionDescPtr,
   r_charP *theSampleCodePtr,
   r_charP *theArgsNameListPtr,
   r_long *theArgsNameListSizePtr,
   r_charP **theArgsDescListPtr,
   r_long *theArgsDescListSizePtr,
   r_short **theArgsFlagListPtr,
   r_long *theArgsDescListSizePtr,
   r_charP *theRetValDescPtr,
   r_short *theRetValFlagPtr);
```

Parameters

Expression	Type	Description
thePackageName	**r_charP**	The name of the package that contains the function.
theFunctionName	**r_charP**	The name of the function.
theVersionPtr	**r_u_long***	Pointer to the version number of the function.
theQuickDescPtr	**r_charP***	Pointer to a one-line description of what the function does.
theFunctionDescPtr	**r_charP***	Pointer to a longer more detailed description of the function.
theSampleCodePtr	**r_charP***	Pointer to an example of the XFDL code used to call your function, including an example of the function parameters.
theArgsNameListPtr	**r_charP****	Pointer to a list of arguments that your function takes.
theArgsNameListSizePtr	**r_long***	A pointer to the size of the theArgsNameList.
theArgsDescListPtr	**r_charP****	Pointer to a description of each of the arguments in the theArgsNameList.
theArgsDescListSizePtr	**r_long***	Pointer to the size of theArgsDescList
theArgsFlagListPtr	**r_short****	Pointer to a list of bit flags representing the type of each argument that the function takes. See **Notes** for more information.
theArgsFlagListSizePtr	**r_long***	Pointer to the size of theArgsFlagList
theRetValDescPtr	**r_charP***	Pointer to a description of your custom function's return value.
theRetValFlagPtr	**r_short***	Pointer to a bit flag representing the type of the return value. See **Notes** for more information. Simply use **UFLSetLiteralEx** on this object to store the result.

Returns

OK on success or an error code on failure.

Notes

Refer to the table of "FunctionCall Constants" on page 183 for possible values for:

- theArgsFlagList
- theRetValFlag

Example

In the example below the function another_test_Package.multiply uses **FCMGetFunctionCallHelp** to call the help function that was defined for the test_Package.multiply function.

```
r_short FCISimpleHelp(FunctionCall *theObject,r_charP thePackageName,
   r_charP theFunctionName,
r_u_long theFunctionID, IFSUserData **theFunctionDataPtr,
   r_charP *theQuickDescPtr, r_charP
*theFunctionDescPtr, r_charP *theSampleCodePtr, r_charP **theArgsNameListPtr,
   r_long
*theArgsNameListSizePtr, r_charP **theArgsDescListPtr,
   r_long *theArgsDescListSizePtr, r_short
**theArgsFlagListPtr, r_long *theArgsFlagListSizePtr, r_charP *theRetValDescPtr,
   r_short
*theRetValFlagPtr)
{
/* Switch on theFunctionID. This makes it easy for a single FunctionCall
      to support multiple functions. */

      switch(theFunctionID)
      {
/* Remember that you must define an ID number for each custom function
      that you create. In the example below the constant FCI_MULTIPLY_ID
      represents the identification number for the multiply function. */
         case FCI_MULTIPLY_ID:

            FCMGetFunctionCallHelp("test_Package", "multiply",theVersion,
               theQuickDesc, theFunctionDesc, theSampleCode,
               theArgsNameListPtr, theArgsNameListSize, theArgsDescListPtr,
               theArgsDescListSizePtr, theArgsFlagListPtr,
               theArgsFlagListSizePtr, theRetValDesc, theRetValFlag);
               /* Additional Code Removed */
            break;

         default:
            break;
      }
      return(OK);
}
```

FCMGetFunctionCallList

Description

Lists the functions that belong to a particular package.

Function

```
r_short FCMGetFunctionCallList(
   r_charP thePackageName,
   r_charP **returnPtr,
   r_long *returnSizePtr);
```

Parameters

Expression	Type	Description
thePackageName	**r_charP**	The package name.
returnPtr	**r_charP****	Pointer to a list of functions that belong to the package specified in thePackageName. Always set to NULL when function is called.
returnSizePtr	**r_long***	Pointer to the size of the list. Always set to NULL when function is called.

Returns

OK on success or an error code on failure.

Example

```
r_charP *theFunctionCallList;
r_long FunctionCallSize;
/* Additional code removed */
      FCMGetFunctionCallPackageList("test_Package",&theFunctionCallList,
         &FunctionCallSize);
```

FCMGetFunctionCallPackageList

Description

Lists the packages that are currently registered with the Function Call Manager.

Function

```
r_short FCMGetFunctionCallPackageList(
   r_charP **returnPtr,
   r_long *returnSizePtr);
```

Parameters

Expression	Type	Description
returnPtr	**r_charP****	Pointer to a list of package names that are registered with the Function Call Manager. Always set to NULL when function is called.
returnSizePtr	**r_long***	Pointer to the size of the list. Always set to NULL when function is called.

Returns

OK on success or an error code on failure.

Example

```
r_charP *thePackageList;
r_long FunctionCallSize;
/* Additional code removed */
      FCMGetFunctionCallPackageList(&thePackageList, &FunctionCallSize);
```

FCMRegisterFunctionCall

Description

Registers your custom function with the Function Call Manager.

Function

```
r_short FCMRegisterFunctionCall(
   FunctionCall *theFunctionCallObject,
   r_charP thePackageName,
   r_charP theFunctionName,
   r_u_long theFunctionID,
   r_u_long theFlags,
   r_charP theCallingParams,
   r_u_long theVersion,
   r_charP theQuickDesc);
```

Parameters

Expression	Type	Description
theFunctionCallObject	**FunctionCall***	The **FunctionCall** that will handle requests for the function. Setting: **Name of the FunctionCall.**
thePackageName	**r_charP**	The name of the package that will contain the function. Setting: **Name of the package**.
theFunctionName	**r_charP**	The name of the function. Setting: **Name of the function.**
theFunctionID	**r_u_long**	A unique number that can be used to identify the function. Setting: **The Function ID**. See **Notes** for more information.
theFlags	**r_u_long**	A set of flags which indicate how the custom function will be evaluated. Setting: Typically **FCI_FOLLOWS_STRICT_CALLING_ PARAMETERS** or **0.** See **Notes** for more information.
theCallingParams	**r_charP**	The list of parameters that this function takes. Setting: **S**, **O**, or **R**. See **Notes** for more information.
theVersion	**r_u_long**	The version number of the function. Setting: **Function Version Number**. See **Defining a Version Number** below for more information.
theQuickDesc	**r_charP**	A quick one-line description of what the function does.

Returns

OK on success or an error code on failure.

Notes

theFunctionID - Each function that you create as part of a particular package must have a unique identification number. Define each function's ID number as a constant at the beginning of the class. For example the multiply function has an ID number of 1:

```
#define MULTIPLY_ID 1;
```

- *theFlags* - Refer to the table of ″FunctionCall Constants″ for a list of possible values for *theFlags*.

theCallingParams - List the type of each parameter that the function will take and separate each value with a comma.

- Use **S** to indicate a string parameter.
- If the parameter is optional, then an **O** is added after the **S**.
- If the parameter can repeat, then an **R** is added after the **S**.
- For example, if you were to register a function that had to have one parameter and optionally a second parameter then the **theCallingParams** would look like the following:

```
"S,SO"
```

- If there are no parameters, use an empty string (″″).

Defining a Version Number

- If multiple **FunctionCall** objects register the same function for the same package, then the function with the highest version number is used.
- Version numbers are defined in hexadecimal format as follows, where the 0300 is a constant and must be present :

```
0x<major><minor><maintenance><0300>
```

- For example, a function that is version 2.1 would be represented as

```
0x02010300
```

- Define a function's version number in the parameter *theVersion*.

For more information about using version numbers refer to 203.

Example

In the following example, the multiply function is registered with the Function Call Manager as a part of a package called test_Package.

The multiply function takes two string parameters and has the version number: 1.0

```
PRE_FUNCTION_DECL r_short POST_FUNCTION_DECL C_ExtensionInit(Extension *theExtension,
   IFX *theIFXManager)
{
FunctionCall *theFunctionCall;
r_short theError;

      if ((theFunctionCall = (FunctionCall*)IFSObject_AllocateObject(
         FUNCTIONCALL_INTERFACE_NAME,FUNCTIONCALL_CURRENT_VERSION, NOTOK,
         0)) == NULL)
      {
         return NOTOK;
      }
      if ((theError = FCMRegisterFunctionCall( theFunctionCall, PACKAGE_NAME,
         "multiply", FCI_MULTIPLY_ID,
         FCI_FOLLOWS_STRICT_CALLING_PARAMETERS, "S,S",
         FCI_MULTIPLY_VERSION, FCI_MULTIPLY_DESCRIPTION )) != OK)
      {
         return(theError);
      }
      return OK;
}
```

Appendix A: Setting Up Your Development Environment

To create Form or FCI applications you must configure your development environment properly.

The API includes a set of project templates that are preconfigured to work with the API libraries. You can use these templates as a starting point in creating your own applications. For more information about project templates, refer to *the IBM Workplace Forms Server - API Installation & Setup Guide*.

If the project templates provided with the API are not compatible with your development environment, follow the procedures in this section to configure your project file.

The following procedures provide specific examples of how to set up your applications for Microsoft Visual C++ 6.0. You can use these procedures as a guideline if you are using other development environments supported by the API. For further information, refer to the documentation for the development environment you are working in.

Note: Both Form and FCI applications use the same development environment with the exception of the type of project that is produced. A project using the Form Library will produce a 32-bit application. A project using the FCI Library will be configured to produce a multi-threaded dynamic link library (DLL) with the file name extension .ifx.

Developing Form Applications

To set up your development environment to use the Form Library, follow this procedure:

1. Create a new 32-bit Windows project.
2. Add the API include file to your project. This folder is located at:
   `<API Installation Folder>/include/`
3. Include **pe_cc.lib** and **pe_crt71.lib** in your project.
4. Ensure **msvcrt.lib** is not included in your project.
5. Set the Preprocessor Definitions in your project to **WINDOWS**.
6. If you are using Microsoft Visual Studio .NET, you must set the project to use a Multithreaded dll. (You can find this option under **Project → Properties**. Open the **C/C++** > **Code Generation** folder, and set the **Run-time** drop-down list to **Multi-threaded DLL**.)

Developing FCI Applications

To set up your development environment to use the FCI Library, follow this procedure:

1. Create a new **Win32 Dynamic-Link Library** project.
2. Add the API include file to your project. This folder is located at:
   `<API Installation Folder>/include/`
3. Include **pe_cc.lib** and **pe_crt71.lib** in your project.

4. Ensure **msvcrt.lib** is not included in your project.
5. Set the Preprocessor Definitions in your project to **WINDOWS**.
6. If you are using Microsoft Visual Studio .NET, you must set the project to use a Multithreaded dll. (You can find this option under **Project → Properties**. Open the **C/C++** > **Code Generation** folder, and set the **Run-time** drop-down list to **Multi-threaded DLL**.)
7. Set your project to output files with an .extension.

Updgrading to Visual Studio .Net

If you are upgrading to Visual Studio .Net from a previous version of Visual Studio, the upgrade process may create file overrides for project settings. If this happens, you must remove those overrides or project settings will not be applied to those files.

Appendix. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Office 4360
One Rogers Street
Cambridge, MA 02142
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX
IBM
Workplace
Workplace Forms

Intel, Intel Inside (logos), MMX, and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Index

Special characters

A

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Program Number: 5724-N08

Printed in USA

S229-1526-00