Transferring Data With the URL Access Manager

For URL Access Manager 2.0.3

Preliminary

5/12/00

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

Simultaneously published in the United States and Canada.

Contents

Figures, Listings, and Tables

Introduction

This document describes the URL Access Manager. You can use the URL Access Manager to perform data transfer to and from a URL from within your application.

Some of the features of URL Access Manager includes support for

- automatic decompression of compressed files
- automatic file extraction from Stuffit archives (with version 5.0 of Stuffit)
- firewalls, HTTP proxy servers, and SOCKS gateways

URL Access Manager allows you to use any of the following protocols during download operations: File Transfer Protocol (FTP), Hypertext Transfer Protocol (HTTP), secure Hypertext Transfer Protocol (HTTPS), or a URL representing a local file (begins with file:/ / /). You might use the latter to test your application on a computer that does not have access to a HTTP or FTP server. For upload operations, you must use an FTP URL.

The URL Access Manager allows you to upload data to an FTP URL using either anonymous or authenticated FTP sessions and supports both passive and active FTP connections. You can use FTP to download and upload files and directories, as well as to set and obtain URL properties.

If you use HTTP or HTTPS when downloading data, you will be able to perform data transfer with 40-bit RSA encryption, send HTML form information to a URL, and set and obtain URL properties.

You should read the following sections to get more information about the URL Access Manager:

- “URL Access Manager Tasks” (page 11) provide an introduction to programming the URL Access Manager.

- "URL Access Manager Reference" (page 31) describes the URL Access Manager API through version 2.0.3, including functions, data types, constants, and result codes.
- "Document Revision History" (page 83) provides a history of changes to this document.

All code listings in this document are shown in C, except for listings that describe resources, which are shown in Rez-input format. Many listings are taken from the SamplePost.h sample application, which is available through Apple's developer site at

<http://developer.apple.com/>

Note: Although the sample code in this document has been compiled and tested to some degree, Apple Computer does not recommend that you directly incorporate this code into your application. For example, only limited error handling is shown—you should develop your own techniques for detecting and handling errors.

URL Access Manager Tasks

This chapter describes how to modify your application to use the URL Access Manager to transfer data to and from a uniform resource locator (URL).

The following sections provide an introduction to programming the URL Access Manager:

URL Access Manager Implementation

The URL Access Manager is supported on computers running Mac OS 8.6 and 9.0. It is implemented as a shared library called the URL Access Library. The URL Access Manager is not currently in Carbon. As a result, in Mac OS X, you should weak link against the URL Access Library and test to see if the functions are available.

Determining Availability and Version Information

You should call the function `URLAccessAvailable` (page 32) to determine whether the URL Access Manager is available before calling any other URL Access Manager functions. Note that because the URL Access Manager is not currently in Carbon, in Mac OS X, you should weak link against the URL Access Library and test to see if the functions are available. The function `URLGetURLAccessVersion` (page 32) returns the version number of the URL Access Manager installed on the current system.

Creating a URL Reference

The URL Access Manager provides the function to create a URL reference. The URL Access Manager uses a URL reference to uniquely identify a URL. The function `URLNewReference` (page 33) enables you to create a URL reference given the name of the URL. After you are finished with a URL reference, you should call the function `URLDisposeReference` (page 33) to dispose of the memory it occupied.

Getting and Setting Information About a URL

The URL Access Manager provides the functions `URLGetProperty` (page 35) and `URLSetProperty` (page 36) to get and set information associated with a URL You must pass the correct data type of the property value you wish to get or set in the propertyBuffer parameter of both functions. Before calling these functions, you should call the function `URLGetPropertySize` (page 34) to determine the size of the buffer to allocate for the property value.

You may wish to call these functions before calling the functions `URLDownload` (page 40) and `URLUpload` (page 45) to get and set information associated with the specified URL in the `urlRef` parameter.

Once you have obtained the value and size of a URL property, you can create a function to display the properties. Listing 2-1 (page 13) illustrates how the function `displayProperties` creates a `propertyList` array containing each of the twenty-one Apple-defined URL properties. It obtains the size of each property by calling `URLGetPropertySize` (page 34), obtains the value of each property by calling and `URLGetProperty` (page 35), and then displays each property value.

Listing 2-1 Displaying the value of each URL property

```
void displayProperties( URLReference urlRef )
{
    OSErr err = noErr;
    int propCount = 0;
    const char* propertyList[21];
    Size propertySize = 0;
    Handle theProperty = NULL;
    propertyList[0]  = kURLURL;
    propertyList[1]  = kURLResourceSize;
    propertyList[2]  = kURLLastModifiedTime;
    propertyList[3]  = kURLMIMEType;
    propertyList[4]  = kURLFileType;
    propertyList[5]  = kURLFileCreator;
    propertyList[6]  = kURLCharacterSet;
```

```
    propertyList[7]  = kURLResourceName;
    propertyList[8]  = kURLHost;
    propertyList[9]  = kURLAuthType;
    propertyList[10] = kURLUserName;
    propertyList[11] = kURLPassword;
    propertyList[12] = kURLStatusString;
    propertyList[13] = kURLIsSecure;
    propertyList[14] = kURLCertificate;
    propertyList[15] = kURLTotalItems;
    propertyList[16] = kURLHTTPRequestMethod;
    propertyList[17] = kURLHTTPRequestHeader;
    propertyList[18] = kURLHTTPRequestBody;
    propertyList[19] = kURLHTTPRespHeader;
    propertyList[20] = kURLHTTPUserAgent;

// Get the size of each property, allocate a handle to store the
// property's value, get the property value, and display it.
    for( propCount = 0; propCount < 21; propCount++)
    {

// Get the size of the property's value.
        err = URLGetPropertySize(urlRef, propertyList[propCount], &propertySize);
        if(err != noErr)
            printf("Error %d getting property size %s. Size returned was: %d\n", err,
                    propertyList[propCount], propertySize);
        else
            printf("Property size is %d: %s\n", propertySize);

// Now get a handle for the property value.
        theProperty = NewHandleClear( propertySize + 1 );
        err = MemError();
        if(err != noErr)
            printf("Error %d getting property handle %s\n", err,
                propertyList[propCount]);
        else
            printf("Got handle for %s: %s\n", propertyList[propCount]);

// Now get the property's value.
        err = URLGetProperty(urlRef, propertyList[propCount],
            *theProperty, propertySize);
        if(err != noErr)
```

```
            printf("Error %d getting property %s\n", err, propertyList[propCount]);
        else
            printf("Property %s: %s\n", propertyList[propCount], *theProperty);

// Clean up.
        DisposeHandle(theProperty);
        printf("\n");
}
return;
}
```

Performing Simple Data Transfer

URL Access Manager provides four high-level functions for downloading and uploading data synchronously. Synchronous functions return control to your application upon completion. If you wish instead to perform asynchronous data transfer, see “Controlling Data Transfer” (page 16).

The functions `URLSimpleDownload` (page 38) and `URLSimpleUpload` (page 43) require that you specify the URL as a character string. These functions are easy to use because they allow you to start a download or upload operation with a minimal amount of preparation or intervention.

The functions `URLDownload` (page 40) and `URLUpload` (page 45) also allow you to download and upload data synchronously. These functions differ from `URLSimpleDownload` and `URLSimpleUpload` in that you use a URL reference to specify the URL. Using a URL reference allows you to get and set information associated with a URL by calling the `URLGetProperty` (page 35) and `URLSetProperty` (page 36) functions.

All four functions have an `openFlags` parameter which you can use to specify whether an existing file should be replaced, a progress indicator is displayed during the transfer, or an authentication dialog box is displayed if the URL requires authentication. You can also use this parameter to specify that encoded files are to be decoded and expanded if the Stuffit Engine is installed, indicate that the URL is a directory, or specify that you want to download a directory listing instead of the contents of a file or directory.

In addition, these functions allow you to specify an application-defined system event callback function in the `eventProc` parameter that the URL Access Manager calls in order to convey system events to your application during the download process.

Controlling Data Transfer

The URL Access Manager provides the function `URLOpen` (page 47) to transfer data to and from a URL asynchronously. Asynchronous functions return control to your application immediately. If you wish instead to perform synchronous data transfer, see “Performing Simple Data Transfer” (page 15).

When you call `URLOpen`, you must specify a valid file for upload operations. For download operations, if you do not specify a file in which to store the data, you must repeatedly call the function `URLGetBuffer` (page 51). This function retrieves the next buffer of data from the URL Access Manager’s buffers so that you can manipulate the data or write it to the destination of your choice. If you do not specify a file in which to store the data, you can define an event notification function and the events for which you want to receive notification.

You can use the `openFlags` parameter of `URLOpen` to specify whether an existing file should be replaced, whether a progress indicator is displayed during the transfer, and whether an authentication dialog is displayed if the URL requires authentication. You can also use this parameter to specify that encoded files are to be decoded and expanded if the Stuffit Engine is installed, indicate that the URL is a directory, or specify that you want to download a directory listing instead of the contents of a file or directory.

The function `URLAbort` (page 49) terminates a data transfer operation that was started by calling `URLOpen`. When your application calls `URLAbort`, the URL Access Manager changes the state returned by the function `URLGetCurrentState` (page 53) to `kURLAbortingState` and passes the constant `kURLAbortInitiatedEvent` to your notification callback function. When data transfer is terminated, the URL Access Manager changes the state returned by `URLGetCurrentState` to `kURLCompletedState` and passes the constant `kURLCompletedEvent` in the `event` parameter of your notification callback function.

The function `URLGetDataAvailable` (page 50) determines the number of bytes remaining in a buffer after a call to the function `URLGetBuffer` (page 51). You should only call `URLGetDataAvailable` if you have initiated a download process. The returned size does not include the number of bytes in transit to a buffer, nor does it include the amount of data not yet transferred from the URL. If you wish to calculate the amount of data remaining to be downloaded, pass the name constant `kURLResourceSize` in the `property` parameter of the function `URLGetProperty` (page 35) and subtract the amount of data copied.

The function `URLGetBuffer` (page 51) obtains the next buffer of data in a download operation. `URLGetBuffer` does not enable you to retain or modify the transferred data. If you pass `NULL` in the `fileSpec` parameter of the function `URLOpen` (page 47), you should call `URLGetBuffer` to retrieve data as it is downloaded.

You should call `URLGetBuffer` repeatedly until URL Access Manager passes the event constant `kURLCompletedEvent` or `kURLAbortInitiatedEvent` in the `event` parameter of your notification callback function, or until the function `URLGetCurrentState` (page 53) returns the state constant `kURLTransactionComplete` or `kURLAbortingState`. Between calls to `URLGetBuffer`, you should call the function `URLIdle` (page 54) to allow time for the URL Access Manager to refill its buffers.

The function `URLReleaseBuffer` (page 52) releases the buffer obtained by calling `URLGetBuffer`. To prevent the URL Access Manager from running out of buffers, you should call `URLReleaseBuffer` after each call to `URLGetBuffer`.

Obtaining Information About a Data Transfer Operation

You can use these functions to determine the error code returned when a data transfer operation fails, determine the status of a data transfer operation, yield time so that the URL Access Manager can refill its buffers, or get information about a file.

You may want to call the function `URLGetError` (page 52) when a data transfer operation fails. `URLGetError` passes back the error code associated with the failed transfer, which may be a system error code, a protocol-specific error code, or one of the error codes listed in “URL Access Result Codes” (page 80).

The function `URLGetCurrentState` (page 53) passes back the status of a data transfer operation. You may wish to call `URLGetCurrentState` periodically to monitor the status of a download or upload operation. If you pass a valid file specification in the `fileSpec` parameter of the function `URLOpen` (page 47), your notification callback function will not be notified of data available and transaction completed states as identified by the constants `kURLDataAvailableState` and `kURLTransactionCompleteState`.

The function `URLGetFileInfo` (page 54) obtains the file type and creator codes for a specified filename. The type and creator codes are determined by the Internet configuration mapping table and are based on the filename extension. For example, if you pass the filename `"jane.txt"`, `URLGetFileInfo` will return `'TEXT'` in the `type` parameter and `'ttxt'` in the `creator` parameter.

The function `URLIdle` (page 54) gives the URL Access Manager time to refill its buffers during download operations. If you pass `NULL` in the `fileSpec` parameter of the function `URLOpen`, you should call the function `URLGetBuffer` (page 51) to retrieve data as it is downloaded. To allow time for the URL Access Manager to refill its buffers, call `URLIdle` in between calls to `URLGetBuffer`.

Responding to Data Transfer Events

Data transfer events are generated during a call to the function `URLOpen` (page 47) when one of the following situations arises:

- `URLOpen` has been called but the location specified by the URL reference has not yet been accessed.
- the location specified by the URL reference has been accessed and is valid.
- a download operation is in progress.
- a data transfer operation has been aborted.
- all operations associated with a call to `URLOpen` have been completed.
- an error occurred during data transfer.
- data is available in buffers.

- a download operation is complete because there is no more data to retrieve from buffers.
- an upload operation is in progress.
- a system event has occurred.
- the size of the data being downloaded is known.
- a time interval of approximately one quarter of a second has passed.
- a property such as a filename has become known or changed.

If you want to be notified of data transfer events, pass a Universal Procedure Pointer (UPP) to your callback in the `notifyProc` parameter of `URLOpen`. To create a UPP to your notification callback, call the function `NewURLNotifyUPP` (page 55). For information on how to write your own notification callback function, see `URLNotifyProcPtr` (page 59). You can specify which data transfer events you want to receive in the `eventRegister` parameter of `URLOpen`. Whenever these data transfer events occur during a call to `URLOpen`, your application will be notified.

Your application's notification callback function should process the event record passed by the event parameter and return 0. The only restriction that the URL Access Manager imposes on the functionality of your notification callback function is that it should not call the function `URLDisposeReference` (page 33). For more information on how to write a notification callback function, see `URLNotifyProcPtr` (page 59).

Responding to System Events During Data Transfer

System events may occur during a call to the functions `URLSimpleDownload` (page 38), `URLDownload` (page 40), `URLSimpleUpload` (page 43), and `URLUpload` (page 45) if you indicate that a progress indicator or authentication dialog box should be displayed by these functions and the user interacts with it. To indicate that these user interfaces should be displayed, you must specify the mask constants `kURLDisplayProgressFlag` and `kURLDisplayAuthFlag` in the `openFlags` parameter.

If you want to be notified of such events, pass a Universal Procedure Pointer (UPP) to an event callback function in the `eventProc` parameter of the these functions. To create a UPP to your notification callback, call the function `NewURLSystemEventUPP`

(page 56). For information on how to write your own event callback function, see `URLSystemEventProcPtr` (page 60). If you do not create a callback function to handle system events and you specify the mask constants `kURLDisplayProgressFlag` and `kURLDisplayAuthFlag` in the `openFlags` parameter, the URL Access Manager displays a nonmovable modal progress indicator and authentication dialog box during when appropriate.

Your application's system event callback function should process the event record passed by the event parameter and return 0. The only restriction that the URL Access Manager imposes on the functionality of your application's system event callback function is that it should not call the function `URLDisposeReference` (page 33). For more information on how to write a system event callback function, see `URLSystemEventProcPtr` (page 60) and "Downloader's system event callback function" (page 28).

Using the URL Access Manager with AppleScript

You can use AppleScript to call URL Access Manager functions. If your AppleScript application uses the URL Access Manager for operations that may take a substantial amount of time, such as transferring large amounts of data over a low-speed connection, be sure to set the timeout to large value. Setting the timeout to a large value , such as 60,000 seconds, will avoid unnecessary AppleEvent errors.

For information about the standard scripting addition commands distributed with AppleScript, see the AppleScript section of the Mac OS Help Center, or visit the following web site: http://www.apple.com/applescript.

A Case Study: Downloading Data From a URL

This section describes how the sample application SamplePost posts information to an HTTP URL and download the URL's response using the URL Access Manager function `URLDownload` (page 40).

Listing 2-2 (page 21) shows the header file for the application, SamplePost.h, which contains definitions of the URL from which data is to be downloaded (`kSampleURL`) and the structure `urlDownInfo`, as well as declarations of the function `DoSamplePost`, which calls `URLDownload`, and a system event callback function, `MyURLCallbackProc`, which is a place holder for code that handles system events that occur during the download.

Listing 2-2 The SamplePost.h file

```
#define kSampleURL  "http://www.internic.net/cgi-bin/itts/whois"
typedef struct urlDownInfo *URLDownInfoPtr;

typedef struct urlDownInfo {
    URLReference urlRef;
    FSSpec * destination;
    Handle destinationHandle;
    URLOpenFlags openFlags;
    URLSystemEventProcPtr eventProc;
    void * userContext;
    Boolean done;
    OSStatus errorCode;
} URLDownloadInfo;

static void DoSamplePost ();
pascal OSStatus MyURLCallbackProc( void*, EventRecord * );
```

SamplePost is a multi-threaded application. As a result, in Listing 2-3 (page 21), SamplePost's main function calls the Memory Manager functions `MaxApplZone` and `MoreMasters` in its main function. Note that all URL Access Manager functions are threaded with Thread Manager cooperative threads. These threads are nonreentrant on PowerPC.

Listing 2-3 SamplePost's main function

```
#include <stdio.h>
#include <Events.h>
#include <Threads.h>
#include <Processes.h>
```

```
#include <Files.h>
#include "URLAccess.h"
#include "SamplePost.h"
int main (void){
    OSStatus err = noErr;
// Call MaxAppleZone() when using the Thread Manager.
    MaxApplZone();
for (i = 0; i < 20; i++) {
    MoreMasters();
}
```

Listing 2-4 (page 22) shows SamplePost calling the function `URLAccessAvailable` (page 32) to verify that the URL Access Manager is available. If the URL Access Manager is available, `DoSamplePost` is called.

Listing 2-4 Verifying the availability of the URL Access Manager

```
// Make sure the URL Access Manager is available.
if ( URLAccessAvailable()) {
    DoSamplePost();
}
else {
    // Call error handling function.
}
```

In Listing 2-5 (page 22), `DoSamplePost` defines a `URLDownloadInfo` structure named `myRef` that is uses to store information for calling `URLDownload`. The DoSamplePost function then calls NewHandle to allocate the memory in which the downloaded information will be stored, and creates a URL reference stores it in `myRef.urlRef`.

Listing 2-5 Allocating memory and creating a URL reference

```
static void DoSamplePost ( void){
    OSStatus err = noErr;
    ThreadID threadID = 0;
    URLDownloadInfo myRef;
    Handle downloadHandle = NULL;
    long downloadSize = 0;
```

```
printf( "<•>DoSamplePost() Enter\n");
    downloadHandle = NewHandle(0);
if ( downloadHandle == NULL) {
    // Call error handling function.
}
//  Create a URLReference
    err = URLNewReference( kSampleURL, &myRef.urlRef );
    if ( err != noErr) {
// Call error handling function.
}
```

As shown in Listing 2-6 (page 23), `DoSamplePost` calls the function `URLSetProperty` (page 36) to set the HTTP request method property value to the 4-byte string `"POST"` and the value of the HTTP request body property value to the 19-byte string `"whois_nic=apple.com"`. When you set the property identified by `kURLHTTPRequestBody`, the URL Access Manager automatically adds the length of the value identifed by `kURLHTTPRequestHeader` to the request, so you do not need to set the request header explicitly.

Listing 2-6 Setting URL properties

```
URLSetProperty (myRef.urlRef, kURLHTTPRequestMethod, "POST", 4);
URLSetProperty (myRef.urlRef, kURLHTTPRequestBody, "whois_nic=apple.com", 19);
```

Next, `DoSamplePost` uses the remaining fields of the `myRef` structure to store values that will be used as parameters for calling `URLDownload`.

- `DoSamplePost` sets `myRef.destination` to `NULL`. When `NULL` is provided as the destination parameter to the `URLDownload`, the calling application indicates that the downloaded data is not going to be written to a file on disk.
- `DoSamplePost` sets `myRef.destinationHandle` to the value of `downloadHandle`, which is the location in memory at which the downloaded data is to be stored.
- `DoSamplePost` sets `myRef.OpenFlags` to `kURLDisplayProgressFlag`. When the value of the `openFlags` parameter to `URLDownload` is `kURLDisplayProgressFlag`, `URLDownload` displays a progress indicator during the download process. You may wish to provide a system event callback function to handle system events that occur.

- DoSamplePost sets myRef.eventProc to the address of the SamplePost application's system event callback function. When DoSamplePost calls URLDownload, it will specify myRef.eventProc as the eventProc parameter. If a system event occurs while the progress indicator is displayed, the URL Access Manager will call the function specified by the eventProc parameter and will pass to it the value of the userContext parameter, which is described next.
- DoSamplePost sets myRef.userContext to 1. When DoSamplePost calls URLDownload, it will specify myRef.userContext as the userContext parameter. Your application can use the user context to associate any particular call of URLDownload with any particular call of the system event callback function.

Listing 2-7 (page 24) illustrates setting these values.

Listing 2-7 Setting the URLDownload parameters

```
myRef.destination = NULL;
myRef.destinationHandle = downloadHandle;
myRef.openFlags = kURLDisplayProgressFlag;
myRef.eventProc = &MyURLCallbackProc;
myRef.userContext = "1";
myRef.errorCode = 0;
```

Once the URL reference has been created, its properties set, and the parameters for URLDownload prepared, DoSamplePost is ready to call URLDownload, as shown in Listing 2-8 (page 24). If the download is successful, DoSamplePost calls the function URLGetProperty (page 35) to obtain the size of the downloaded data using the downloadSize parameter.

Listing 2-8 Calling the URLDownload function

```
err = URLDownload( myRef.urlRef,myRef.destination,myRef.destinationHandle,
                        myRef.openFlags, myRef.eventProc, myRef.userContext);
myRef.errorCode = err;
if (  myRef.errorCode != noErr) {
    // Call error handling function.
}
else {
    // Successful download. Get the size of the downloaded data.
```

```
    err = URLGetProperty(myRef.urlRef, kURLResourceSize, &downloadSize, 4);
if ( err != noErr) {
    // Call error handling function.
}
```

In Listing 2-9 (page 25) `DoSamplePost` calls `SetHandleSize` to set the size of `downloadHandle` to `downloadSize + 1` and sets the value of the last byte of downloaded data to `NULL`. Now that data can be displayed by calling the function `printf`. `DoSamplePost` concludes by disposing of the URL reference.

Listing 2-9 Displaying the downloaded data

```
downloadSize = GetHandleSize(downloadHandle);
SetHandleSize(downloadHandle, (downloadSize+1));
(*myRef.destinationHandle)[downloadSize] = NULL;
printf( "<•>==================== Downloaded Data ==================\n" );
printf( "%s", *myRef.destinationHandle );
DisposeHandle(downloadHandle);
URLDisposeReference(myref.urlRef);
}
```

Listing 2-10 (page 25) shows a placeholder for SamplePost's system event callback function. The `userContext` parameter can be used to associate any particular call of `URLDownload` with any particular call of the system event callback function.

Listing 2-10 SamplePost's system event callback function

```
pascal OSStatus MyURLCallbackProc ( void *userContext, EventRecord *event )
{
    printf( "<•>System callback thread fired! Thread: %u\n", userContext );
    return 0;
}
```

A Case Study: Downloading Data From Multiple URLs

This section describes how the sample application Downloader downloads data from multiple URLs and stores it in multiple files using the URL Access Manager function `URLDownload` (page 40). Downloader obtains the URLs to be downloaded by reading a text file in which they have been stored.

Listing 2-11 (page 26) illustrates how Downloader's main function sets up the main event loop and calls the function `getURL` to obtain a URL from a file of URLs.

Listing 2-11 The Downloader application's main function

```
#include <Events.h>
#include <stdio.h>
#include "URLAccess.h"
#include "string.h"
#include "Memory.h"

void main (void){
    OSStatus err = noErr;
    char url[255];
    int count, fileCount = 0;
    EventRecord ev;

// Call MaxApplZone, MoreMasters.
// Initialize graphics port, fonts, menus, cursor, and dialogs.
// Clear the screen.
while ( url != NULL ) {

// Handle Events through each loop
    WaitNextEvent(everyEvent, &ev, 0, NULL);
    eventHandler( NULL, &ev );

// Obtain a URL from the file of URLs
    result = getURL(url); // getURL function not shown
```

```
if ( result == eofErr ) { // Handle error condition. }

    // Call Download function.
    result = DoDownload( url );
if ( result != noErr) { // Handle error condition. }
{
printf("\n All of the URLs have been downloaded.\n");
}
```

The `DoDownload` function shown in Listing 2-12 (page 27) does the actual work of downloading data from the URL. It creates a file specification for the data that is to be downloaded and a URL reference. It specifies the mask kURLReplaceExistingFlag in the `openFlags` parameter to replace an existing file (if any) with the downloaded data and to display a progress indicator during the download. Lastly, it calls the function `URLDownload` (page 40) to download the data.

Listing 2-12 Downloader's `DoDownload` function

```
void DoDownload (void){
    URLReference urlRef;
    FSSpec dest, *destPtr = NULL;
    destPtr = &dest;
    Handle destHandle = NULL;
    int openFlags = kURLReplaceExistingFlag + kURLDisplayProgressFlag;
    Str255 newFile;

// Create the file specification for the download.
    sprintf((char*)newFile, "File %d", fileCount);
    c2pstr((char*)newFile);
    fileCount++;
    err = FSMakeFSSpec(0, 0, newFile, &dest);

// Create the URLReference.
    err = URLNewReference( theURL, &urlRef );
    if (err != noErr) printf("URLNewReference failed\n");

// Download the data.
    err = URLDownload( urlRef, destPtr, destHandle, openFlags, &eventHandler,
                        (void*)&fileCount );
    if (err != noErr) printf("URLDownload failed\n");
```

```
// Clean up.
    err = URLDisposeReference( urlRef );
    if (err != noErr) printf("URLDisposeReference failed\n");
    return err;
}
```

Listing 2-10 (page 25) illustrates Downloader's general `event handling function` `eventHandler`. This function handles system events that might occur during calls to the functions `URLSimpleDownload` (page 38), `URLDownload` (page 40), `URLSimpleUpload` (page 43), and `URLUpload` (page 45). The `userContext` parameter can be used to associate any particular call of `URLDownload` with any particular call of the system event callback function. In this context, it is an integer.

Listing 2-13 Downloader's system event callback function

```
pascal long eventHandler( void * userContext, EventRecord* eventPtr ){
EventRecord* ev;
int what = 0;
int context = 0;
int* intPtr = NULL;

// Convert the event pointer into an event record.
ev = (EventRecord*)eventPtr;
what = ev->what;

// Convert the void* to an integer.
intPtr = (int*)userContext;
context = *intPtr;
if (context < 0 || context > 99)
    context = -1; // Unknown context
switch (what) {
    case 0 : // Null Event
        break;
    case mouseDown:
        printf("Handler Called: mouseDown  User Context: %d\n", context);

// Call function to handle event.
        break;
```

```
    case updateEvt:
        printf("Handler Called: updateEvt  User Context: %d\n", context);

// Call function to handle event.
        break;
    case activateEvt:
        printf("Handler Called: activateEvt  User Context: %d\n", context);

// Call function to handle event.
        break;
    case keyDown:
        printf("Handler Called: keyDown  User Context: %d\n", context);

    // Call function to handle event.
        break;
    default:
        printf("Handler Called: Default  User Context: %d\n", context);
        break;
}
return NULL;
}
```

CHAPTER 2

URL Access Manager Tasks

URL Access Manager Reference

The following sections provide a complete description of the URL Access Manager 2.0.3 API, including functions, callbacks, data types, constants, and result codes.

URL Access Functions

Determining Availability and Version Information

`URLAccessAvailable` — Indicates whether the URL Access Manager is available. (page 32)

`URLGetURLAccessVersion` — Determines the version of URL Access Manager installed on the user's system. (page 32)

URLAccessAvailable

Indicates whether the URL Access Manager is available.

```
Boolean URLAccessAvailable ();
```

function result A `Boolean` value indicating whether the URL Access Manager is available. If `true`, your application can call URL Access Manager functions. You should call the `URLAccessAvailable` function to determine whether the URL Access Manager is available before calling any other URL Access Manager functions. In Mac OS X, `URLAccessAvailable` is always available.

Discussion

The URL Access Manager is available on computers running Mac OS 9.0, but is not currently supported in Carbon. As a result, in Mac OS X, you should weak link against the shared library and test to see if the functions are there.

URLGetURLAccessVersion

Determines the version of URL Access Manager installed on the user's system.

```
OSStatus URLGetURLAccessVersion (UInt32 *returnVers);
```

`returnVers`

On return, a pointer to the version number of the URL Access Manager installed on the user's system.

Creating and Disposing of URL References

`URLNewReference` — Creates a URL reference. (page 33)

URLDisposeReference — Disposes of the memory associated with a URL reference. (page 33)

URLNewReference

Creates a URL reference.

```
OSStatus URLNewReference (
                    const char *url,
                    URLReference *urlRef);
```

url

A pointer to a C string representing the name of the URL you want to create a reference for.

urlRef

On return, a pointer to the newly-created URL reference.

Discussion

The URLNewReference function creates a URL reference that you can use in subsequent calls to the URL Access Manager. When you no longer need a URL reference, you should dispose of its memory by calling the function URLDisposeReference (page 33).

URLDisposeReference

Disposes of the memory associated with a URL reference.

```
OSStatus URLDisposeReference  (URLReference urlRef);
```

urlRef

A reference to the URL whose associated memory you wish to dispose of. You should call the URLDisposeReference function to release the memory occupied by a URL reference when you are finished with it.

Special Considerations

You must call the URLDisposeReference function to dispose of the reference associated with a URL reference even if the data transfer operation fails. Failure to call URLDisposeReference may result in thread or memory leaks.

Getting and Setting Information About a URL

You can use these functions to set and obtain information about a URL or the resource the URL points to:

`URLGetPropertySize` — Determines the size of a URL property. (page 34)

`URLGetProperty` — Obtains the value of a URL property. (page 35)

`URLSetProperty` — Sets the value of a URL property. (page 36)

URLGetPropertySize

Determines the size of a URL property.

```
OSStatus URLGetPropertySize (
                    URLReference urlRef,
                    const char *property,
                    Size *propertySize);
```

`urlRef`

A reference to the URL whose property size you want to determine.

`property`

A pointer to a C string representing the name of the property value whose size you want to determine. For a description of property name constants, see "Universal URL Property Name Constants" (page 78) and "HTTP and HTTPS URL Property Name Constants" (page 76).

`propertySize`

On return, a pointer to the size (in bytes) of the specified property value. If the size is not available, `URLGetPropertySize` passes back `-1` in this parameter.

Discussion

The `URLGetProperty` function obtains the size of the property value identified by the property name constant passed in the property parameter. For a description of property name constants and data types of the corresponding property values, see "Universal URL Property Name Constants" (page 78) and "HTTP and HTTPS URL Property Name Constants" (page 76).

You should call the `URLGetPropertySize` function before calling the functions `URLGetProperty` (page 35) and `URLSetProperty` (page 36) to determine the size of the buffer containing the property value you wish to obtain or set. Pass the returned value in the `bufferSize` parameter of these functions.

URLGetProperty

Obtains the value of a URL property.

```
OSStatus URLGetProperty (
                    URLReference urlRef,
                    const char *property,
                    void *propertyBuffer,
                    Size bufferSize);
```

`urlRef`

A reference to the URL whose property value you want to determine.

`property`

A pointer to a C string representing the name of the property value you want to determine. For a description of property name constants and their corresponding data types, see "Universal URL Property Name Constants" (page 78) and "HTTP and HTTPS URL Property Name Constants" (page 76).

`propertyBuffer`

A pointer to a buffer containing the property value you want to obtain. You must also pass the correct data type of the property value you wish to obtain. Before calling `URLGetProperty`, allocate enough memory in this buffer to contain the property value you wish to obtain. On return, a pointer to a buffer containing the property value. If you do not allocate enough memory for the buffer, `URLGetProperty` does not pass back the property value in this parameter and returns the result code `kURLPropertyBufferTooSmallError`.

`bufferSize`

The size (in bytes) of the buffer pointed to by `propertyBuffer`. To determine the buffer size, call the function `URLGetPropertySize` (page 34). If the buffer size is too small, `URLGetProperty` returns the result code `kURLPropertyBufferTooSmallError` and does not pass back the property value in the `propertyBuffer` parameter.

function result The result code `kURLPropertyBufferTooSmallError` indicates that you did not allocate enough memory for the buffer in the `propertyBuffer` parameter. The result code `kURLPropertyNotYetKnownError` indicates that the value of the property is not yet available.

Discussion

The `URLGetProperty` function obtains the value of a URL property identified by the property name constant specified in the property parameter. Note that you must also pass the correct data type of the property value in the `propertyBuffer` parameter so you know how to display it.

URLSetProperty

Sets the value of a URL property.

```
OSStatus URLSetProperty (
                    URLReference urlRef,
                    const char *property,
                    void *propertyBuffer,
                    Size bufferSize);
```

`urlRef`

A reference to the URL whose property value you want to set.

`property`

A pointer to a C string representing the name of the property value you want to set. You can only set property values identified by the constants `kURLFileType`, `kURLFileCreator`, `kURLUserName`, `kURLHTTPRequestMethod`, `kURLHTTPRequestHeader`, `kURLHTTPRequestBody`, and `kURLHTTPUserAgent`. For a description of these property name constants and their corresponding data types, see “Universal URL Property Name Constants” (page 78) and “HTTP and HTTPS URL Property Name Constants” (page 76). To clear the value of a property, pass `NULL` in this parameter.

`propertyBuffer`

A pointer to a buffer containing the property value you want to set. You must also pass the correct data type of the property value you wish to set.

`bufferSize`

The size (in bytes) of the buffer pointed to by `propertyBuffer`. To determine the buffer size, call the function `URLGetPropertySize` (page 34). If the buffer size is too small, `URLSetProperty` returns the result code `kURLPropertyBufferTooSmallError` and does not pass back the property value in the `propertyBuffer` parameter.

function result If there is a function error, `URLSetProperty` will not set the property value. The result code `kURLUnsettablePropertyError` indicates that a property value cannot be set. The result code `kURLUnknownPropertyError` indicates that a property is invalid or undefined.

Discussion

The `URLSetProperty` function enables you to set those property values identified by the following constants: `kURLFileType`, `kURLFileCreator`, `kURLUserName`, `kURLPassword`, `kURLHTTPRequestMethod`, `kURLHTTPRequestHeader`, `kURLHTTPRequestBody`, and `kURLHTTPUserAgent`. For a description of these property name constants and their corresponding data types, see “Universal URL Property Name Constants” (page 78) and “HTTP and HTTPS URL Property Name Constants” (page 76).

You may wish to call `URLSetProperty` before calling the function `URLDownload` (page 40) or `URLUpload` (page 45) to set a URL property before a data transfer operation.

Performing Simple Data Transfer

You can use these high-level function to download from and upload to a URL Unlike the function `URLOpen` (page 47), these functions are synchronous, returning control to your application only after the function is done executing.

`URLSimpleDownload` — Downloads data from a URL specified by a character string. (page 38)

`URLDownload` — Downloads data from a URL specified by a URL reference. (page 40)

`URLSimpleUpload` — Uploads a file or directory to an FTP URL specified by a character string. (page 43)

`URLUpload` — Uploads a file or directory to an FTP URL specified by a URL reference. (page 45)

URLSimpleDownload

Downloads data from a URL specified by a character string.

```
OSStatus URLSimpleDownload (
                    const char *url,
                    FSSpec *destination,
                    Handle destinationHandle,
                    URLOpenFlags openFlags,
                    URLSystemEventUPP eventProc,
                    void *userContext);
```

`url`

A pointer to a C string representing the pathname of the URL from which data is to be downloaded. If the pathname specifies a file, the file is downloaded regardless of whether you specify `kURLDirectoryListingFlag` or `KURLIsDirectoryHintFlag` in the `openFlags` parameter.

`destination`

A pointer to a file specification structure that identifies the file or directory into which data is to be downloaded. If you wish to download data into memory, pass `NULL` in this parameter and a valid handle in the `destinationHandle` parameter. If you pass a file specification that does not identify a file or directory, the name of the file or directory specified by the pathname in the url parameter is used. If you pass a file or directory that already exists, and do not specify `kURLReplaceExistingFlag` in the `openFlags` parameter, `URLSimpleDownload` creates a new file or directory whose name has a number appended before the extension. For example, if the URL specifies a file named `file.txt`, `URLSimpleDownload` changes the filename to `file1.txt`.

`destinationHandle`

A handle to the destination in memory where you want the data downloaded. Before calling `URLDownload`, create a zero-sized handle. If you wish to download data into a file or directory, pass `NULL` in this parameter and a valid file specification in the `destination` parameter.

`openFlags`

A bitmask that indicates the data transfer options to use. You can specify any of the following masks for downloading options: `kURLReplaceExistingFlag`, `kURLExpandFileFlag`, `kURLExpandAndVerifyFlag`, `kURLDisplayProgressFlag`, `kURLDisplayAuthFlag`, `kURLIsDirectoryHintFlag`,

kURLDoNotTryAnonymousFlag, and kURLDirectoryListingFlag. See "Data Transfer Options Mask Constants" (page 71) for a description of possible values.

eventProc

A Universal Procedure Pointer (UPP) to your system event callback function, if one exists. For information on how to write a system event callback, see URLSystemEventProcPtr (page 60). If you want to handle events that occur while a progress indicator or authentication dialog box is being displayed, specify the appropriate mask (either kURLDisplayProgressFlag or kURLDisplayAuthFlag) in the openFlags parameter and pass a UPP to your callback function in this parameter. Pass NULL if you do not want to receive notification of these events. In this case, the URL Access Manager displays a nonmovable modal progress indicator or authentication dialog box.

userContext

A pointer to application-defined storage that will be passed to your system event callback function, if one exists. Your application can use this to associate any particular call of URLSimpleDownload with any particular call of the system event callback function.

function result If your application is multi-threaded, and more than one thread calls URLSimpleDownload simultaneously, URLSimpleDownload returns the result code kURLProgessAlreadyDisplayedError if you specify kURLDisplayProgressFlag in the openFlags parameter and the URL Access Manager is already displaying a progress indicator.

Discussion

The URLSimpleDownload function downloads data from a URL specified by a pathname to a specified file, directory, or memory. It does not return until the download is complete. If you want to download data from a URL identified by a reference rather than a pathname, call the function URLDownload (page 40). The difference between the two functions is that URLDownload allows you to set URL property values by calling the function URLSetProperty (page 36) prior to the call. If you want more control over a data transfer operation, call the function URLOpen (page 47).

If you wish to download data to a file or directory, pass a valid file specification in the destination parameter. If you instead wish to download data to memory, pass a valid handle in the destinationHandle parameter.

When URLSimpleDownload downloads data from a URL that represents a local file (that is, a URL that begins with file:/ / /), the data fork is downloaded but the resource fork is not.

Version Notes

In Mac OS 8.6, if the file type of the file to be downloaded is unknown, URLSimpleDownload assumes that the file is of type text. In this case, the end-of-line character is changed to the Mac style end-of-line character 0x0d0x0a. In Mac OS 9, when a file of unknown type is downloaded, URLSimpleDownload does not assume that it is a text file. Instead, it identifies it as an unknown file type and does not change the end-of-line character.

Special Considerations

URLSimpleDownload yields time to other threads. Your application should call URLSimpleDownload from a thread other than the main thread so that other processes have time to run.

URLDownload

Downloads data from a URL specified by a URL reference.

```
OSStatus URLDownload (
                    URLReference urlRef,
                    FSSpec *destination,
                    Handle destinationHandle,
                    URLOpenFlags openFlags,
                    URLSystemEventUPP eventProc,
                    void *userContext);
```

urlRef

A reference to the URL from which data is to be downloaded. You call URLDownload, you cannot use the same reference if you call URLDownload again. Instead, you must create a new URL reference by calling the function URLNewReference (page 33).

destination

A pointer to a file specification structure that identifies the file or directory into which data is to be downloaded. If you wish to download data into memory, pass NULL in this parameter and a valid handle in the destinationHandle parameter. If you pass a file specification that does not identify a file or directory, the name of the file or directory specified by the pathname in the urlRef parameter is used. If you want to replace the file you pass in this parameter,

terminate the pathname with a slash character (/), and specify `kURLReplaceExistingFlag` in the `openFlags` parameter. If you specify a name that already exists on the server and do not specify `kURLReplaceExistingFlag`, `URLDownload` returns the result code `kURLDestinationExistsError`. If you do not specify a name, do not specify `kURLReplaceExistingFlag` in the `openFlags` parameter, and the name already exists on the server, the URL Access Manager creates a unique name by appending a number to the original name before the extension, if any. For example, if the URL specifies a file named `file.txt`, `URLDownload` changes the filename to `file1.txt`.

`destinationHandle`

A handle to the destination in memory where you want the data downloaded. Before calling `URLDownload`, create a zero-sized handle. If you wish to download data into a file or directory, pass `NULL` in this parameter and a valid file specification in the `destination` parameter.

`openFlags`

A bitmask that indicates the data transfer options to use. You can specify any of the following masks for downloading options: `kURLReplaceExistingFlag`, `kURLExpandFileFlag`, `kURLExpandAndVerifyFlag`, `kURLDisplayProgressFlag`, `kURLDisplayAuthFlag`, `kURLIsDirectoryHintFlag`, `kURLDoNotTryAnonymousFlag`, and `kURLDirectoryListingFlag`. See “Data Transfer Options Mask Constants” (page 71) for a description of possible values.

`eventProc`

A Universal Procedure Pointer (UPP) to your system event callback function, if one exists. For information on how to write a system event callback, see `URLSystemEventProcPtr` (page 60). If you want to handle events that occur while a progress indicator or authentication dialog box is being displayed, specify the appropriate mask (either `kURLDisplayProgressFlag` or `kURLDisplayAuthFlag`) in the `openFlags` parameter and pass a UPP to your callback function in this parameter. Pass `NULL` if you do not want to receive notification of these events. In this case, the URL Access Manager displays a nonmovable modal progress indicator or authentication dialog box.

`userContext`

A pointer to application-defined storage that will be passed to your system event callback function, if one exists. Your application can use this to associate any particular call of `URLDownload` with any particular call of the system event callback function.

function result If your application is multi-threaded, and more than one thread calls `URLDownload` simultaneously, `URLDownload` returns the result code `kURLProgessAlreadyDisplayedError` if you specify `kURLDisplayProgressFlag` in the `openFlags` parameter and the URL Access Manager is already displaying a progress indicator.

Discussion

The `URLDownload` function downloads data from a URL specified by a URL reference to a file, directory, or memory. It does not return until the download is complete. If you want to download data from a URL identified by a pathname rather than a reference, call the function `URLSimpleDownload` (page 38). The difference between the two functions is that URLDownload allows you to set URL property values by calling the function `URLSetProperty` (page 36) prior to the call. If you want more control over a data transfer operation, call the function `URLOpen` (page 47).

If you wish to download data to a file or directory, pass a valid file specification in the `destination` parameter. If you instead wish to download data to memory, pass a valid handle in the `destinationHandle` parameter. If the URL specified in the `urlRef` parameter is a file, the file is uploaded regardless of whether the bit specified by the mask constant `kURLDirectoryListingFlag` or `KURLIsDirectoryHintFlag` is set in the `openFlags` parameter.

When `URLDownload` downloads data from a URL that represents a local file (that is, a URL that begins with file:///), the data fork is downloaded but the resource fork is not.

Version Notes

In Mac OS 8.6, if the file type of the file to be downloaded is unknown, `URLDownload` assumes that the file is of type text. In this case, the end-of-line character is changed to the Mac style end-of-line character `0x0d0x0a`. In Mac OS 9, when a file of unknown type is downloaded, `URLDownload` does not assume that it is a text file. Instead, it identifies it as an unknown file type and does not change the end-of-line character.

Special Considerations

`URLDownload` yields time to other threads. Your application should call `URLDownload` from a thread other than the main thread so that other processes have time to run.

URLSimpleUpload

Uploads a file or directory to an FTP URL specified by a character string.

```
OSStatus URLSimpleUpload (
                    const char *url,
                    const FSSpec *source,
                    URLOpenFlags openFlags,
                    URLSystemEventUPP eventProc,
                    void *userContext);
```

`url`

A pointer to a C string representing the pathname of the URL to which a file or directory is to be uploaded. If you wish to replace the destination directory of this URL with the file or directory that you pass in the source parameter, terminate this pathname with a slash character (/), and specify `kURLReplaceExistingFlag` in the `openFlags` parameter. If you specify a name that already exists on the server and do not specify `kURLReplaceExistingFlag`, `URLSimpleUpload` returns the result code `kURLDestinationExistsError`. If you do not specify a name, do not specify `kURLReplaceExistingFlag` in the `openFlags` parameter, and the name already exists on the server, the URL Access Manager creates a unique name by appending a number to the original name before the extension, if any. For example, if the URL specifies a file named `file.txt`, `URLSimpleUpload` changes the filename to `file1.txt`.

`source`

A pointer to a file specification structure that describes the file or directory you want to upload.

`openFlags`

A bitmask that indicates the data transfer options to use. You can specify any of the following masks for uploading options: `kURLReplaceExistingFlag`, `kURLBinHexFileFlag`, `kURLDisplayProgressFlag`, `kURLDisplayAuthFlag`, and `kURLDoNotTryAnonymousFlag`. See “Data Transfer Options Mask Constants” (page 71) for a description of possible values.

`eventProc`

A Universal Procedure Pointer (UPP) to your system event callback function, if one exists. For information on how to write a system event callback, see `URLSystemEventProcPtr` (page 60). If you want to handle events that occur while a progress indicator or authentication dialog box is being displayed, specify the appropriate mask (either

kURLDisplayProgressFlag or kURLDisplayAuthFlag) in the openFlags parameter and pass a UPP to your callback function in this parameter. Pass NULL if you do not want to receive notification of these events. In this case, the URL Access Manager displays a nonmovable modal progress indicator or authentication dialog box.

userContext

A pointer to application-defined storage that will be passed to your system event callback function, if one exists. Your application can use this to associate any particular call of URLSimpleUpload with any particular call of the system event callback function.

function result The result code kURLDestinationExistsError indicates that you specified a pathname that already exists on the server but did not set the bit specified by the mask constant kURLReplaceExistingFlag in the openFlags parameter. If your application is multi-threaded, and more than one thread calls URLSimpleUpload simultaneously, URLSimpleUpload returns the result code kURLProgessAlreadyDisplayedError if you specify kURLDisplayProgressFlag in the openFlags parameter and the URL Access Manager is already displaying a progress indicator.

Discussion

The URLSimpleUpload function uploads a file or directory to an FTP URL specified by a pathname. It does not return until the upload is complete. If you want to upload data from a URL identified by a reference rather than a pathname, call the function URLUpload (page 45). The difference between the two functions is that URLUpload allows you to set URL property values by calling the function URLSetProperty (page 36) prior to the call. If you want more control over a data transfer operation, call the function URLOpen (page 47).

When URLSimpleUpload uploads data to a URL that represents a local file (that is, a URL that begins with file:/ / /), the data fork is uploaded but the resource fork is not.

Version Notes

In Mac OS 8.6, if the file type of the file to be uploaded is unknown, URLSimpleUpload assumes that the file is of type text. In this case, the end-of-line character is changed to the Mac style end-of-line character 0x0d0x0a. In Mac OS 9, when a file of unknown type is uploaded, URLSimpleUpload does not assume that it is a text file. Instead, it identifies it as an unknown file type and does not change the end-of-line character.

Special Considerations

`URLSimpleUpload` yields time to other threads. Your application should call `URLSimpleUpload` from a thread other than the main thread so that other processes have time to run.

URLUpload

Uploads a file or directory to an FTP URL specified by a URL reference.

```
OSStatus URLUpload (
                    URLReference urlRef,
                    const FSSpec *source,
                    URLOpenFlags openFlags,
                    URLSystemEventUPP eventProc,
                    void *userContext);
```

`urlRef`

A reference to the URL to which data is to be uploaded. You cannot use the same reference if you call `URLUpload` again. Instead, you must create a new URL reference by calling the function `URLNewReference` (page 33). If the URL refers to a file, the file is uploaded regardless of whether you specify `kURLDirectoryListingFlag` or `KURLIsDirectoryHintFlag` in the `openFlags` parameter.

`source`

A pointer to a file specification structure that describes the file or directory you want to upload.

`openFlags`

A bitmask that indicates the data transfer options you want used. You can specify any of the following masks for uploading options: `kURLReplaceExistingFlag`, `kURLBinHexFileFlag`, `kURLDisplayProgressFlag`, `kURLDisplayAuthFlag`, and `kURLDoNotTryAnonymousFlag`. See “Data Transfer Options Mask Constants” (page 71) for a description of possible values.

`eventProc`

A Universal Procedure Pointer (UPP) to your system event callback function, if one exists. For information on how to write a system event callback, see `URLSystemEventProcPtr` (page 60). If you want to handle events that occur while a progress indicator or authentication dialog box is being displayed, specify the appropriate mask (either `kURLDisplayProgressFlag` or `kURLDisplayAuthFlag`) in the `openFlags` parameter and pass a UPP to your callback function in this

parameter. Pass `NULL` if you do not want to receive notification of these events. In this case, the URL Access Manager displays a nonmovable modal progress indicator or authentication dialog box.

`userContext`

A pointer to application-defined storage that will be passed to your system event callback function, if one exists. Your application can use this to associate any particular call of URLUpload with any particular call of the system event callback function.

function result The result code `kURLDestinationExistsError` indicates that you specified a pathname that already exists on the server but did not specify `kURLReplaceExistingFlag` in the `openFlags` parameter. If your application is multi-threaded, and more than one thread calls `URLUpload` simultaneously, `URLUpload` returns the result code `kURLProgessAlreadyDisplayedError` if you specify `kURLDisplayProgressFlag` in the `openFlags` parameter and the URL Access Manager is already displaying a progress indicator.

Discussion

The `URLUpload` function uploads a file or directory to an FTP URL specified by a URL reference. It does not return until the upload is complete. If you want to upload data from a URL identified by a pathname rather than a reference, call the function `URLSimpleUpload` (page 43). The difference between the two functions is that `URLUpload` allows you to set URL property values by calling the function `URLSetProperty` (page 36) prior to the call. If you want more control over a data transfer operation, call the function `URLOpen` (page 47).

If the URL specified in the `urlRef` parameter is a file, the file is uploaded regardless of whether you specified `kURLDirectoryListingFlag` or `KURLIsDirectoryHintFlag` in the `openFlags` parameter.

When `URLUpload` uploads data to a URL that represents a local file (that is, a URL that begins with file:/ / /), the data fork is uploaded but the resource fork is not.

Version Notes

In Mac OS 8.6, if the file type of the file to be uploaded is unknown, `URLUpload` assumes that the file is of type text. In this case, the end-of-line character is changed to the Mac style end-of-line character `0x0d0x0a`. In Mac OS 9, when a file of unknown type is uploaded, `URLUpload` does not assume that it is a text file. Instead, it identifies it as an unknown file type and does not change the end-of-line character.

Special Considerations

`URLUpload` yields time to other threads. Your application should call URLUpload from a thread other than the main thread so that other processes have time to run.

Controlling Data Transfer

You can use these functions to have more control over a data transfer operation than is afforded by the functions `URLSimpleDownload` (page 38), `URLDownload` (page 40), `URLSimpleUpload` (page 43), and `URLUpload` (page 45). These functions are asynchronous, returning control to your application immediately.

`URLOpen` — Opens a URL and starts an asynchronous download or upload operation. (page 47)

`URLAbort` — Terminates a data transfer operation. (page 49)

`URLGetDataAvailable` — Determines the amount of data available for retrieval in a download operation. (page 50)

`URLGetBuffer` — Obtains the next buffer of data in a download operation. (page 51)

`URLReleaseBuffer` — Releases a buffer. (page 52)

URLOpen

Opens a URL and starts an asynchronous download or upload operation.

```
OSStatus URLOpen (
                    URLReference urlRef,
                    FSSpec *fileSpec,
                    URLOpenFlags openFlags,
                    URLNotifyUPP notifyProc,
                    URLEventMask eventRegister,
                    void *userContext);
```

`urlRef`

A reference to the URL to or from which you wish to transfer data. You cannot use the same reference if you call `URLOpen` again. Instead, you must create a new URL reference by calling the function `URLNewReference` (page 33). If the URL refers to a file, the file is uploaded regardless of whether you specify `kURLDirectoryListingFlag` or `KURLIsDirectoryHintFlag` in the `openFlags` parameter.

`fileSpec`

A pointer to a file specification that identifies the file or directory into which data is to be uploaded or downloaded. For upload operations, you must pass a valid file specification. For download operations, you can pass `NULL`. In this case, you must call the function `URLGetBuffer` (page 51) to retrieve the data as it is downloaded. For more information, see the discussion.

`openFlags`

A bitmask that indicates the data transfer options to use. You can specify any of the following masks when uploading data: `kURLReplaceExistingFlag`, `kURLBinHexFileFlag`, `kURLDisplayProgressFlag`, `kURLDisplayAuthFlag`, and `kURLDoNotTryAnonymousFlag`. You can specify any of the following masks when downloading data: `kURLReplaceExistingFlag`, `kURLExpandFileFlag`, `kURLExpandAndVerifyFlag`, `kURLDisplayProgressFlag`, `kURLDisplayAuthFlag`, `kURLIsDirectoryHintFlag`, `kURLDoNotTryAnonymousFlag`, and `kURLDirectoryListingFlag`. See “Data Transfer Options Mask Constants” (page 71) for a description of possible values.

`notifyProc`

A Universal Procedure Pointer (UPP) to your notification callback function, described in `URLNotifyProcPtr` (page 59). You can define an event notification callback function if you do not specify a file in which to store the data. You indicate the type of data transfer events you want to receive by passing a bitmask of the desired events in the `eventRegister` parameter. Pass `NULL` in this parameter if you do not want to receive notification of data transfer events.

`eventRegister`

A bitmask that `URLOpen` will test to determine the data transfer events your notification callback function (if one exists) will receive. See “Data Transfer Event Mask Constants” (page 67) for a description of this mask.

`userContext`

A pointer to application-defined storage that will be passed to your notification callback function, if one exists. Your application can use this to associate any particular call of URLOpen with any particular call of the notification event callback function.

Discussion

The `URLOpen` function starts an asynchronous download or upload operation and returns control to your application immediately. For download operations, if you do not specify a file in which to store the data, you must repeatedly call the function `URLGetBuffer` (page 51). This function retrieves the next buffer of data from the URL Access Manager's buffers so that you can manipulate the data or write it to the destination of your choice. If you do not specify a file in which to store the data, you can define an event notification function and the events for which you want to receive notification.

For download operations, if you specify a valid file, your application will not have access to buffer state information provided by the function `URLGetCurrentState` (page 53) and cannot call the functions `URLGetBuffer` (page 51), `URLReleaseBuffer` (page 52), or `URLGetDataAvailable` (page 50).

If you want to replace the file you passed in the `fileSpec` parameter, terminate the pathname with a slash character (/), and specify `kURLReplaceExistingFlag` in the `openFlags` parameter. If you specify a name that already exists on the server and do not specify `kURLReplaceExistingFlag`, `URLOpen` returns the result code `kURLDestinationExistsError`. If you do not specify a name, do not specify `kURLReplaceExistingFlag` in the `openFlags` parameter, and the name already exists on the server, the URL Access Manager creates a unique name by appending a number to the original name before the extension, if any. For example, if the URL specifies a file named `file.txt`, `URLOpen` changes the filename to `file1.txt`.

When `URLOpen` uploads data to or downloads data from a URL that represents a local file (that is, a URL that begins with file:///), the data fork is uploaded or downloaded but the resource fork is not.

Version Notes

In Mac OS 8.6, if the file type of the file to be uploaded or downloaded is unknown, `URLOpen` assumes that the file is of type text. In this case, the end-of-line character is changed to the Mac style end-of-line character `0x0d0x0a`. In Mac OS 9, when a file of unknown type is uploaded or downloaded, `URLOpen` does not assume that it is a text file. Instead, it identifies it as an unknown file type and does not change the end-of-line character.

URLAbort

Terminates a data transfer operation.

```
OSStatus URLAbort (URLReference urlRef);
```

urlRef

A reference to the URL whose data transfer operation you wish to terminate.

Discussion

The URLAbort function terminates a data transfer operation that was started by calling the function URLOpen (page 47). When your application calls URLAbort, the URL Access Manager changes the state returned by the function URLGetCurrentState (page 53) to kURLAbortingState and passes the constant kURLAbortInitiatedEvent to your notification callback function. When data transfer is terminated, the URL Access Manager changes the state returned by URLGetCurrentState to kURLCompletedState and passes the constant kURLCompletedEvent in the event parameter of your notification callback function.

URLGetDataAvailable

Determines the amount of data available for retrieval in a download operation.

```
OSStatus URLGetDataAvailable (
                    URLReference urlRef,
                    Size *dataSize);
```

urlRef

A reference to the URL whose buffer size you want.

dataSize

On return, a pointer to the size (in bytes) of data available for retrieval in a download operation.

Discussion

The URLGetDataAvailable function determines the number of bytes remaining in a buffer after a call to the function URLGetBuffer (page 51). The returned size does not include the number of bytes in transit to a buffer, nor does it include the amount of data not yet transferred from the URL. You should only call URLGetDataAvailable if you have initiated a download process.

URLGetDataAvailable does not calculate the amount of data remaining to be downloaded. If you wish to determine this, pass the name constant kURLResourceSize in the property parameter of the function URLGetProperty (page 35) and subtract the amount of data copied.

URLGetBuffer

Obtains the next buffer of data in a download operation.

```
OSStatus URLGetBuffer (
                    URLReference urlRef,
                    void **buffer,
                    Size *bufferSize);
```

`urlRef`
A reference to the URL whose data is being downloaded.

`buffer`
On return, a handle to a buffer containing the downloaded data.

`bufferSize`
On return, a pointer to the number of bytes of data in the buffer.

Discussion

The `URLGetBuffer` function obtains the next buffer of data in a download operation. `URLGetBuffer` does not enable you to retain or modify the transferred data. If you pass `NULL` in the `fileSpec` parameter of the function `URLOpen` (page 47), you should call `URLGetBuffer` to retrieve data as it is downloaded.

You should call `URLGetBuffer` repeatedly until URL Access Manager passes the event constant `kURLCompletedEvent` or `kURLAbortInitiatedEvent` in the `event` parameter of your notification callback function, or until the function `URLGetCurrentState` (page 53) returns the state constant `kURLTransactionComplete` or `kURLAbortingState`. Between calls to `URLGetBuffer`, you should call the function `URLIdle` (page 54) to allow time for the URL Access Manager to refill its buffers.

To determine the number of bytes remaining in the buffer, call the function `URLGetDataAvailable` (page 50). The size returned by `URLGetDataAvailable` does not include the number of bytes in transit to a buffer, nor does it include the amount of data not yet transferred from the URL.

Special Considerations

You should release the returned buffer as soon as possible after a call to `URLGetBuffer` by calling the function `URLReleaseBuffer` (page 52). This prevents the URL Access Manager from running out of buffers.

URLReleaseBuffer

Releases a buffer.

```
OSStatus URLReleaseBuffer (
                    URLReference urlRef,
                    void *buffer);
```

`urlRef`
A reference to the URL whose buffer you want to release.

`buffer`
A pointer to the buffer you want to release.

Discussion

The `URLReleaseBuffer` function releases the buffer obtained by calling the function `URLGetBuffer` (page 51). To prevent the URL Access Manager from running out of buffers, you should call `URLReleaseBuffer` after each call to `URLGetBuffer`.

Obtaining Information About a Data Transfer Operation

You can use these functions to determine the error code returned when a data transfer operation fails, determine the status of a data transfer operation, yield time so that the URL Access Manager can refill its buffers, or get information about a file.

`URLGetError` — Determines the error code of a failed data transfer operation. (page 52)

`URLGetCurrentState` — Determines the status of a data transfer operation. (page 53)

`URLGetFileInfo` — Obtains the file type and creator of a file. (page 54)

`URLIdle` — Gives the URL Access Manager time to refill its buffers during download operations. (page 54)

URLGetError

Determines the error code of a failed data transfer operation.

```
OSStatus URLGetError (
                    URLReference urlRef,
                    OSStatus *urlError);
```

`urlRef`

A reference to the URL whose data transfer operation failed.

`urlError`

A pointer to a C string representing the name of the error code returned by the failed operation.

Discussion

The `URLGetError` function determines the error code returned when a data transfer operation fails. The error code may be a system error code, a protocol-specific error code, or one of the error codes listed in "URL Access Result Codes" (page 80).

URLGetCurrentState

Determines the status of a data transfer operation.

```
OSStatus URLGetCurrentState (
                    URLReference urlRef,
                    URLState *state);
```

`urlRef`

A reference to the URL whose data transfer state you want to determine.

`state`

On return, a pointer to the state of data transfer. See "Data Transfer State Constants" (page 74) for a description of possible values. All constants except `kURLDataAvailableState` and `kURLCompletedState` can be returned at any time. If you pass a valid file specification in the `fileSpec` parameter of the function `URLOpen` (page 47), your notification callback function will not be notified of data available and transaction completed states as identified by the constants `kURLDataAvailableState` and `kURLTransactionCompleteState`.

Discussion

The `URLGetCurrentState` function determines the current status of a data transfer operation. You may wish to call `URLGetCurrentState` periodically to monitor the status of a download or upload operation.

URLGetFileInfo

Obtains the file type and creator of a file.

```
OSStatus URLGetFileInfo (
                    StringPtr fName,
                    OSType *fType,
                    OSType *fCreator);
```

`fName`

A pointer to a Pascal string representing the name of the file for which you want information.

`fType`

On return, a pointer to the file type code of the specified filename.

`fCreator`

On return, a pointer to the file creator code of the specified filename.

Discussion

The `URLGetFileInfo` function obtains the file type and creator codes for a specified filename. The type and creator codes are determined by the Internet configuration mapping table and are based on the filename extension. For example, if you pass the filename "`jane.txt`", `URLGetFileInfo` will return `'TEXT'` in the `type` parameter and `'ttxt'` in the `creator` parameter.

URLIdle

Gives the URL Access Manager time to refill its buffers during download operations.

```
OSStatus URLIdle (void);
```

Discussion

The `URLIdle` function gives the URL Access Manager time to refill its buffers during download operations. If you pass `NULL` in the `fileSpec` parameter of the function `URLOpen` (page 47), you should call `URLGetBuffer` (page 51) to retrieve data as it is downloaded. To allow time for the URL Access Manager to refill its buffers, call `URLIdle` in between calls to `URLGetBuffer`.

Creating and Managing Universal Procedure Pointers

You can use these functions to create and manage universal procedure pointers (UPPs) to your notification and system event callback functions.

`NewURLNotifyUPP` — Creates a UPP to your notification callback function. (page 55)

`NewURLSystemEventUPP` — Creates a UPP to your system event callback function. (page 56)

`InvokeURLNotifyUPP` — Invokes your notification callback function. (page 56)

`InvokeURLSystemEventUPP` — Invokes your system event callback function. (page 57)

`DisposeURLNotifyUPP` — Disposes of a UPP to your notification callback function. (page 58)

`DisposeURLSystemEventUPP` — Disposes of a UPP to your system event callback function. (page 58)

NewURLNotifyUPP

Creates a UPP to your notification callback function.

```
URLNotifyUPP NewURLNotifyUPP (URLNotifyProcPtr userRoutine);
```

`userRoutine`
A pointer to your notification callback function. For information on how to create a notification callback function, see `URLNotifyProcPtr` (page 59).

function result A UPP to your notification callback function. You can register your notification callback function by passing this UPP in the `notifyProc` parameter of the function `URLOpen` (page 47).

Discussion

The `NewURLNotifyUPP` function creates a pointer to your notification callback function. You pass a pointer to your callback function in the `notifyProc` parameter of the function `URLOpen` (page 47) if you want your application to receive data transfer events. Pass a bitmask in the `eventRegister` parameter of `URLOpen` indicating which data transfer events you want to receive.

Special Considerations

When you are finished with a UPP to your notification callback function, you should dispose of it by calling the function `DisposeURLNotifyUPP` (page 58).

NewURLSystemEventUPP

Creates a UPP to your system event callback function.

```
URLSystemEventUPP NewURLSystemEventUPP (
                    URLSystemEventProcPtr userRoutine);
```

`userRoutine`

A pointer to your system event callback function. For information on how to create a system event callback function, see `URLSystemEventProcPtr` (page 60).

function result A UPP to your system event callback function. You can register your system event callback function by passing this UPP in the `eventProc` parameter of the functions `URLSimpleDownload` (page 38), `URLSimpleUpload` (page 43), `URLDownload` (page 40), and `URLUpload` (page 45).

Discussion

The `NewURLSystemEventUPP` function creates a pointer to your system event callback function. You pass a pointer to your callback function in the `notifyProc` parameter of the functions `URLSimpleDownload` (page 38), `URLSimpleUpload` (page 43), `URLDownload` (page 40), and `URLUpload` (page 45) if you want to handle events that occur while a progress indicator or authentication dialog box is being displayed.

Special Considerations

When you are finished with a UPP to your notification callback, you should dispose of it by calling the function `DisposeURLNotifyUPP` (page 58).

InvokeURLNotifyUPP

Invokes your notification callback function.

```
OSStatus InvokeURLNotifyUPP (
                    void *userContext,
                    URLEvent event,
                    URLCallbackInfo *callbackInfo,
                    URLNotifyUPP userUPP);
```

`userContext`

A pointer to application-defined storage. The URL Access Manager passes this value in the `userContext` parameter of your notification callback function.

`event`

The data transfer events you want your application to receive. See "Data Transfer Event Constants" (page 65) for a description of possible values. The URL Access Manager tests the bitmask you pass in the `eventRegister` parameter of the function `URLOpen` (page 47) to determine which events to pass to your callback function. See "Data Transfer Event Mask Constants" (page 67) for a description of this bitmask.

`callbackInfo`

A pointer to a structure of type `URLCallbackInfo` (page 62) that provides information about the data transfer event to your callback function. The URL Access Manager passes a pointer to this structure in the `callbackInfo` parameter of your notification callback function.

`userUPP`

A Universal Procedure Pointer to your notification callback function. For information on how to create a notification callback function, see `URLNotifyProcPtr` (page 59).

Discussion

The URL Access Manager calls the `InvokeURLNotifyUPP` function when you pass a UPP to your callback function in the notifyProc parameter of the function `URLOpen` (page 47), and the data transfer event that you specified in the `eventRegister` parameter occurs.

InvokeURLSystemEventUPP

Invokes your system event callback function.

```
OSStatus InvokeURLSystemEventUPP (
                    void *userContext,
                    EventRecord *event,
                    URLSystemEventUPP userUPP);
```

`userContext`

A pointer to application-defined storage. The URL Access Manager passes this value in the `userContext` parameter of your system event callback function.

event

A pointer to an event record that provides information about the system event to your callback function.

userUPP

A Universal Procedure Pointer to your system event callback function. For information on how to create a system event callback function, see URLSystemEventProcPtr (page 60).

Discussion

The URL Access Manager calls the InvokeURLSystemEventUPP function when you pass a UPP to your callback function in the eventProc parameter of the functions URLSimpleDownload (page 38), URLSimpleUpload (page 43), URLDownload (page 40), or URLUpload (page 45), and a system event occurs while a progress indicator or authentication dialog box is being displayed.

DisposeURLNotifyUPP

Disposes of a UPP to your notification callback function.

```
void DisposeURLNotifyUPP (
                    URLNotifyUPP userUPP);
```

userUPP

A UPP to your notification callback function.

Discussion

When you are finished with a UPP to your notification callback function, you should dispose of it by calling the DisposeURLNotifyUPP function.

DisposeURLSystemEventUPP

Disposes of a UPP to your system event callback function.

```
void DisposeURLSystemEventUPP (
                    URLSystemEventUPP userUPP);
```

userUPP

A UPP to your system event callback function.

Discussion

When you are finished with a UPP to your system event callback function, you should dispose of it by calling the DisposeURLSystemEventUPP function.

URL Access Callbacks

`URLNotifyProcPtr` — Defines a pointer to your notification callback function. Your notification callback function handles certain data transfer events that occur during data transfer operations. (page 59)

`URLSystemEventProcPtr` — Defines a pointer to your system event callback function. Your system event callback function handles system events that occur during a data tranfer operation. (page 60)

URLNotifyProcPtr

Defines a pointer to your notification callback function. Your notification callback function handles certain data transfer events that occur during data transfer operations.

```
OSStatus *URLNotifyProcPtr (
                    void *userContext,
                    URLEvent event,
                    URLCallbackInfo *callbackInfo);
```

You would declare your notification callback function like this if you were to name it `MyURLNotifyCallback`:

```
OSStatus MyURLNotifyCallback(
                    void *userContext,
                    URLEvent event,
                    URLCallbackInfo *callbackInfo);
```

`userContext`

A pointer to application-defined storage that your application previously passed to the function `URLOpen` (page 47). Your application can use this to associate any particular call of `URLOpen` with any particular call of the notification callback function.

`event`

The data transfer event that your application wishes to be notified of. See “Data Transfer Event Constants” (page 65) for a description of possible values. The type of event that can trigger your callback

depends on the event mask you passed in the `eventRegister` parameter of the function `URLOpen` (page 47), and whether you pass a valid file specification in the `fileSpec` parameter of `URLOpen`. For more information, see the discussion.

`callbackInfo`

A pointer to a structure of type `URLCallbackInfo` (page 62). On return, the structure contains information about the data transfer event that occurred. The URL Access Manager passes this information to your callback function via the `callbackInfo` parameter of the function `InvokeURLNotifyUPP` (page 56).

function result Your notification callback function should process the system event and return `noErr`.

Discussion

Your notification callback function handles certain data transfer events that occur during data transfer operations performed by the function `URLOpen` (page 47). You can define an event notification function and the events for which you want to receive notification only if you do not specify a file in which to store the data for download operations. In order to be notified of these events, you must pass a UPP to your notification callback function in the `notifyProc` parameter. You indicate the type of data transfer events you want to receive via a bitmask in the `eventRegister` parameter.

Special Considerations

Do not call the function `URLDisposeReference` (page 33) from your notification callback function. Doing so may cause your application to stop working. Your application may call any function. For example, your notification callback can update its user interface, allocate and deallocate memory, or call the Thread Manager function `NewThread`.

URLSystemEventProcPtr

Defines a pointer to your system event callback function. Your system event callback function handles system events that occur during a data tranfer operation.

```
void *URLSystemEventProcPtr (
                    void *userContext.
                    EventRecord *event);
```

You would declare your system event callback function like this if you were to name it `MyURLSystemEventProc`:

```
void MyURLSystemEventProc (
                    void *userContext.
                    EventRecord *event);
```

`userContext`

A pointer to application-defined storage that your application previously passed to the function `URLSimpleDownload` (page 38), `URLDownload` (page 40), `URLSimpleUpload` (page 43), or `URLUpload` (page 45). Your application can use this to associate any particular call of these functions with any particular call of the system event callback function.

`event`

A pointer to an event record containing information about the system event that occurred during the data transfer operation.

function result Your system event callback function should process the system event and return `noErr`.

Discussion

Your system event callback function handles system events that may occur during a call to the functions `URLSimpleDownload` (page 38), `URLDownload` (page 40), `URLSimpleUpload` (page 43), and `URLUpload` (page 45) if you indicate that a progress indicator or authentication dialog box should be displayed by these functions and the user interacts with it. If you wish these functions to display a movable progress indicator or authentication dialog box during the data transfer operation, you should specify the appropriate masks, `kURLDisplayProgressFlag` or `kURLDisplayAuthFlag`, in the `openFlags` parameter and a UPP to your callback function in the `eventProc` parameter of these functions. If you pass `NULL` in the `eventProc` parameter, the URL Access Manager displays a nonmovable modal progress indicator and authentication dialog box.

When your system event callback function is called, it should process the event immediately.

Special Considerations

Do not call the function `URLDisposeReference` (page 33) from your system event callback function. Doing so may cause your application to stop working. Your application may call any function. For example, your system event callback can update its user interface, allocate and deallocate memory, or call the Thread Manager function `NewThread`.

URL Access Data Types

`URLReference` — Represents a reference to a URL. (page 62)

`URLCallbackInfo` — Contains information about a data transfer event. (page 62)

URLReference

Represents a reference to a URL.

```
typedef struct OpaqueURLReference* URLReference;
```

Discussion

The `URLReference` type represents a reference to an opaque structure that identifies a URL. You should call the function `URLNewReference` (page 33) to create a URL reference. The function `URLDisposeReference` (page 33) disposes of a URL reference when no longer needed. You pass a reference of this type to URL Access Manager functions that operate on a URL in some way.

URLCallbackInfo

Contains information about a data transfer event.

```
struct URLCallbackInfo{
   UInt32 version;
   URLReference urlRef;
   const char *property;
   UInt32 currentSize;
   EventRecord *systemEvent;
   };
   typedef struct URLCallbackInfo URLCallbackInfo;
```

Field descriptions

`version`

The version of this structure.

`urlRef`

A reference to the URL associated with the data transfer event.

`property`

A pointer to a C string representing the name of the URL property that has changed, if relevant. This field is only valid if a property change event occurs as identified by the event constant `kURLPropertyChangedEvent`, described in "Data Transfer Event Constants" (page 65). For a description of name constants and data types of the corresponding property values, see "Universal URL Property Name Constants" (page 78) and "HTTP and HTTPS URL Property Name Constants" (page 76).

`currentSize`

The size (in bytes) of the property value identified by the name constant in the property parameter, if relevant. This field is only valid if a property change event occurs as identified by the constant `kURLPropertyChangedEvent`.

`systemEvent`

A pointer to an event record containing information about the system event that occurred, if relevant. If the event is not a system event, as identified by the event constant `kURLSystemEvent`, described in "Data Transfer Event Constants" (page 65), this field is not valid.

Discussion

The `URLCallbackInfo` type represents a structure that contains information about the data transfer event that you want to notification of. The URL Access Manager passes a pointer to this structure in the `callbackInfo` parameter of your notification callback function. For information on how to write a notification callback function, see `URLNotifyProcPtr` (page 59).

URL Access Constants

`Authentication Flag Constant` — Represents the default value of the property value identified by the property name constant kURLAuthType. (page 64)

`Data Transfer Event Constants` — Identify data transfer events that occur during a data transfer operation. (page 65)

`Data Transfer Event Mask Constants` — Represent a mask that identifies the data transfer events occurring during a data transfer operation that your application wants notification of. (page 67)

`Data Transfer Options Mask Constants` — Represent a mask that identifies the data transfer options to use when uploading or downloading data. (page 71)

`Data Transfer State Constants` — Identifies the current state of a data transfer operation. (page 74)

`HTTP and HTTPS URL Property Name Constants` — Identify property values specific to HTTP and HTTPS URLs. (page 76)

`Universal URL Property Name Constants` — Identify property values universal to all URLs. (page 78)

Authentication Type Constant

Represents the default value of the property value identified by the property name constant `kURLAuthType`.

```
enum{
        kUserNameAndPasswordFlag      = 0x00000001
    };
```

Constant descriptions

`kUserNameAndPasswordFlag`

Represents the default value of the property value identified by the property name constant kURLAuthType, described in "Universal URL Property Name Constants" (page 78). This value indicates that both the user name and password are used for authentication.

Discussion

This constant represents the default value of the authentication type property value. The authentication type property value is identified by the property name constant `kURLAuthType`, described in "Universal URL Property Name Constants" (page 78). If you do not set the `kURLAuthType` property, the default value will be used for the authentication type. In this case, both the user name and password are used for authentication purposes.

Data Transfer Event Constants

Identify data transfer events that occur during a data transfer operation.

```
enum{
   kURLInitiatedEvent               = kURLInitiatingState,
   kURLResourceFoundEvent           = kURLResourceFoundState,
   kURLDownloadingEvent             = kURLDownloadingState,
   kURLAbortInitiatedEvent          = kURLAbortingState,
   kURLCompletedEvent               = kURLCompletedState,
   kURLErrorOccurredEvent           = kURLErrorOccurredState,
   kURLDataAvailableEvent           = kURLDataAvailableState,
   kURLTransactionCompleteEvent     = kURLTransactionCompleteState,
   kURLUploadingEvent               = kURLUploadingState,
   kURLSystemEvent                  = 29,
   kURLPercentEvent                 = 30,
   kURLPeriodicEvent                = 31,
   kURLPropertyChangedEvent         = 32
   };
   typedef UInt32 URLEvent;
```

Constant descriptions

`kURLInitiatedEvent`

Indicates the function `URLOpen` (page 47) has been called but the location specified by the URL reference has not yet been accessed.

`kURLResourceFoundEvent`

Indicates that the location specified by the URL reference has been accessed and is valid.

`kURLDownloadingEvent`

Indicates that a download operation is in progress.

`kURLAbortInitiatedEvent`

Indicates that a data transfer operation has been aborted. When your application calls the function `URLAbort` (page 49), the URL Access Manager changes the state returned by `the function URLGetCurrentState` (page 53) to `kURLAbortingState` and passes the constant `kURLAbortInitiatedEvent` to your notification callback function. When data transfer is terminated, the URL Access Manager changes the state returned by `URLGetCurrentState` to `kURLCompletedState` and passes the constant `kURLCompletedEvent` in the `event` parameter of your notification callback function.

`kURLCompletedEvent`

Indicates that all operations associated with a call to `URLOpen` have been completed. This includes the successful completion of a download or upload operation or the completion of cleanup work after aborting a download or upload operation. For example, when a data transfer operation is aborted, the URL Access Manager changes the state returned by the function `URLGetCurrentState` to `kURLCompletedState` and passes the constant `kURLCompletedEvent` in the `event` parameter of your notification callback function.

`kURLErrorOccurredEvent`

Indicates that an error occurred during data transfer. If you receive this event, you may wish to call the function `URLGetError` (page 52) to determine the nature of the error.

`kURLDataAvailableEvent`

Indicates that data is available in buffers. If you receive this event, you can call the function `URLGetBuffer` (page 51) to obtain the next buffer of data. You may wish to call the function `URLGetDataAvailable` (page 50) to determine the amount of data available for retrieval in a download operation. Note that if you pass a valid file specification in the `fileSpec` parameter of `URLOpen`, your notification callback function will not be called for data available events.

`kURLTransactionCompleteEvent`

Indicates that a download operation is complete because there is no more data to retrieve from buffers. Note that if you pass a valid file specification in the `fileSpec` parameter of `URLOpen`, your notification callback function will not be called for transaction completed events.

`kURLUploadingEvent`

Indicates that an upload operation is in progress.

`kURLSystemEvent`

Indicates that a system event has occurred.

`kURLPercentEvent`

Indicates that the size of the data being downloaded is known. In this case, an increment of one percent of the data was transferred into buffers.

`kURLPeriodicEvent`

Indicates that a time interval of approximately one quarter of a second has passed.

`kURLPropertyChangedEvent`

Indicates that a property such as a filename has become known or changed. In this case, the name of the changed property will be passed to your notification function via the `property` field of the `callbackInfo` structure.

Discussion

The `URLEvent` enumeration defines constants that identify data transfer events that occur during a data transfer operation performed by `URLOpen`. You can define an event notification function and the events for which you want to receive notification only if you do not specify a file in which to store the data for downloads. In order to be notified of these events, you must pass a UPP to your notification callback function in the `notifyProc` parameter. You indicate the type of data transfer events you want to receive via a bitmask in the `eventRegister` parameter. For a description of this bitmask, see "Data Transfer Event Mask Constants" (page 67).

Data Transfer Event Mask Constants

Represent a mask that identifies the data transfer events occurring during a data transfer operation that your application wants notification of.

```
enum{
   kURLInitiatedEventMask           = 1 << (kURLInitiatedEvent - 1),
   kURLResourceFoundEventMask       = 1 << (kURLResourceFoundEvent - 1),
   kURLDownloadingMask              = 1 << (kURLDownloadingEvent - 1),
   kURLUploadingMask                = 1 << (kURLUploadingEvent - l),
   kURLAbortInitiatedMask           = 1 << (kURLAbortInitiatedEvent - 1),
   kURLCompletedEventMask           = 1 << (kURLCompletedEvent - 1),
   kURLErrorOccurredEventMask       = 1 << (kURLErrorOccurredEvent - 1),
   kURLDataAvailableEventMask       = 1 << (kURLDataAvailableEvent - 1),
   kURLTransactionCompleteEventMask = 1 << (kURLTransactionCompleteEvent-1),
   kURLSystemEventMask              = 1 << (kURLSystemEvent - 1 ),
   kURLPercentEventMask             = 1 << (kURLPercentEventMask -1 ),
   kURLPeriodicEventMask            = 1 << (kURLPeriodicEvent -1 ),
   kURLPropertyChangedEventMask     = 1 << (kURLPropertyChangedEvent - 1 ),
   kURLAllBufferEventsMask          = kURLDataAvailableEventMask +
                                        kURLTransactionCompleteMask,
   kURLAllNonBufferEventsMask       = kURLinitiatedEventMask+
                                        kURLDownloadingMask+
                                        kURLUploadingMask+
                                        kURLAbortInitiatedMask+
                                        kURLCompletedEventMask+
                                        kURLErrorOccurredEventMask+
                                        kURLPercentEventMask+
```

```
                                        kURLPeriodicEventMask+
                                        kURLPropertyChangedEventMask,
    kURLAllEventsMask                = (long)0xFFFFFFFF
    };
    typedef UInt32 URLEventMask;
```

Constant descriptions

`kURLInitiatedEventMask`

If the bit specified by this mask is set, your notification callback function will be notified when the function `URLOpen` (page 47) has been called but the location specified by the URL reference has not yet been accessed.

`kURLResourceFoundEventMask`

If the bit specified by this mask is set, your notification callback function will be notified when the location specified by a URL reference has been accessed and is valid.

`kURLDownloadingMask`

If the bit specified by this mask is set, your notification callback function will be notified when a download operation is in progress.

`kURLUploadingMask`

If the bit specified by this mask is set, your notification callback function will be notified when an upload operation is in progress.

`kURLAbortInitiatedMask`

If the bit specified by this mask is set, your notification callback function will be notified when a download or upload operation has been aborted. When your application calls the function `URLAbort` (page 49), the URL Access Manager changes the state returned by `the function URLGetCurrentState` (page 53) to `kURLAbortingState` and passes the constant `kURLAbortInitiatedEvent` to your notification callback function. When data transfer is terminated, the URL Access Manager changes the state returned by `URLGetCurrentState` to `kURLCompletedState` and passes the constant `kURLCompletedEvent` in the `event` parameter of your notification callback function.

`kURLCompletedEventMask`

If the bit specified by this mask is set, your notification callback function will be notified when all operations associated with a call to the function `URLOpen` (page 47) have been completed. This indicates either the successful completion of an operation or the completion of cleanup work after aborting the operation. For example, when a data transfer operation is aborted, the URL Access Manager changes the

state returned by the function `URLGetCurrentState` (page 53) to `kURLCompletedState` and passes the constant `kURLCompletedEvent` in the `event` parameter of your notification callback function.

`kURLErrorOccurredEventMask`

If the bit specified by this mask is set, your notification callback function will be notified when an error has occurred. If you receive this event, you may wish to call the function `URLGetError` (page 52) to determine the nature of the error.

`kURLDataAvailableEventMask`

If the bit specified by this mask is set, your notification callback function will be notified when data is available in buffers. If you receive this event, you may wish to call the function `URLGetDataAvailable` (page 50) to determine the amount of data available for retrieval in a download operation. Note that if you pass a valid file specification in the `fileSpec` parameter of the function `URLOpen` (page 47), your notification callback function will not be called for data available events.

`kURLTransactionCompleteEventMask`

If the bit specified by this mask is set, your notification callback function will be notified when the operation is complete because there is no more data to retrieve from buffers. Note that if you pass a valid file specification in the `fileSpec` parameter of the function `URLOpen` (page 47), your notification callback function will not be called for transaction completed events.

`kURLPercentEventMask`

If the bit specified by this mask is set, your notification callback function will be notified when an increment of one percent of the data has been transferred into buffers. This occurs only when the size of the data being transferred is known. This information is useful if you want the URL Access Manager to display a progress indicator.

`kURLPeriodicEventMask`

If the bit specified by this mask is set, your notification callback function will be notified when a time interval of approximately one quarter of a second has passed. You can use this event to report the progress of the download operation when the size of the data is unknown or for other processing that you wish to perform at regular intervals.

`kURLPropertyChangedEventMask`

If the bit specified by this mask is set, your notification callback function will be notified when the value of a URL property, such as a filename or user name, has become known or changes.

`kURLAllBufferEventsMask`

If the bit specified by this mask is set, your notification callback function will be notified when a buffer-related event indicated by the event constants `kURLDataAvailableEvent` or `kURLTransactionCompleteEvent` occurred. If you pass a file specification in the `fileSpec` parameter of the function `URLOpen` (page 47), your notification callback function will not be called for buffer-related events.

`kURLAllNonBufferEventsMask`

If the bit specified by this mask is set, your notification callback function will be notified when an event unrelated to a buffer occurred. This includes all events except those represented by the constants `kURLDataAvailableEvent` and `kURLTransactionCompleteEvent`.

`kURLAllEventsMask`

If the bit specified by this mask is set, your notification callback function will be notified when any of the above data transfer events occur. If you pass a file specification in the `fileSpec` parameter of the function `URLOpen` (page 47), your notification callback function will not be called for buffer-related events.

Discussion

The `URLEventMask` enumeration defines masks that identify the data transfer events occurring during a call to the function `URLOpen` (page 47) that your application wants notification of. For a description of data transfer events, see “Data Transfer Event Constants” (page 65). You can define an event notification function and the events for which you want to receive notification only if you do not specify a file in which to store the data for downloads. You can indicate which events you want to receive notification of via a bitmask in the `eventRegister` parameter of `URLOpen`.

Data Transfer Options Mask Constants

Represent a mask that identifies the data transfer options to use when uploading or downloading data.

```
enum{
   kURLReplaceExistingFlag      = 1 << 0,
   kURLRBinHexFileFlag          = 1 << 1, /* Binhex before uploading if
                                   necessary*/
   kURLExpandFileFlag           = 1 << 2, /* Use StuffIt engine to expand
                                   file if necessary*/
   kURLDisplayProgressFlag      = 1 << 3,
   kURLDisplayAuthFlag          = 1 << 4, /* Display auth dialog if guest
                                   connection fails*/
   kURLUploadFlag               = 1 << 5, /* Do an upload instead of a
                                   download*/
   kURLIsDirectoryHintFlag      = 1 << 6, /* Hint: the URL is a directory*/
   kURLDoNotTryAnonymousFlag    = 1 << 7, /* Don't try to connect anonymously
                                   before getting logon info*/
   kURLDirectoryListingFlag     = 1 << 8, /* Download the directory listing,
                                   not the whole directory*/
   kURLExpandAndVerifyFlag      = 1 << 9, /* Expand file and then verify using
                                   signature resource*/
   kURLNoAutoRedirectFlag       = 1 << 10, /* Do not automatically redirect
                                   to new URL*/
   kURLDebinhexOnlyFlag         = 1 << 11, /* Do not use Stuffit Expander -
                                   just internal debinhex engine*/
   kURLReservedFlag             = 1 << 31 /* reserved for Apple internal
                                   use*/

};
   typedef UInt32 URLOpenFlags;
```

Constant descriptions

`kURLReplaceExistingFileFlag`

If the bit specified by this mask is set and the file or directory exists, the file or directory contents are replaced by the newly downloaded or uploaded data. If the name of the file or directory is not specified, the file or directory already exists, and the bit specified by this mask is not set, a number is appended to the name before any extension until a unique name is created, and the data is transferred to the new file or directory name without notifying the calling application that the name has changed. In the case of a download operation, your application can check the `destination` parameter of the functions `URLSimpleDownload` (page 38) and `URLDownload` (page 40) to obtain the new filename.

`kURLBinHexFileFlag`

If the bit specified by this mask is set, the URL Access Manager converts a nontext file that has a resource fork to `BinHex` format before uploading it.

`kURLExpandFileFlag`

If the bit specified by this mask is set, files in `BinHex` format are decoded. If version 5.0 of the Stuffit Engine is installed in the System Folder, the URL Access Manager uses it to expand the file.

`kURLDisplayProgressFlag`

If the bit specified by this mask is set, the URL Access Manager displays a nonmovable modal progress indicator during data transfer operations. If you want to handle events that occur while a progress indicator is being displayed, pass a UPP to your system event callback in the `eventProc` parameter of the functions `URLSimpleDownload` (page 38), `URLDownload` (page 40), `URLSimpleUpload` (page 43), and `URLUpload` (page 45).

`kURLDisplayAuthFlag`

If the bit specified by this mask is set, the URL Access Manager displays a nonmovable modal authentication dialog box when user authentication is required. If you want to handle events that occur while an authentication dialog box is being displayed, pass a UPP to your system event callback in the `eventProc` parameter of the functions `URLSimpleDownload` (page 38), `URLDownload` (page 40), `URLSimpleUpload` (page 43), and `URLUpload` (page 45).

`kURLUploadFlag`

If the bit specified by this mask is set, the functions `URLUpload` (page 45) and `URLOpen` (page 47) upload the file or directory to the specified URL.

`kURLIsDirectoryHintFlag`

If the bit specified by this mask is set, download operations assume that the URL points to a directory. Note that if you pass a pathname that specifies a file in the `url` parameter of the function `URLSimpleDownload` (page 38), the file is downloaded regardless of whether you specify `kURLDirectoryListingFlag` or `KURLIsDirectoryHintFlag` in the `openFlags` parameter.

`kURLDoNotTryAnonymousFlag`

If the bits specified by this mask and the mask `kURLDisplayAuth` are set, the functions `URLSimpleDownload` (page 38), `URLDownload` (page 40), `URLSimpleUpload` (page 43), and `URLUpload` (page 45)

display an authentication dialog box when attempting to log on to an FTP server. If this bit is not set, these functions will attempt to log on anonymously.

`kURLDirectoryListingFlag`

If the bit specified by this mask is set, a listing of the directory, rather than the entire directory, is downloaded. If the URL points to a file instead of a directory, the file is downloaded. Note that if you pass a pathname that specifies a file in the `url` parameter of the function `URLSimpleDownload` (page 38), the file is downloaded regardless of whether you specify `kURLDirectoryListingFlag` or `KURLIsDirectoryHintFlag` in the `openFlags` parameter.

`kURLExpandAndVerifyFlag`

If this flag is available (that is, the File Signing shared library is available) and the bit specified by this mask is set, the signature attached to the file is verified. Success indicates that the file was signed by the certificate authority, but the certificate will not be displayed until after the file is downloaded.

`kURLNoAutoRedirectFlag`

If the bit specified by this mask is set, if an HTTP request returns a “redirect”status (300, 301, or 302), the URL Access Manager will return without attempting to redirect to the next URL.

`kURLDebinhexOnlyFlag`

If the bit specified by this mask is set, the internal engine is used to decode files, rather than the external Stuffit Engine, even if Stuffit is installed. This prevents the display of the Stuffit progress user interface.

`kURLReservedFlag`

Reserved for internal use.

Discussion

The `URLOpenFlags` enumeration defines masks you can use to identify the data transfer options you want used when performing data transfer operations. You pass this mask in the `openFlags` parameter of the functions `URLSimpleDownload` (page 38), `URLDownload` (page 40), `URLSimpleUpload` (page 43), `URLUpload` (page 45), and `URLOpen` (page 47).

For upload operations, you can specify any of the following masks: `kURLReplaceExistingFlag`, `kURLBinHexFileFlag`, `kURLDisplayProgressFlag`, `kURLDisplayAuthFlag`, and `kURLDoNotTryAnonymousFlag`. For download operations, you can specify any of the following masks: `kURLReplaceExistingFlag`,

kURLExpandFileFlag, kURLExpandAndVerifyFlag, kURLDisplayProgressFlag, kURLDisplayAuthFlag, kURLIsDirectoryHintFlag, kURLDoNotTryAnonymousFlag, and kURLDirectoryListingFlag.

Data Transfer State Constants

Identifies the current state of a data transfer operation.

```
enum{
   kURLNullState               = 0,
   kURLInitiatedState          = 1,
   kURLLookingUpHostState      = 2,
   kURLConnectingState         = 3,
   kURLResourceFoundState      = 4,
   kURLDownloadingState        = 5,
   kURLDataAvailableState      = 0x10 + kURLDownloadingState,
   kURLTransactionCompleteState = 6,
   kURLErrorOccurredState      = 7,
   kURLAbortingState           = 8,
   kURLCompletedState          = 9,
   kURLUploadingState          = 10
   };
   typedef UInt32 URLState;
```

Constant descriptions

kURLNullState

Indicates that the function URLOpen (page 47) has not yet been called.

kURLInitiatingState

Indicates that the function URLOpen (page 47) has been called, but the location specified by the URL reference has not yet been accessed. The stream enters this state from the kURLNullState state.

kURLLookingUpHostState

Indicates that the function URLOpen (page 47) has been called, and that the host is being looked up. The stream enters this state from the kURLInitiatingState state.

kURLConnectingState

Indicates that the function URLOpen (page 47) has been called, and a connection is being made. The stream enters this state from the kURLInitiatingState state.

`kURLResourceFoundState`

Indicates that the location specified by the URL reference has been accessed and is valid. The stream enters this state from the `kURLInitiatingState` state.

`kURLDownloadingState`

Indicates that the download operation is in progress but there is currently no data in the buffers. The stream enters this state initially from the `kURLResourceFoundState` state. During a download operation, the stream's state may alternate between the `kURLDownloadingState` and the `kURLDataAvailableState` states.

`kURLDataAvailableState`

Indicates that the download operation is in progress and data is available in the buffers. The stream initially enters this state from the `kURLDownloadingState` state. During a download operation, the stream's state may alternate between the `kURLDownloadingState` and the `kURLDataAvailableState` states. If the stream is in the data available state, you may want to call the function `URLGetDataAvailable` (page 50) to determine the amount of data available for download. If you pass `NULL` in the `fileSpec` parameter of the function `URLOpen` (page 47), you will need to call the function `URLGetBuffer` (page 51) to obtain the next buffer of data.

`kURLTransactionCompleteState`

Indicates that a download or upload operation is complete. The stream can enter this state from the `kURLDownloadingState` state.

`kURLErrorOccurredState`

Indicates that an error occurred during data transfer. The stream can enter this state from any state except the `kURLAbortingState` state. If the stream is in this state, you may wish to call the function `URLGetError` (page 52) to determine the nature of the error.

`kURLAbortingState`

Indicates that a download or upload operation is aborting. The stream enters this state from the `kURLErrorOccurredState` state or as a result of calling the function `URLOpen` (page 47) when the stream is in any other state. When your application calls the function `URLAbort` (page 49), the URL Access Manager changes the state returned by `the function URLGetCurrentState` (page 53) to `kURLAbortingState` and passes the constant `kURLAbortInitiatedEvent` to your notification callback function.

`kURLCompletedState`

Indicates that there is no more activity to be performed on this stream. In this case, the data transfer has either completed successfully, been aborted, or an error has occurred. The stream enters this state from the `kURLTransactionCompleteState` or the `kURLAbortingState` state. When data transfer is terminated after a data transfer operation is aborted, the URL Access Manager changes the state returned by `URLGetCurrentState` to `kURLCompletedState` and passes the constant `kURLCompletedEvent` in the `event` parameter of your notification callback function.

`kURLUploadingState`

Indicates that an upload operation is in progress.

Discussion

The `URLState` enumeration defines constants that identify the status of a data transfer operation with respect to a URL. The function `URLGetCurrentState` (page 53) passes back one of these constants in the `state` parameter to indicate the status of a data transfer operation. All constants except `kURLDataAvailableState` and `kURLCompletedState` can be returned at any time. If you pass a valid file specification in the `fileSpec` parameter of the function `URLOpen` (page 47), your notification callback function will not be notified of data available and transaction completed states as identified by the constants `kURLDataAvailableState` and `kURLTransactionCompleteState`.

HTTP and HTTPS URL Property Name Constants

Identify property values specific to HTTP and HTTPS URLs.

```
#define kURLHTTPRequestMethod "URLHTTPRequestMethod";
#define kURLHTTPRequestHeader "URLHTTPRequestHeader";
#define kURLHTTPRequestBody "URLHTTPRequestBody";
#define kURLHTTPRespHeader "URLHTTPRespHeader";
#define kURLHTTPUserAgent "URLHTTPUserAgent";
#define kURLHTTPRedirectedURL "URLHTTPRedirectedURL"
```

Constant descriptions

`kURLHTTPRequestMethod`

Identifies the HTTP request method property value. You use this name constant to set or obtain a C string that represents the HTTP method to be used in the request. If you are posting a form, you must set this property.

`kURLHTTPRequestHeader`

Identifies the HTTP request header property value. You use this name constant to set or obtain a C string that represents the HTTP header to be used in the request. You may set this property to contain all headers needed for the request. If you are posting a form and have set the properties identified by the name constants `kURLHTTPRequestMethod` and `kURLHTTPRequestBody`, you do not need to set the property identified by this tag.

`kURLHTTPRequestBody`

Identifies the HTTP request body property value. You use this name constant to set or obtain a buffer of data that represents the HTTP body to be provided in the request. If you set the property identified by this tag but not that identified by the name constant `kURLHTTPHeader`, a body-length header is automatically added to the request. If you are posting a form, you must set this property.

`kURLHTTPRespHeader`

Identifies the HTTP response header property value. You use this name constant to obtain a C string that represents the HTTP response header to be provided in the request.

`kURLHTTPUserAgent`

Identifies the user agent property value. You use this name constant to set or obtain a C string that represents the HTTP user agent string that is embedded in HTTP requests. By default, the URL Access Manager sets the user agent string to “URL Access 1.0 (Macintosh; PPC)”.

`kURLHTTPRedirectedURL`

Identifies the redirected URL property value.

Discussion

These constants represent Apple-defined name constants that identify property values specific to HTTP and HTTPS URLs. For a description of the name constants that identify property values universal to all URLs, see “Universal URL Property Name Constants” (page 78).

You pass one of these name constants in the `property` parameter of the functions `URLSetProperty` (page 36) and `URLGetProperty` (page 35), respectively, to set or obtain a particular property value. Note that you can only set property values identified by the name constants `kURLFileType`, `kURLFileCreator`, `kURLUserName`, and `kURLPassword`. You must also pass the correct data type corresponding to the property value in the `propertyBuffer` parameter of these functions.

Version Notes

Prior to version 2.0.3 of the URL Access Manager, the data type of the property value identified by the name constant `kURLHTTPRequestBody` was a C string. In 2.0.3 and later, the data type is a buffer of data.

Universal URL Property Name Constants

Identify property values universal to all URLs.

```
#define kURLURL "URLString";
#define kURLResourceSize "URLResourceSize";
#define kURLLastModifiedTime "URLLastModifiedTime";
#define kURLMIMEType "URLMIMEType";
#define kURLFileType "URLFileType";
#define kURLFileCreator "URLFileCreator";
#define kURLCharacterSet "URLCharacterSet";
#define kURLResourceName "URLResourceName";
#define kURLHost "URLHost";
#define kURLAuthType "URLAuthType";
#define kURLUserName "URLUserName";
#define kURLPassword "URLPassword";
#define kURLStatusString "URLStatusString";
#define kURLIsSecure "URLIsSecure";
#define kURLCertificate "URLCertificate";
#define kURLTotalItems "URLTotalItems";
```

Constant descriptions

`kURLURL`

Identifies the name string property value. You use this name constant to obtain a C string that represents the pathname of the URL.

`kURLResourceSize`

Identifies the resource size property value. You use this name constant to obtain a value of type `Size` that represents the total size of the data at the location specified by the URL.

`kURLLastModifiedTime`

Identifies the modification time property value. You use this name constant to obtain a value of type `UInt32` that represents the last time the data was modified.

`kURLMIMEType`

Identifies the `MIME` type property value. You use this name constant to obtain a Pascal string that represents the `MIME` type of the URL.

`kURLFileType`

Identifies the file type property value. You use this name constant to set or obtain a value of type `OSType` that represents the file type as specified in a call to the function `URLOpen` (page 47). If the file type was not specified, `kURLFileType` obtains the file type compatible with the `MIME` type.

`kURLFileCreator`

Identifies the file creator property value. You use this name constant to set or obtain a value of type `OSType` that represents the file creator as specified in a call to the function `URLOpen` (page 47). If the file creator was not specified, `kURLFileType` obtains the file type compatible with the `MIME` type.

`kURLCharacterSet`

Identifies the character set property value. You use this name constant to obtain a Pascal string that represents the character set used by the URL, as returned by the HTTP server.

`kURLResourceName`

Identifies the resource name property value. You use this name constant to obtain a Pascal string that represents the name associated with the data to be downloaded.

`kURLHost`

Identifies the host property value. You use this name constant to obtain a Pascal string that represents the host on which the data is located.

`kURLAuthType`

Identifies the authentication type property value. You use this name constant to obtain a value that represents the type of authentication that the download operation requires. The default authentication type is `kUserNameAndPasswordFlag`, described in “Authentication Type Constant” (page 64).

`kURLUserName`

Identifies the user name property value. You use this name constant to set or obtain a Pascal string that represents the user name used for authentication.

`kURLPassword`

Identifies the password property value. You use this name constant to set or obtain a Pascal string that represents the password used for authentication.

`kURLStatusString`

Identifies the status property value. You use this name constant to obtain a Pascal string that represents the current status of the data stream. You can use this property to display the status of the data transfer operation.

`kURLIsSecure`

Identifies the security property value. You use this name constant to get a `Boolean` value that indicates whether the download operation is secure.

`kURLCertificate`

Identifies the certificate property value. You use this name constant to obtain a buffer of data that represents the certificate provided by a remote server.

`kURLTotalItems`

Identifies the total items property value. You use this name constant to obtain a value of type `UInt32` that represents the total number of items being uploaded or downloaded.

Discussion

These constants represent Apple-defined name constants that identify property values universal to all URLs. For a description of the name constants that identify property values specific to HTTP and HTTPS URLs, see "HTTP and HTTPS URL Property Name Constants" (page 76).

You pass one of these name constants in the `property` parameter of the functions `URLSetProperty` (page 36) and `URLGetProperty` (page 35), respectively, to set or obtain a particular property value. Note that you can only set property values identified by the name constants `kURLFileType`, `kURLFileCreator`, `kURLUserName`, and `kURLPassword`. You must also pass the correct data type corresponding to the property value in the `propertyBuffer` parameter of these functions.

URL Access Result Codes

Most URL Access Manager functions return result codes of type `OSStatus`. This includes general result codes such as `noErr`, indicating that the function completed successfully, and `paramErr`, indicating that you passed an invalid parameter.

The result codes specific to URL Access Manager are listed in Table 3-1 (page 81). In some cases, the function result section for a particular function provides more detail about the meaning of the result code specific to that function.

Table 3-1 URL Access Manager result codes

Result code constant	Value	Description
kURLInvalidURLReferenceError	-30770	Returned by functions that operate on URLs to indicate that the URL reference is invalid.
kURLProgressAlreadyDisplayedError	-30771	Returned by the functions URLSimpleDownload (page 38), URLDownload (page 40), URLSimpleUpload (page 43), and URLUpload (page 45) to indicate that a progress indicator is already displayed.
kURLDestinationExistsError	-30772	Returned by the functions URLSimpleUpload (page 43) and URLUpload (page 45) to indicate that the destination file already exists.
kURLInvalidURLError	-30773	Returned by functions that operate on URL references to indicate that the format of the URL is invalid.
kURLUnsupportedSchemeError	-30774	Indicates that the transfer protocol is not supported.
kURLServerBusyError	-30775	Indicates that the server is busy.
kURLAuthenticationError	-30776	Returned by URLSimpleDownload (page 38), URLDownload (page 40), URLSimpleUpload (page 43), and URLUpload (page 45) functions when displaying an authentication dialog box to indicate that user authentication failed. <<correct?>>
kURLPropertyNotYetKnownError	-30777	Returned by the function URLGetProperty (page 35) to indicate that the value of the property is not yet available.

Table 3-1 URL Access Manager result codes

Result code constant	Value	Description
kURLUnknownPropertyError	-30778	Returned by function URLSetProperty (page 36) to indicate that the property is invalid or undefined.
kURLPropertyBufferTooSmallError	-30779	Returned by the function URLGetProperty (page 35) to indicate that the buffer is too small to receive the requested property.
kURLUnsettablePropertyError	-30780	Returned by the function URLSetProperty (page 36) to indicate that the property cannot be set.
kURLInvalidCallError	-30781	Indicates that the call is invalid.
kURLFileEmptyError	-30783	Indicates that the resource file associated with the URL is empty.
kURLExtensionFailureError	-30785	Indicates that your extension failed to load.
kURLInvalidConfigurationError	-30786	Indicates that the configuration is invalid. This is returned when you attempt to FTP to an HTTP proxy, since upload through proxies won't work.
kURLAccessNotAvailableError	-30787	Indicates that the URL Access Manager is not available.
kURLAccessNotAvailableError	-30788	Only available in Mac OS 9. Returned from a printer when URL Access Manager is called from within a 68K context to indicate that it isn't supported.

Document Revision History

This document has had the following releases:

Table 4-1 Transferring Data With the URL Access Manager revision history

Publication date	Notes
May 15, 2000	First public release of document, expanded and updated for URL Access Manager 2.0.3. Documentation includes concepts, tasks, and reference material.
May 7, 1999	First draft of URL Access Manager 1.0 API documentation. This document was distributed in limited release as a seed draft.

CHAPTER 4

Document Revision History