URL Monikers

Last updated: 5/30/96

Distribution: Public

DRAFT

Monikers offer a convenient programming abstraction for *Uniform Resource Locators (URLs)*. This document contains the specification for the URL Moniker, a new *asynchronous* moniker implementation which can be used to encapsulate the locating and download capabilities of URLs.

1 URL Monikers

Introduction

The World Wide Web uses *Uniform Resource Locators* (URLs) to encode the names and addresses of objects on the Internet.[1] Refer to the following pages for background material about URLs:

- Uniform Resource Locators (URL) (http://ds.internic.net/rfc/rfc1738.txt)
- Relative Uniform Resource Locators (http://ds.internic.net/rfc/rfc1808.txt)
- Universal Resource Identifiers in WWW (RFC1630) (http://ds.internic.net/rfc/rfc1630.txt)
- Names and Addresses, URIs, URLs, URNs, URCs (http://www.w3.org/pub/WWW/Addressing/Addressing.html)
- IETF - Hypertext Transfer Protocol (HTTP) Working Group (http://www.ics.uci.edu/pub/ietf/http)

URL syntax was fundamentally designed to be

- **Extensible**, such that new naming schemes could be added later.
- **Complete**, such that it is possible to encode any naming scheme.
- **Printable**, such that it is possible to express any URL string using 7-bit ASCII characters so that URLs can, if necessary, be passed using pen and ink.

Due to these simple goals, the OLE moniker architecture in fact offers a convenient programming model for working with URLs in practice. The moniker architecture supports *extensible* and *complete* name parsing through the MkParseDisplayName(Ex) API and the IParseDisplayName and IMoniker interfaces, as well as *printable* names through IMoniker::GetDisplayName method. Monikers allow applications to spend time and resources on *what* they want to do with objects and resources and to leverage an extensible infrastructure that takes care of *how* things actually gets done, such as downloading files, finding and launching code, or encoding or decoding raw data into appropriate formats. The IMoniker interface is the way you actually *use* URLs you encounter, and building components that fit into the moniker architecture is the way to actually extend URL namespaces *in practice*.

This document describes a new system-provided moniker class, the **URL Moniker**, which provides a framework for building and using certain URLs. Since URLs frequently refer to resources across high-latency networks, the URL Moniker supports asynchronous as well as synchronous binding. Refer to the **Asynchronous Moniker** specification for low-level details about asynchronous binding; this document assumes a working knowledge of that specification.

Overview

The following diagram shows the components involved in using URL monikers. All of these components should be familiar from the Asynchronous Moniker specification.

[1] In reality, *Universal Resource Identifiers* (URIs) are actually the general naming mechanism of the Internet, however, URLs are the common subset of URIs that are actually in use in today's Internet software..

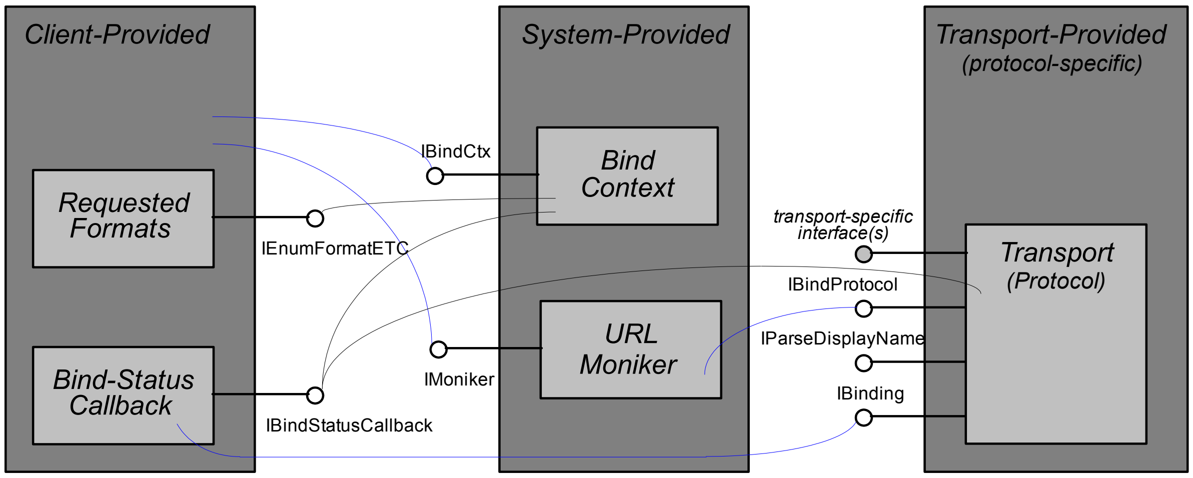

Figure 1. Components Involved in using URL Moniker (small light-gray boxes), who they are implemented by (larger dark-gray boxes), and their references to one-another (dotted lines).

As shown in the diagram, a client or user of URL monikers typically creates and holds a reference to the *moniker* as well as the *bind-context* to be used during binding (BindToStorage or BindToObject). This is identical to normal moniker usage. To support asynchronous binding, the client can implement a *bind-status-callback* object supporting IBindStatusCallback and register it with the bind-context using RegisterBindStatusCallback. This object will receive the IBinding interface of the transport during IBindStatusCallback::OnStartBinding. The URL moniker identifies the protocol by the URL prefix and retrieves the IBinding interface from the transport layer. The client uses IBinding to support pausing, cancellation, and prioritization of the binding operation. The callback object also receives progress notification through IBindStatusCallback::OnProgress, data availability notification through IBindStatusCallback::OnDataAvailable, as well as various other notifications from the transport about the status of the binding. The URL moniker or specific transport layers may also request extended information from the client via IBindStatusCallback::QueryInterface, allowing the client to provide protocol-specific information that will affect the bind operation. To support *media-type negotiation* the client can register a *format enumerator* implementing the IEnumFORMATETC interface on the bind-context using RegisterFormatEnumerator. URL Moniker translates these formats into *MIME Types* when performing binding to HTTP URLs.

Callback Synchronization

The asynchronous WinInet API (used for the most common protocols) leave the synchronization of the callback mechanism and the calling application as an exercise for the client. This is intentional as it allows the greatest degree of flexibility. The default protocols and the URL moniker implementation perform this synchronization and guarantee that single- and apartment-threaded applications will never have to deal with free-thread style contention. That is, the client's IEnumFORMATETC and IBindStatusCallback interfaces are only ever called on their proper thread. This feature is transparent to the user of the URL moniker as long the thread that calls BindToStorage and BindToObject have a message queue.

Fine-grain Priority Control

The Asynchronous Moniker specification requires a finer grain control over the prioritization and management of downloads than allowed for at either the WinSock or WinInet level. URL Moniker actually manages all the downloads for any given caller's thread and using – as part of its synchronization – a priority scheme based on the IBinding specification.

MIME

Many application-level[2] Internet protocols are based on the exchange of messages in a simple, flexible format known as MIME, or *Multipurpose Internet Mail Extensions*. This format was developed for exchanging electronic mail messages with rich content across heterogeneous networking, machine, and e-mail environments. MIME has since been adopted in numerous non-mail applications, and several of its useful core features extended by further protocols, such as *Hyper-Text Transfer Protocol* (HTTP). For an overview of MIME and its use in HTTP, refer to the following pages:

- MIME Overview (http://ds.internic.net/rfc/rfc1630.txt)
- Media Type Registration Procedure (http://ds.internic.net/rfc/rfc1590.txt)
- IETF - Hypertext Transfer Protocol (HTTP) Working Group (http://www.ics.uci.edu/pub/ietf/http)

Media Types (MIME Content Types)

One recurring construct, the MIME *Content Type*, or *Media Type*[3] for short, is used extensively in Web applications to allow data format negotiation between a client and an object or resource. Media Types are simple strings which denote a *type* and *subtype* (such as “text/plain” or “text/html”) and are used to label data or qualify a request. A Web browser, for example, lists as part of an HTTP request-for-data (Get *Request-Line*) or request-for-info (Head *Request-Line*) that it is requesting “image/gif” or “image/jpeg” Media Types, to which a Web server responds by returning (as a *Response*) the Media Type label of “image/gif” and optionally the image data itself in the GIF format if the call was a request-for-data.

Media type negotiation is often similar to how existing desktop applications negotiate with the system clipboard to determine which data format to paste when the users chooses Edit/Paste or when they query for formats when they receive an IDataObject during drag-and-drop. The subtle difference in HTTP media type negotiation is that the client lists media types in fidelity order up-front and the server responds with the best available format rather than the client knowing up-front which formats the server has available.

Media Type Negotiation with URL Moniker

URL monikers support Media Type negotiation in order to allow clients to negotiate the data format to be downloaded in BindToStorage scenarios.

Requesting Media Types

The possible media types requested by the client are represented to URL monikers through FORMATETC structures available from the IEnumFORMATETC enumerator registered by the caller on the bind-context:

```
// implement enumfmtetc to hold a set of FORMATETCs, then do the following or
// use CreateAsyncBindCtx, which encapsulates this functionality
CreateBindCtx(0, &pbc);
RegisterFormatEnumerator(pbc, &enumfmtetc, 0);
```

Each FORMATETC specifies a clipboard format identifying the media type, a NULL target device, DVASPECT_CONTENT, a value of -1 for lindex, and TYMED_NULL, for example:

```
FORMATETC  fmtetc;
fmtetc.cfFormat = RegisterClipboardFormat(CF_MIME_POSTSCRIPT);
fmtetc.ptd = NULL;
fmtetc.dwAspect = DVASPECT_CONTENT;
fmtetc.lindex = -1;
fmtetc.tymed = TYMED_NULL;
```

A special clipboard format value, CF_NULL, is used to indicate that the default media type of the resource pointed to by the URL should be retrieved. This FORMATETC can be placed anywhere within the enumerator to indicate the priority which the client places on the default media type, although it is typically the last format the client is interested in if it is bothering to list formats it prefers. When no enumerator is registered with the bind context, URL Moniker works as if an enumerator containing a

[2] Distinguishable from *transport-level* protocols, such as TCP or UDP.
[3] According to RFC1590, the term “Media Type” is to be used in preference to “MIME Type”.

single FORMATETC with cfFormat=CF_NULL is available; that is, URL Moniker will automatically bind to or download the default media-type of the resource, which is a common case.

Receiving Media Types

In all cases, the client is notified of the actual media type (if applicable) that it receives during BindToStorage through the pformatetc argument on its IBindStatusCallback::OnDataAvailable method.[4]

Overriding Default Viewers

When using IMoniker::BindToObject, it is possible to override the object registered as the default class for handling a particular media type. This is useful, for example, when binding to a .txt file within a web browser−the web browser should be used to view the "text/plain" MIME type even though there is a default viewer registered as the class for .txt files. Note, however, that this function must be used with special care, and it is only advised for advanced programmers.

The RegisterMediaTypeClass API may be used to register a mapping of media types to CLSIDs with a bind context. When binding to objects using this bind context, this mapping overrides the default mapping of file types to CLSIDs. The FindMediaTypeClass API may be used to query the particular CLSID registered on a bind context as the appropriate class for a given media type.

Complex Media Types

How important are Media Type parameters, i.e. "image/jpeg; quality=low"? If parameters are generally used more in a global sense rather than on a per-transfer basis, they should be established elsewhere, perhaps as strings floating around on another object on the bind-context. If not, putting them in the clipboard format by concatenating strings and then using RegisterClipboardFormat is not unreasonable (there are >16000 clipboard formats available to RegisterClipboardFormat)

Also, should clients be able to specify Content-Transfer-Encoding? The answer appears to be no from the client perspective, as it should really be mostly transparent. Will eventually describe how Content-Transfer-Encodings are bound into the HTTP protocol, and how you build a custom decode piece for media-types.

Examples

Downloading an Image from a Full URL

```
// register the clipboard formats and create an IEnumFORMATETC object around them, enumfmtetc
// this is probably such a common form of request that this will be a global object that is multi-usable
cfGIF = RegisterClipboardFormat(CFSTR_MIME_GIF);
cfJPEG = RegisterClipboardFormat(CFSTR_MIME_JPEG);

// create some enumfmtetc object based on the above formats

// create a bind-status-callback object, bsc, that supports IBindStatusCallback

CreateURLMoniker(NULL, L"http://www.foo.com/some_image", &pmk);

// create the bind-context for the bind operation
CreateAsyncBindCtx(0, &bsc, &enumfmtetc, &pbc)

// start the binding operation.
pmk->BindToStorage(pbc, NULL, IID_IStream, &pstm);
pbc->Release();
pmk->Release();

// the actual work to interpret the data stream will occur during ::OnDataAvailable to bsc
// at some later date, at which point the transport will tell us which media type is actually
// being downloaded and chunks of it will begin arriving.
// over the duration of the download, IBindStatusCallback will be receiving progress
// notifications as well, which we can use to update our UI's progress meter
```

Downloading an Image from a Relative/Partial URL

```
// as above, except that the created URL moniker is a partial URL…
CreateURLMoniker(NULL, L"./some_other_image", &pmk);

// …and furthermore the caller registers the full URL of the current document with the bind-context
```

[4] Note: if received content is of an unrecognized MIME type, the new type is automatically registered using RegisterMediaTypes.

```
// before binding so that the partial URL can pull context from the document's full URL, which is
// "http://www.foo.com/main.html"
pbc->RegisterObjectParam(SZ_URLCONTEXT, &m_pmkCurrentDocument);

// as before, except that pmk will end up binding to "http://www.foo.com/some_other_image"
pmk->BindToStorage(pbc, NULL, IID_IStream, &pstm);
…
```

Technical Details of URL Monikers

```
// CLSID_URLMoniker: {79EAC9E0-BAF9-11CE-8C82-00AA004BA90B}
DEFINE_GUID(CLSID_URLMoniker, 0x79eac9e0, 0xbaf9, 0x11ce, 0x8c, 0x82, 0x00, 0xaa, 0x00, 0x4b, 0xa9, 0x0b);

HRESULT     CreateURLMoniker([in] IMoniker* pmkContext, [in] LPWSTR szURL, [out] IMoniker** ppmk);
HRESULT     IsValidURL(LPBC pBC, LPCWSTR szURL, DWORD dwReserved);
HRESULT     RegisterMediaTypes([in] UINT ctypes, [in, length_is(ctypes)] LPTSTR* rgszTypes,
                [out, length_is(ctypes)] CLIPFORMAT* rgcfTypes);
HRESULT     CreateFormatEnumerator([in] UINT cfmtetc, [in, length_is(cfmtetc)] FORMATETC* rgfmtetc,
                [out] IEnumFORMATETC** ppenumfmtetc);
HRESULT     RegisterFormatEnumerator( [in] LPBC pBC, [in] IEnumFORMATETC* pEFetc, [in] DWORD dwReserved);
HRESULT     RevokeFormatEnumerator( [in] LPBC pBC, [in] IEnumFORMATETC* pEFetc);
HRESULT     RegisterMediaTypeClass( [in] LPBC pBC, [in] UINT ctypes, [in] const LPCSTR* rgszTypes,
                [in] CLSID *rgclsID, [in] DWORD dwReserved);
HRESULT     FindMediaTypeClass( [in] LPBC pBC, [in] LPCSTR szType, [out] CLSID *pclsID, [in] DWORD dwReserved);
HRESULT     UrlMkSetSessionOption( [in] DWORD dwOption, [in] LPVOID pBuffer,
                [in] DWORD dwBufferLength, [in] DWORD dwReserved);

#define   CF_NULL                       0
#define   SZ_URLCONTEXT                 (L"URL Context")
#define   CFSTR_MIME_FRACTALS           (TEXT("application/fractals"))
#define   CFSTR_MIME_RAWDATA            (TEXT("application/octet"))
#define   CFSTR_MIME_POSTSCRIPT         (TEXT("application/postscript"))
#define   CFSTR_MIME_AIFF                     (TEXT("audio/aiff"))
#define   CFSTR_MIME_BASICAUDIO         (TEXT("audio/basic"))
#define   CFSTR_MIME_WAV                (TEXT("audio/wav"))
#define   CFSTR_MIME_X_AIFF             (TEXT("audio/x-aiff"))
#define   CFSTR_MIME_X_REALAUDIO        (TEXT("audio/x-pn-realaudio"))
#define   CFSTR_MIME_X_WAV              (TEXT("audio/x-wav"))
#define   CFSTR_MIME_BMP                (TEXT("image/bmp"))
#define   CFSTR_MIME_GIF                (TEXT("image/gif"))
#define   CFSTR_MIME_JPEG               (TEXT("image/jpeg"))
#define   CFSTR_MIME_TIFF               (TEXT("image/tiff"))
#define   CFSTR_MIME_XBM                (TEXT("image/xbm"))
#define   CFSTR_MIME_X_BITMAP           (TEXT("image/x-bitmap"))
#define   CFSTR_MIME_HTML               (TEXT("text/html"))
#define   CFSTR_MIME_TEXT               (TEXT("text/plain"))
#define   CFSTR_MIME_AVI                (TEXT("video/avi"))
#define   CFSTR_MIME_MPEG               (TEXT("video/mpeg"))
#define   CFSTR_MIME_QUICKTIME          (TEXT("video/quicktime"))
#define   CFSTR_MIME_X_MSVIDEO          (TEXT("video/x-msvideo"))
#define   CFSTR_MIME_X_SGI_MOVIE        (TEXT("video/x-sgi-movie"))

// URL Moniker possible error return values - returned in IBindStatusCallback::OnStopBinding
INET_E_INVALID_URL
INET_E_NO_SESSION
INET_E_CANNOT_CONNECT
INET_E_RESOURCE_NOT_FOUND
INET_E_OBJECT_NOT_FOUND
INET_E_DATA_NOT_AVAILABLE
INET_E_DOWNLOAD_FAILURE
INET_E_AUTHENTICATION_REQUIRED
INET_E_NO_VALID_MEDIA
INET_E_CONNECTION_TIMEOUT
INET_E_INVALID_REQUEST
INET_E_UNKNOWN_PROTOCOL
```

IsValidURL

HRESULT IsValidURL(pBC, szURL, dwReserved);

This simple API is used to determine if a given string szURL is a valid absolute URL that may be used to construct a URL moniker. **Note:** relative URLs are not considered “valid” by this API.

Argument	Type	Description
pBC	IBindCtx *	Optional bind context parameter. Currently ignored, should be set to NULL.
szURL	LPCWSTR	The URL to check for validity.
dwReserved	DWORD	Reserved, must be set to zero.
Returns	S_OK	The szURL is a valid URL address.
	S_FALSE	The szURL is **not** a valid URL address.
	E_INVALIDARG	One or more arguments are invalid.

RegisterMediaTypes

HRESULT RegisterMediaTypes(ctypes, rgszTypes, rgcfTypes);

Registers media types strings.

Argument	Type	Description
ctypes	UINT	The number of media type strings in the rgszTypes array. May not be zero.
rgszTypes	LPTSTR*	Array of strings identifying the media types to be registered. None may be NULL.
rgcfTypes	CLIPFORMAT*	An array of 32-bit values that should be assigned to the corresponding media types in rgszTypes.
Returns	S_OK	Success.
	E_INVALIDARG	One or more arguments are invalid.

CreateFormatEnumerator

HRESULT CreateFormatEnumerator(cfmtetc, rgfmtetc, ppenumfmtetc);

Creates an object which implements IEnumFORMATETC over a static array of cfmtetc FORMATETCs.

Argument	Type	Description
cfmtetc	UINT	The number of FORMATETCs in rgfmtetc. May not be zero.
rgfmtetc	CLIPFORMAT*	Static array of formats.
ppenumfmtetc	IEnumFORMATETC**	Location to return the IEnumFORMATETC interface of the enumerator.
Returns	S_OK	Success.
	E_INVALIDARG	One or more arguments are invalid.

RegisterFormatEnumerator

HRESULT RegisterFormatEnumerator(pbc, pEFetc, dwReserved);

Registers a format enumerator object onto the given bind context. This format enumerator is used to determine what format types are prefered for the bind operation.

Argument	Type	Description
pbc	LPBC	The pointer to the bind context.
pEFetc	IEnumFORMATETC *	The format enumerator.
dwReserved	DWORD	Reserved for future use, must be zero.
Returns	S_OK	Success.
	E_INVALIDARG	One or more arguments are invalid.

RevokeFormatEnumerator

HRESULT RevokeFormatEnumerator(pbc, pEFetc);

Removes a format enumerator from the given bind context.

Argument	Type	Description
pbc	LPBC	The pointer to the bind context.
pEFetc	IEnumFORMATETC *	The format enumerator.
Returns	S_OK	Success - the format enumerator was removed.
	E_INVALIDARG	One or more arguments are invalid.

RegisterMediaTypeClass

HRESULT RegisterMediaTypeClass(pbc, ctypes, rgszTypes, rgclsid, dwReserved);

Registers a mapping of media types to CLSIDs to override the default mapping when binding to objects using the given bind context. This function is primarily used by moniker clients (e.g. web browsers) that wish to override the default registry file-type to CLSID mapping. The function creates an object that is registered on the bind context and is used by monikers when choosing the class to instantiate as the result of a IMoniker::BindToObject operation. **Note: this function is only intended for use by very advanced users!**

Argument	Type	Description
pbc	LPBC	The pointer to the bind context.
ctypes	UINT	The number of media type strings in the rgszTypes array. May not be zero.
rgszTypes	LPCSTR *	Array of strings identifying the media types to be registered. None may be NULL.
rgszclsid	CLSID *	Array of CLSIDs corresponding to the given media types.
dwReserved	DWORD	Reserved for future use, must be zero.
Returns	S_OK	Success.
	E_INVALIDARG	One or more arguments are invalid.

FindMediaTypeClass

HRESULT FindMediaTypeClass(pbc, szType, pclsID, dwReserved);

Queries the particular CLSID registered on a bind context as the appropriate class for a given media type. This function is primarily used by a moniker implementation when servicing a call to BindToObject.

Argument	Type	Description
pbc	LPBC	The pointer to the bind context.
szType	LPCSTR	A string identifying the media types. May not be NULL.
pclsid	CLSID *	On return, this contains the corresponding CLSID.
dwReserved	DWORD	Reserved for future use, must be zero.
Returns	S_OK	Success.
	E_INVALIDARG	One or more arguments are invalid.

UrlMkSetSessionOption

HRESULT UrlMkSetSessionOption(dwOption, pBuffer, dwBufferLength, dwReserved);

This API can be used by URL Moniker clients to set options for the current "internet session". This function maps directly to the Windows Internet API InternetSetOption(), although this API only allows session global options to be set. **Note: in order to use this function the client code muse #include the "wininet.h" header file which declares values for the dwOption parameter and structures for the pBuffer parameter.**

Argument	Type	Description
dwOption	DWORD	The session option to set. For instance, a value of INTERNET_OPTION_PROXY allows the client to overwrite the current proxy settings
pBuffer	LPVOID	Buffer containing new session settings. May not be NULL.
dwBufferLength	DWORD	The size of pBuffer.
dwReserved	DWORD	Reserved for future use, must be zero.
Returns	S_OK	Success.
	E_INVALIDARG	One or more arguments are invalid.

URL Moniker class

CreateURLMoniker

HRESULT CreateURLMoniker(pmkContext, szURL, ppmk);

Creates a URL moniker from either a full URL string or from a base context URL moniker and a partial URL string.

Partial URLs are similar to relative paths within file systems in that resolution to an object requires context outside the partial string alone, whereas full URL strings are like fully-qualified paths – self-contained and often location independent. When creating an URL moniker from a partial URL string, the caller may specify through pmkContext the URL moniker to draw context from when creating a full URL string from the partial URL string. In this case, CreateURLMoniker retrieves the display name of pmkContext (via IMoniker::GetDisplayName) and manually composes it with szURL according to URL composition rules. The caller may alternately create a moniker from a partial URL string without context (pmkContext=NULL), in which case the resulting moniker will draw further context during binding (IMoniker::BindToObject and IMoniker::BindToStorage) by looking first in the passed IBindCtx for a contextual URL moniker object parameter by using IBindCtx::GetObjectParam(SZ_URLCONTEXT, (IUnknown**)&pmkContext), and next to its left for another URL Moniker from which to draw context.

Argument	Type	Description
pmkContext	IMoniker*	The URL to use as the base context when szURL is a partial URL string. NULL when szURL is a full URL string or if this moniker will retrieve full URL context from its left or from the bind-context during IMoniker::BindToObject or IMoniker::BindToStorage.
szURL	LPWSTR	The display name to be parsed.
ppmk	IMoniker**	Location to return a moniker if successful.
Returns	S_OK	Success.
	E_OUTOFMEMORY	Out of memory.
	MK_E_SYNTAX	A moniker could not be created because szURL does not correspond to valid URL syntax for a full or partial URL. This is uncommon, since most parsing of the URL occurs during binding and also since the syntax for URLs is extremely flexible.

URL Moniker–IUnknown::QueryInterface

URL Moniker supports IUnknown, IAsyncMoniker, IPersist, IPersistStream, and IMoniker. Recall that IMonikerAsync is simply IUnknown (there are no actual methods), and is used to allow clients to determine transparently if a moniker supports asynchronous binding.

URL Moniker–IPersist::GetClassID

Returns CLSID_StdURLMoniker.

URL Moniker–IPersistStream::IsDirty

Returns S_OK if the Moniker has changed since it was last saved (IPersistStream::Save with fClearDirty==TRUE), S_FALSE otherwise.

URL Moniker–IPersistStream::Load

Initializes a URL moniker from data within a stream, usually stored there previously using its IPersistStream::Save (via OleSaveToStream or OleSaveToStreamEx). The binary format of URL Moniker is its URL string in Unicode™ (may be a full or partial URL string, see CreateURLMoniker for details). This is represented as a ULONG (32-bit) count of characters followed by that many Unicode characters.

URL Moniker–IPersistStream::Save

Saves a URL moniker to a stream. The binary format of URL Moniker is its URL string in Unicode (may be a full or partial URL string, see CreateURLMoniker for details). This is represented as a ULONG (32-bit) count of characters followed by that many Unicode™ characters.

URL Moniker–IPersistStream::GetSizeMax

Returns the maximum number of bytes in the stream that will be required by a subsequent call to IPersistStream::Save. This value is SIZEOF(ULONG)==4 plus SIZEOF(WCHAR)*n where n is the length of the full or partial URL string including the NULL terminator.

URL Moniker–IMoniker::BindToObject

Since the URL Moniker supports asynchronous binding, the actual return value of its BindToObject may vary depending on the object parameters established in the bind-context, however the semantics of the bind operation are identical regardless of synchronous or asynchronous usage, and are as follows:

1. URL Moniker pulls further information for the bind operation from the bind-context (for example, the IBindStatusCallback and IEnumFORMATETC interfaces).[5]
2. It next checks the Running Object Table of the bind-context to determine if it is already running, using IBindCtx::GetRunningObjectTable(&prot) followed by prot->IsRunning(this). If it is already running, it retrieves the running object with prot->GetObject(this, &punk) and QueryInterface's for the requested interface.
3. Otherwise, it queries the client in IBindStatusCallback::GetBindInfo, initiates the bind operation, and passes the resulting IBinding to the client via IBindStatusCallback::OnStartBinding.
4. If in step 1 it was determined that this was an asynchronous bind, BindToObject returns MK_S_ASYNCHRONOUS at this point with NULL in ppv. The caller will receive the actual object pointer during IBindStatusCallback::OnObjectAvailable at some later point. The following steps then occur asynchronously to the caller, typically on another thread of execution.
5. The class of the resource designated by the URL Moniker is next determined in one of the following ways:
   - In the case of HTTP, the initial HTTP response packet header may contain the CLSID of the actual (Get) or referred to (Head) resource as an *extension-header* to the *Entity Header* section of the *Full-Response* message of the form:

     ```
     CLSID            = "CLSID" ": " stringized-clsid
     ```

     where stringized-clsid can be created using StringFromCLSID and interpreted using CLSIDFromString. New HTTP servers can support this functionality readily.[6]
   - URL Moniker examines the media type of the data. If the media type is "application/x-oleobject"[7] the first 16-bytes of the actual data (*Content-Body*) contain the CLSID of the resource and subsequent data is to be interpreted by the class itself. For all other media types, URL Moniker looks in the system registry for the HKEY_CLASSES_ROOT\MIME\Database\Content-Type\<media-type>\CLSID key.
   - URL Moniker matches portions of arriving data to patterns registered in the system registry under HKEY_CLASSES_ROOT\FileTypes.
   - Finally, if all else fails, URL Moniker correlates the trailing extension of the resource, if any, to a CLSID using the HKEY_CLASSES_ROOT\.??? keys in the system registry, as is done by GetClassFile and the shell.

1. Having determined the class, URL moniker creates an instance using CoCreateInstance of CLSCTX_SERVER asking for the IUnknown interface.
2. URL Moniker next QueryInterfaces the newly created object for IPersistMoniker and if successful calls IPersistMoniker::Load passing itself (this) as the moniker parameter. The object typically turns around and calls IMoniker::BindToStorage asking for the storage interface that they're interested in.
3. Otherwise, URL Moniker QueryInterfaces for IPersistStream and if successful calls IPersistStream::Load, passing the object an IStream which is being filled asynchronously by the transport. If the class being called is not marked with the category CATID_AsyncAware, calls to IStream::Read or IStream::Write which reference data not yet available block[8] until the data becomes available. If the class is marked with the category CATID_AsyncAware, calls to IStream::Read or IStream::Write which reference data not yet available return E_PENDING.
4. Otherwise, URL Moniker asks for QueryInterfaces for IPersistFile, and if successful completes the download into a temporary file and then calls IPersistFile::Load. The created file is cached along with other Internet-downloaded data. The client must be sure not to delete this file.

[5] Note that "further information" may include additional bind options specified on the bind context BIND_OPTS using IBindCtx::SetBindOptions, such as dwTickCountDeadline or the grfFlags value of BIND_MAYBOTHERUSER.

[6] Refer to the HTTP Specification, Hypertext Transfer Protocol – HTTP 1.0 (http://www.ics.uci.edu/pub/ietf/http/draft-ietf-http-v10-spec-04.html), as well as further references at IETF - Hypertext Transfer Protocol (HTTP) Working Group (http://www.ics.uci.edu/pub/ietf/http) for details about *extension-headers*.

[7] This media-type will be used initially, until we receive approval for "application/oleobject" from IANA as outlined in Media Type Registration Procedure (http://ds.internic.net/rfc/rfc1590.txt).

[8] These calls block in the traditional OLE-sense: a message-loop is entered which allows certain messages to be processed and the IMessageFilter of the thread is called appropriately. See the Win32 documentation of IMessageFilter for details.

5. When the object returns from one of the various IPersistXXX::Load calls above, URL Moniker returns the client the interface pointer originally requested in IMoniker::BindToObject using the callback IBindStatusCallBack::OnObjectAvailable.

URL Moniker–IMoniker::BindToStorage

The system implementation of URL Moniker supports BindToStorage for IStream on all URLs and for IStorage in the case where the designated resource is in fact a compound file.[9]

Since the URL Moniker supports asynchronous binding, the actual return value of its BindToStorage may vary depending on the object parameters established in the bind-context, however the semantics of the bind operation are identical regardless of synchronous or asynchronous usage, and are as follows:

1. URL Moniker pulls further information for the bind operation from the bind context (for example, the IBindStatusCallback and the IEnumFORMATETC interfaces).[10] The Moniker then queries the client in IBindStatusCallback::GetBindInfo, initiates the bind operation with the transport, and passes the resulting IBinding to the client via IBindStatusCallback::OnStartBinding.
2. If the caller requested an asynchronous IStream or IStorage via the BINDF_ASYNCSTORAGE flag in the BINDINFO retrieved from IBindStatsCallback::GetBindInfo, URL Moniker returns the object as soon as possible. Calls to these IStorage or IStream objects which reference data not yet available return E_PENDING.
3. If the caller does not specify asynchronous IStream or IStorage as described above, URL Moniker will still return an object through IBindStatusCallback::OnDataAvailable as soon as possible. However calls to these objects which reference data not yet available will block until the data becomes available. For some applications this will require the least modification of their existing I/O code, yet may still result in improved performance depending on their access-patterns.

URL Moniker–IMoniker::Reduce

Returns MK_S_REDUCED_TO_SELF and itself (this) in *ppmkReduced.

URL Moniker–IMoniker::ComposeWith

URL Monikers support composition of two URLs (a base URL composed with a relative URL). This composition is done according to the RFC on relative URLs[11]. URL monikers do *not* currently support the composition of a base URL moniker with a relative *file* moniker, although this may be supported in the future. However, URL Monikers do support generic composition. If fOnlyIfNotGeneric==FALSE, this method returns CreateGenericComposite(this, pmkRight, ppmkComposite). See the Win32 documentation about IMoniker::ComposeWith for details.

URL Moniker–IMoniker::Enum

Returns S_OK and sets *ppenumMoniker to NULL, indicating that the moniker does not contain sub-monikers.

URL Moniker–IMoniker::IsEqual

Returns S_FALSE if the other moniker (pmkOtherMoniker) is not a URL moniker, which it checks using IPersist::GetClassID to see if the CLSID is CLSID_URLMoniker. If the other moniker is an URL moniker, it compares the display names of the monikers for equality, returning either S_OK if they are identical or S_FALSE if not.

[9] In the future support for ILockBytes may also be added.

[10] Note that "further information" may include additional bind options specified on the bind context BIND_OPTS using IBindCtx::SetBindOptions, such as dwTickCountDeadline or the grfFlags value of BIND_MAYBOTHERUSER.

[11] See http://ds.internic.net/rfc/rfc1808.txt.

URL Moniker–IMoniker::Hash

Creates a hash value based on the URL string of the moniker. This hash value is identical when URL strings are identical, although it may also be identical for different URL strings. This method is used to speed up comparisons by reducing the amount of time that it is necessary to call IsEqual.

URL Moniker–IMoniker::IsRunning

Returns S_OK if this moniker is currently "running", otherwise returns S_FALSE. URL Moniker determines if it is running by first checking if it is equal to the newly running moniker (by calling pmkNewlyRunning->IsEqual(this) which is typically an inexpensive operation) and next by checking if it is registered with the Running Object Table of the passed-in bind-context.

URL Moniker–IMoniker::GetTimeOfLastChange

Returns the time of last change of an object that is registered in the running object table

URL Moniker–IMoniker::Inverse

Returns MK_E_NOINVERSE.

URL Moniker–IMoniker::CommonPrefixWith

Currently returns E_NOTIMPL. May in the future properly compute the proper common prefix of two URL monikers. See the Win32 documentation about IMoniker::CommonPrefixWith for details.

URL Moniker–IMoniker::RelativePathTo

Returns E_NOTIMPL. May in the future properly compute the relative path between two URL monikers. See the Win32 documentation about IMoniker::RelativePathTo for details.

URL Moniker–IMoniker::GetDisplayName

URL Moniker attempts to return its full URL string. If the moniker was created with a partial URL string (see CreateURLMoniker), it will first attempt to find an URL moniker in the bind-context under SZ_URLCONTEXT, and will next look to the moniker to its left for contextual information. If it can not return its full URL string, it will return its partial URL string.

URL Moniker–IMoniker::ParseDisplayName

Parses a full or partial URL string into a result moniker (ppmkOut). If szDisplayName represents a full URL string (i.e. "http://foo.com/default.html"), the result is a new full URL moniker. If szDisplayName represents a partial URL string (i.e. "..\default.html"), the result is a full URL that takes its context from either the bind-context's SZ_URLCONTEXT object-parameter or from this URL moniker (i.e., if the context moniker was "http://foo.com/pub/list.html" and szDisplayName was "..\default.html", the resulting URL moniker would represent "http://foo.com/default.html").

URL Moniker–IMoniker::IsSystemMoniker

Returns S_OK and MKSYS_URLMONIKER in *pdwMksys.

Extension services requested during a bind operation

A URL bind may require additional services from the client in order to complete negotiations necessary for a download operation. These services may be additional callbacks that are implemented by the client and are requested by the moniker. These callback extensions are commonly requested using IBindStatusCallback::QueryInterface. However, a moniker client may also provide these extension callback interfaces via an IServiceProvider interface. After the moniker uses IBindStatusCallback::QueryInterface to directly query the client for an extension interface, the moniker will then query for the IServiceProvider interface, and will then try using IServiceProvider::QueryService to query for the desired extension interface.[12]

Two such extension callback services that may be needed for URL downloads are authentication and "generic HTTP negotiation". The corresponding interfaces requested in IBindStatusCallback::QueryInterface or IServiceProvider::QueryService are IAuthenticate and IHttpNegotiate. A more generic interface is IWindowForBindingUI, an interface which is requested from the client when the moniker wishes to display UI. Another extension callback is IHttpSecurity, an interface which a client may implement in order to provide custom UI or UI-less operation when HTTP security issues arise. Note that for all these extension callbacks, if there are multiple clients for an asynchronous moniker bind operation (e.g. an OLE object and its container are both receiving callbacks), it is usually the responsibility of the *container* to implement the extension callbacks, thus relieving the *contained* object (e.g. controls) of extra work.

Alternatively, a client may request protocol-specific services from the URL moniker during a bind operation. Such services are requested by querying for additional interfaces from the IBinding object corresponding to a particular bind operation. Two such protocol-specific service are IWinInetInfo and IWinInetHttpInfo, interfaces used for querying specific information related to Internet protocols.

IWindowForBindingUI

This simple interface is implemented by URL Moniker clients that wish to allow the moniker to display UI when necessary. As with other callback extension interfaces, this interface is usually implemented by OLE containers that also implement the IBindHost interface. Note: the URL Moniker will not request this interface from clients unless the bind context BIND_OPTS specify the grfFlags value of BIND_MAYBOTHERUSER using IBindCtx::SetBindOptions.[13] The IWindowForBindingUI interface is very similar to the existing IOleWindow interface. A new interface is used here because IOleWindow is semantically used only for in-place activation.

```
interface IWindowForBindingUI : IUnknown {
    HRESULT    GetWindow([in] REFGUID rguidReason, [out] HWND* phwnd);
};
```

IWindowForBindingUI::GetWindow

This function is called by the URL Moniker when it needs a window to present UI during a bind operation.

Argument	Type	Description
rguidReason	REFGUID	The reason why the moniker is requesting to display UI during a bind operation. This value may be one of IID_IAuthenticate, IID_IHttpSecurity, IID_ICodeInstall, or any other future reasons for displaying UI.
phwnd	HWND *	Client-provided HWND of the parent window to use for displaying UI.
Returns	S_OK	Success.
	S_FALSE	No window is available for UI.
	E_INVALIDARG	One or more arguments are invalid.

IAuthenticate

This interface is implemented by URL Moniker clients that are interested in participating in user authentication. This interface allows clients a chance to display custom UI or use a custom password-list to specify a username and password for accessing secure Internet resources. **Note: this interface should only be implemented by URL Moniker clients that wish to display custom authentication UI or wish to do UI-less authentication. If default UI is acceptable, implementing IWindowForBindingUI is all that is needed to make sure bind operations go smoothly.**

[12] Because IServiceProvider::QueryService is not restricted by COM identity rules in the same way as QueryInterface, this mechanism allows moniker clients to delegate such extension services to other objects.

[13] The CreateAsyncBindCtx API will automatically set this flag on all bind contexts.

```
interface IAuthenticate : IUnknown {
      HRESULT      Authenticate( [out] HWND* phwnd, [out] LPWSTR *pszUsername, [out] LPWSTR *pszPassword);
};
```

IAuthenticate::Authenticate

This function is called by the URL Moniker when it needs basic authentication information from a bind client. The client may choose to return username and password strings, or it may provide an HWND that is used to present default authentication UI.

Argument	Type	Description
phwnd	HWND *	Client-provided HWND of the parent window for default authentication UI. If no UI is desired, the client must provide a username and password in the other parameters, and this handle.is set to the value -1.
pszUsername	LPWSTR *	Client-provided username for authentication. If the client returns a value here it should also set *phwnd = -1.
pszPassword	LPWSTR *	Client-provided password for authentication. If the client returns a value here it should also set *phwnd = -1.
Returns	S_OK	Success.
	E_INVALIDARG	One or more arguments are invalid.

IHttpNegotiate

This interface is implemented by those moniker clients that are interested in participating in the HTTP negotiations that take place when binding to “http:” URLs. The callbacks in this interface provide clients with the opportunity to add headers to HTTP requests and to examine HTTP response headers.[14] Note that this callback may be sent to multiple clients of an HTTP bind operation−each client may participate in the HTTP negotiation process by looking at existing request headers and adding additional ones.

```
interface IHttpNegotate : IUnknown {
      HRESULT      BeginningTransaction( [in] LPCWSTR szURL, [in] DWORD dwReserved,
                                  [in] LPCWSTR szHeaders, [out] LPWSTR *pszAdditionalHeaders);
      HRESULT      OnResponse( [in] DWORD dwResponseCode, [in] LPCWSTR szResponseHeaders,
                                  [in] LPCWSTR szRequestHeaders, [out] LPWSTR *pszAdditionalRequestHeaders);
      };
```

IHttpNegotiate::BeginningTransaction

This function is called by the URL Moniker before sending an HTTP request. It notifies the client of the URL being bound to at the beginning of the HTTP transaction. It also allows the client to add additional headers (such as Accept-Language) to the request.

[14] Note: although this interface allows clients to add HTTP headers, not all header types may be added at this point. Specifically, clients should not use this mechanism for adding Content-Length or Accept headers to HTTP requests. Instead of adding Accept headers, the RegisterFormatEnumerator API should be used for specifying accepted types for a bind operation.

Argument	Type	Description
szURL	LPCWSTR	The URL for the HTTP transaction.
dwReserved	DWORD	Reserved for future use.
szHeaders	LPCWSTR	The current request headers.
pszAdditionalHeaders	LPWSTR *	Optional additional headers to append to the HTTP request. If these conflict with existing values in szHeaders, then the new request headers take precedence. **Note: if no headers are provided, then no headers are appended to the HTTP request.**
Returns	S_OK	Success, append the headers (if any) in pszAdditionalHeaders to the HTTP request headers.
	E_INVALIDARG	The argument is invalid.
	E_ABORT	Abort the HTTP transaction.

IHttpNegotiate::OnResponse

This function is called upon receiving a response to an HTTP request. In success cases, this callback allows the client of a bind operation to examine the response headers and possibly abort the bind operation. In error cases (errors that cannot be resolved using default behavior), this callback allows the client to add HTTP headers to the request before it is sent again.[15]

Argument	Type	Description
dwResponseCode	DWORD	HTTP response code (see HTTP specification).
szResponseHeaders	LPCWSTR	Response headers from the HTTP server.
szRequestHeaders	LPCWSTR	In dwResponseCode error cases, the HTTP headers that will be used when the request is resent.
pszAdditionalRequestHeaders	LPWSTR *	In dwResponseCode error cases, the client may use this optional parameter to add to the request headers before resending. If the specified header value conflicts with existing values in szRequestHeaders, then the new request headers take precedence. **Note: if no headers are provided in this parameter, then no headers are appended to the HTTP request.**
Returns	S_OK	Success. In dwResponseCode error cases, append pszAdditionalRequestHeaders (if any) to the resent request headers.
	E_INVALIDARG	The argument is invalid.
	E_ABORT	Abort the HTTP transaction.

IHttpSecurity

This interface is implemented by URL Moniker clients that are interested in hearing about HTTP security problems. This interface allows clients a chance to display custom UI or make decisions about whether or not to ignore various security issues. **Note: this interface should only be implemented by URL Moniker clients that wish to display custom UI or wish to make security decisions without displaying UI. If default UI is acceptable, implementing IWindowForBindingUI is all that is needed to make sure bind operations go smoothly.** URL Moniker will query for this interface and use it to resolve

[15] Note: if there are multiple callbacks registered on the BindCtx for a bind operation, it is possible that more than one client provides an IHttpNegotiate callback. In such cases, every registered client receives the IHttpNegotiate callback methods, and each is given a chance to add HTTP headers or to abort the HTTP transaction. In such cases, the last client to receive the callback (the client driving the download operation) will dictate the final decision.

security issues if it is implemented by the client, but if the interface is absent the URL Moniker will resort to using IWindowForBindingUI in order to display default UI. **Note: in order to use this interface, client code must also #include the wininet.h header file because of flag definitions in that header file.**

```
interface IHttpSecurity : IWindowForBindingUI {
    HRESULT     OnSecurityProblem( [in] DWORD dwProblem);
};
```

IHttpSecurity::OnSecurityProblem

This function is called by the URL Moniker when it needs to decide whether or not to abort a bind operation when a security problem has occured. The client may choose to abort the bind operation, continue the operation, or “claim ignorance”. Note that an URL Moniker client may choose to ignore all security issues by specifying BINDF_IGNORESECURITYPROBLEM during IBindStatusCallback::GetBindInfo

Argument	Type	Description
dwProblem	DWORD	Identifies the security problem that has occurred. Possible values are the security related error return values from the Windows Internet API InternetSendRequest(), such as ERROR_INTERNET_SEC_CERT_DATE_INVALID, ERROR_INTERNET_SEC_CERT_CN_INVALID, ERROR_INTERNET_HTTP_TO_HTTPS_ON_REDIR, ERROR_INTERNET_HTTPS_TO_HTTP_ON_REDIR.
Returns	S_OK	The client wishes to continue the bind operation.
	S_FALSE	The client does not understand the security problem.
	E_ABORT	The client wishes to abort the bind operation..

IWinInetInfo

This interface is implemented by the IBinding object for standard Internet protocols. An URL Moniker client may request this interface via IBinding::QueryInterface, and may then query protocol-specific information from the IBinding object. **Note: in order to use this interface, client code must also #include the wininet.h header file because of flags and structure definitions in that header file.**

```
interface IWinInetInfo : IUnknown {
    HRESULT     QueryOption([in] DWORD dwOption, [out] LPVOID pBuffer, [in/out] DWORD *pcbBuf);
};
```

IWinInetInfo::QueryOption

This function is called by a client of URL Moniker in order to query protocol-specific information pertaining to Internet bind operations. The implementation of this method maps directly to the Windows Internet InternetQueryOption() API.

Argument	Type	Description
dwOption	DWORD	Specifies what information to query. Valid values are documented along with InternetQueryOption.
pBuffer	LPVOID	Buffer to receive the result of the query.
pcbBuf	DWORD *	Pointer to the size of the given buffer. Upon return, this contains the length of the data written into pBuffer.
Returns	S_OK	Success.
	E_INVALIDARG	One or more arguments are invalid.

IWinInetHttpInfo

This interface is implemented by the IBinding object for bind operations to HTTP URLs. An URL Moniker client may request this interface via IBinding::QueryInterface, and may then query HTTP-specific information from the IBinding object. **Note: in order to use this interface, client code must also #include the wininet.h header file because of flags and structure definitions in that header file.**

```
interface IWinInetHttpInfo : IWinInetInfo {
        HRESULT        QueryInfo([in] DWORD dwOption, [out] LPVOID pBuffer, [in/out] DWORD *pcbBuf,
                                        [in/out] DWORD *pdwFlags, [in] DWORD dwReserved);
};
```

IWinInetHttpInfo::QueryInfo

This function is called by a client of URL Moniker in order to query HTTP-specific information pertaining to Internet bind operations. The implementation of this method maps directly to the Windows Internet HttpQueryInfo() API.

Argument	Type	Description
dwOption	DWORD	Specifies what information to query and flags which modify the request. Valid values are documented along with HttpQueryInfo.
pBuffer	LPVOID	Buffer to receive the result of the query.
pcbBuf	DWORD *	Pointer to the size of the given buffer. Upon return, this contains the length of the data written into pBuffer.
pdwIndex	DWORD *	A pointer to a zero-based index. See documentation for HttpQueryInfo for details.
dwReserved	DWORD	Reserved for future use, must be zero.
Returns	S_OK	Success.
	E_INVALIDARG	One or more arguments are invalid.

Technical Review of Monikers

```
interface IParseDisplayName : IUnknown {
        HRESULT        ParseDisplayName([in] IBindCtx* pbc, [in] LPWSTR szDisplayName, [out] ULONG* pcchEaten, [out] IMoniker**
                ppmkOut);
        };
HRESULT        MkParseDisplayNameEx([in] IBindCtx* pbc, [in] LPWSTR szDisplayName, [out] ULONG* pcchEaten, [out] IMoniker** ppmk);
```

MkParseDisplayNameEx

HRESULT MkParseDisplayNameEx(pbc, szDisplayName, pcchEaten, ppmk);

Given a string, this function returns a moniker of the object that the string denotes. This operation is known as *parsing.* A display name is parsed into a moniker; it is resolved into its component moniker parts.

If a syntax error occurs, than an indication of how much of the string was successfully parsed is returned in pcchEaten and NULL is returned through ppmk. Otherwise, the value returned through pcchEaten indicates the entire size of the display name.

This API differs from the original MkParseDisplayName in that it supports *Universal Resource Indicator* (URI) syntax as established in IETF RFC1630 (http://ds.internic.net/rfc/rfc1590.txt).

Argument	Type	Description
pbc	IBindCtx*	The binding context in which to accumulate bound objects.
szDisplayName	LPCWSTR	The display name to be parsed.
pcchEaten	ULONG*	On exit the number of characters of the display name that was successfully parsed. Most useful on syntax error, when a non-zero value is often returned and therefore a subsequent call to MkParseDisplayNameEx with the same pbc and a shortened szDisplayName should return a valid moniker.
ppmk	IMoniker**	Location to return a moniker if successful.
Returns	S_OK	Success.
	MK_E_SYNTAX	Parsing failed because szDisplayName could only be partially resolved into a moniker. In this case, *pcchEaten holds the number of characters that were successfully resolved into a moniker prefix.
	E_OUTOFMEMORY	Out of memory.

Parsing a display name may in some cases be as expensive as binding to the object that it denotes, since along the way various non-trivial name space managers (such as a spreadsheet application that can parse into ranges in its sheets) need to be connected to by the parsing mechanism to succeed. As might be expected, objects are not released by the parsing operation itself, but are instead handed over to the passed-in binding context (via IBindCtx::RegisterObjectBound). Thus, if the moniker resulting from the parse is immediately bound using this same binding context, redundant loading of objects is maximally avoided.

In many other cases, however, parsing a display name may be quite inexpensive since a single name-space manager may quickly return a moniker that will perform further expensive analysis on any acceptable name during IMoniker::BindToObject or other methods. An example of such an inexpensive parser is the Win32 implementation of a File Moniker. A theoretical example would be a naïve URL moniker which parsed from any valid URL strings (i.e., “http:...”, “file:...”) and only during binding took time to resolve the string against the Internet, a potentially expensive operation.

The parsing process is an inductive one, in that there is an initial step that gets the process going, fol-lowed by the repeated application of an inductive step. At any point after the beginning of the parse, a certain prefix of szDisplayName has been parsed into a moniker, and a suffix of the display name remains not understood. This is illustrated in Figure 2.

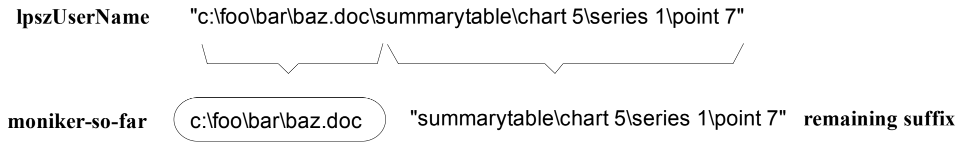
lpszUserName
"c:\foo\bar\baz.doc\summarytable\chart 5\series 1\point 7"
moniker-so-far
c:\foo\bar\baz.doc
"summarytable\chart 5\series 1\point 7"
remaining suffix

Figure 3. Intermediate stage in parsing a display name into a moniker.

The *inductive step* asks the moniker-so-far using IMoniker::ParseDisplayName to consume as much as it would like of the remaining suffix and return the corresponding moniker and the new suffix. The moniker is composed onto the end of the existing moniker-so-far, and the process repeats.

Implementations of IMoniker::ParseDisplayName vary in exactly where the knowledge of how to carry out the parsing is kept. Some monikers by their nature are only used in particular kinds of containers. It is likely that these monikers themselves have the knowledge of the legal display name syntax within the objects that they themselves denote, and so they can carry out the processes completely within IMoniker::-ParseDisplayName. The common case, however, is that the moniker-so-far is generic in the sense that is not specific to one kind of container, and thus cannot know the legal syntax for elements within the con tain-er. File monikers are an example of these, as are Item Monikers. These monikers in general employ the following strategy to carry out parsing. First, the moniker connects to the *class* of object that it currently denotes, asking for IParseDisplayName interface. If that succeeds, then it uses the obtained interface pointer to attempt to carry out the parse. If the class refuses to handle the parse, then the moniker binds to the *object* it denotes, asking again for IParseDisplayName interface. If this fails, then the parse is aborted.

The effect is that ultimately an object always gets to be in control of the syntax of elements contained inside of itself. It's just that objects of a certain nature can carry out parsing more efficiently by having a moniker or their class do the parsing on their behalf.

Notice that since MkParseDisplayNameEx knows nothing of the legal syntax of display names (with the exception of the initial parsing step; see below). It is of course beneficial to the user that display names in different contexts not have gratuitously different syntax. While there some rare situations which call for special purpose syntax, it is recommended that, unless there are compelling reasons to do otherwise, the syntax for display names should be the same as or similar to the native file system syntax; the aim is to build on user familiarity. Most important about this are the characters allowed for the delimiters used to separate the display name of one of the component monikers from the next. Unless through some special circumstances they have *very* good reason not to, all moniker implementations should use inter-moniker delimiters from the character set:

```
\ / : ! [
```

Standardization in delimiters promotes usability. But more importantly, notice that the parsing algorithm has the characteristic that a given container consumes as much as it can of the string being parsed before passing the remainder on to the designated object inside themselves. If the delimiter expected of the next-to-be-generated moniker in fact forms (part of) a valid display name in the container, then the container's parse will consume it!

Monikers and objects which have implementations on more than one platform (such as File Monikers) should always parse according to the syntax of the platform on which they are currently running. When asked for their display name, monikers should also show delimiters appropriate to the platform on which they are currently running, even if they were originally created on a different platform. In total, users will always deal with delimiters appropriate for the host platform.

The *initial step* of the parsing process is a bit tricky, in that it needs to somehow determine the initial moniker-so-far. MkParseDisplayNameEx is omniscient with respect to the syntax with which the display name of a moniker may legally begin, and it uses this omniscience to choose the initial moniker.

The initial moniker is determined by trying the following strategies in order, using the first to succeed.

1. ***"ProgID:" Case:*** *If a prefix of szDisplayName conforms to the legal ProgID syntax, is more than 1 character long, and is followed by a colon (':'), the ProgID is converted to a CLSID with CLSIDFromProgID, an instance of this class is asked for the IParseDisplayName interface, and IParseDisplayName:ParseDisplayName is called with the entire szDisplayName.*[16]
2. **ROT Case:** All prefixes of szDisplayName that consist solely of valid file name characters are consulted as file monikers in the Running Object Table.
3. **File-System Case:** The file system is consulted to check if a prefix of szDisplayName matches an existing file. Said file name may be drive absolute, drive relative, working-directory relative, or begin with an explicit network share name. This is a common case.
4. **"@ProgID" Case:** If the initial character of szDisplayName is '@', then the maximal string immediately following the '@' which conforms to the legal ProgID syntax is determined. This is converted to a CLSID with CLSIDFromProgID. An instance of this class is asked in turn for IParseDisplayName interface; the IParseDisplayName interface so found is then given the whole string (starting with the '@') to continue parsing.

IParseDisplayName Interface

The IParseDisplayName interface is implemented by objects supporting their own namespace and with custom requirements for parsing it.

[16] This case distinguishes MkParseDisplayNameEx from MkParseDisplayName.

IParseDisplayName::ParseDisplayName

HRESULT IParseDisplayName::ParseDisplayName(pbc, szDisplayName, pcchEaten, ppmkOut)

Parse szDisplayName and return a moniker representing it. In general, the maximal prefix of szDisplayName which is syntactically valid and which currently *represents an existing object* should be consumed.

The initial step of MkParseDisplayName(Ex) may retrieve this interface directly from an instance of the class identified with either "@ProgID" or "ProgID:" notation, or later parsing steps may request this object on an intermediary object.

The main loop of MkParseDisplayName(Ex) finds the next piece moniker piece by calling the IMoniker-equivalent function (IMoniker::ParseDisplayName) on the moniker-so-far that it holds on to, passing NULL through pmkToLeft. In the case that the moniker-so-far is a generic composite, this is forwarded by that composite onto its last piece, passing the prefix of the composite to the left of the piece in pmkToLeft.

Some moniker classes will be able to handle this parsing internally to themselves since they are designed to designate only certain kinds of objects. Others will need to bind to the object that they designate in order to accomplish the parsing process. As is usual, these objects should not be released by IMoniker::ParseDisplayName but instead should be transferred to the bind context (via IBindCtx::RegisterObjectBound or IBindCtx::GetRunningObjectTable followed by IRunningObjectTable::Register) for release at a later time.

If a syntax error occurs, then NULL should be returned through ppmkOut and MK_E_SYNTAX returned. In addition, the number of characters of the display name that were *successfully* parsed should be returned through pcchEaten.

Argument	Type	Description
pbc	IBindCtx*	The binding context in which to accumulate bound objects.
szDisplayName	LPWSTR	The display name to be parsed.
pcchEaten	ULONG*	The number of characters of the input name that this parse consumed.
ppmkOut	IMoniker*	Location to return a result moniker if successful.
Returns	S_OK	Success.
	E_INVALIDARG	One or more arguments are invalid.
	MK_E_SYNTAX	Parsing failed because szDisplayName could only be partially resolved into a moniker. In this case, *pcchEaten holds the number of characters that were successfully resolved into a moniker prefix.