

7. Linking and Naming Support

OLE 1 introduced the notion two architecturally distinct kinds of compound-document objects: *links* and *embeddings*. The fundamental distinction between the two is that of where the bits that are the native representation of the object are persistently stored. With an embedded object, these bits are stored inside the file[61] of the container document, which also contains the presentation of the object. As a result, the object appears to be part of the document. In contrast, the storage for an link object is not kept with the object itself, but is found elsewhere. Figure 72 illustrates this difference in storage between a linked and an embedded object.

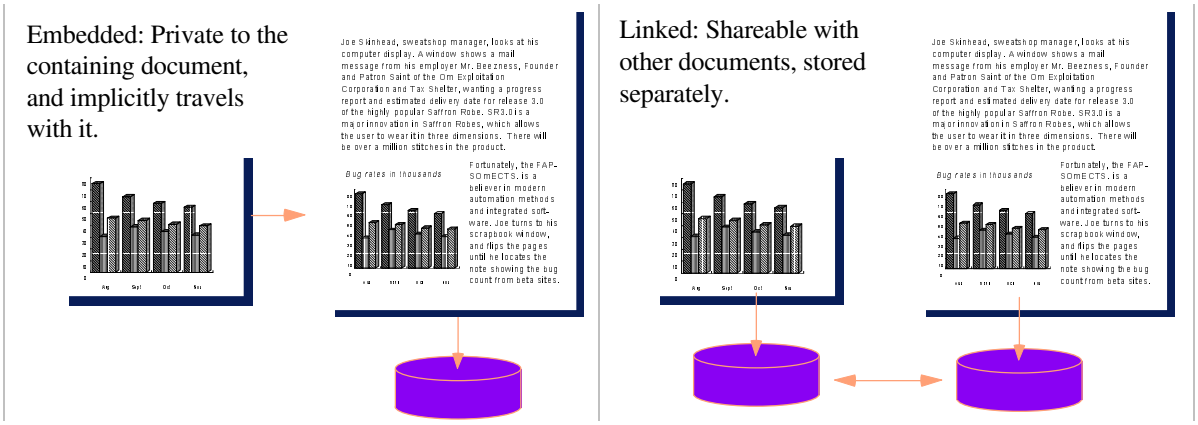

Figure 72. Embedded vs. Linked objects

Linking support in OLE 1, though it was quite functional, suffered from some technical deficiencies:

(1) The specification of the source[62] of a link was strictly a two-part (*document, item*) pair of strings. As a result, it was impossible to create a link whose source was inside an object which itself was an embedded object in another container.

(2) The name of a link source was recorded and stored in the link consumer using an absolute path name. If the source of the link moved, then the link would break, and had to manually be repaired by the user.

(3) Automatic links (formerly called "hot links") did not reconnect themselves if the link source was opened independently of the link consumer; the connection was only made if the source was opened by traversing to it from its presentation in the consumer.

(4) The OLE 1 API function OleQueryOutOfDate(), intended to indicate whether the presentation of a linked or embedded object in its container was out of date or not, could not be properly implemented. As a result, it was impossible to determine if all the links inside a document were up to date or not, leading to the need for annoying dialogs every time the user opened a document that contained links.

A concrete example will help illustrate these problems. Imagine that the OLE specification that you are now reading is instead a quarterly report for a small company. One thing that the author of this report wishes to do is to include an embedded table of sales results in the text, and also include an embedded graph whose data is linked to the table. This is shown in Figure 73.

It is not possible in OLE 1 to construct such a link: the source of the data for the chart is a range in a spreadsheet table which is itself embedded in the word processor document; this is an illustration of the first problem described above. The name for the source of such a link needs at least three parts: the name of the word processor document in the file system, the name of the spreadsheet in the word processor document, and the name of the range of interest in the spreadsheet. If it *were* possible to create this in OLE 1, then if the user were to bring up the **Edit / Links** dialog for the chart, she would see something like

---

[61] or other persistent storage location, if appropriate. A relational database, for example.

[62] In this chapter, the "source" of a link will mean the end of the link at which the data is actually stored; the "target", the "consumer", or the "receiver" of the link will refer to the other end, the location which merely contains a presentation of the information.

as is shown in Figure 74. Note the multi-part name: the word processor document has the drive and file name "C:\Q3RPT.DOC", the embedded spreadsheet table has the bookmark name "SALESTBL" in the document, and it is the range "R2C2:R7C7" in the spreadsheet that is actually being used.

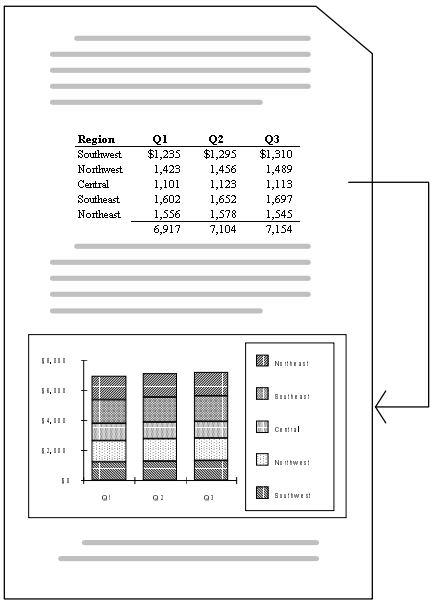

Figure 73. An embedded chart linked to a table embedded in the same container.

This example also illustrates the second problem described above: the only way in OLE 1 in which we can reference the name of the link source is with an absolute path name. Here, we use the path name "C:\Q3RPT.DOC". If we were to move the source word processing document elsewhere in the file system, perhaps to the "D:" drive, perhaps to the subdirectory "C:\QUARTERS", or whatever, then the link inside the chart will break, since it still contains the old absolute path. In OLE 1, we do absolutely no form of link source tracking at all.

The third problem described above is a little more subtle. Suppose for the moment that OLE 1 did not have the two-part name limitation and therefore we could construct this scenario. Imagine that we opened up Q3RPT.DOC, then opened up the embedded spreadsheet within it and started making changes to the data that it contains. Recall from Figure 74 that the chart has an automatic link to this data; one would expect therefore that the chart would automatically update itself as we changed the data. In OLE 1, however, this does not happen; automatic link connections can only be established from the link consumer end, not from the link source end. In this situation, the user would have to actually open the chart in order to establish the connection.

The fourth problem is perhaps a little more straightforward: when opening a document that contains a link there is simply no way to know whether the presentation in the link is up-to-date or not.

Figure 74. OLE 1 Edit / Links dialog modified to illustrate a multi-part link source.

OLE 2 provides the infrastructure by which these problems can be addressed. Central to this infrastructure is a new referencing mechanism known as a *moniker.*[63] A moniker is a conceptual handle to or name of the object at the source of the link which can be persistently stored in the link consumer. It acts much like a pointer in programming languages: at a later time, the link consumer can *bind* ("dereference") the mon-

[63] "**Monicker *n*** (Variations: **moniker** or **monniker** or **monacer** or **monica** or **monaker**) *fr middle 1800s British hoboes* A person's name, nickname, alias, etc.; =HANDLE: *His* "*monica*" *was Skysail Jack*—Jack London / *Ricord picked up a new moniker among US narcotics agents*—Time [origin unknown and very broadly speculated upon; perhaps fr transference fr earlier sense, "guinea, sovereign," when used by hoboes as an identifying mark; perhaps related to the fact that early-19th-century British tramps referred to themselves as "in the monkery," that monks and nuns take a new name when they take their vows, and *monaco* means "monk" in Italian; perhaps, as many believe, an alteration of *monogram*]"

The New Dictionary of American Slang, Edited by Robert L. Chapman, Ph.D.

iker in order to load the object to which it refers and get a real memory pointer to it. The binding process may be complex: it may involve starting one or more servers, loading objects from files, etc. In particular, some kinds of monikers will cause one or more levels of embedded object to be activated in order to reach the data, and other kinds will track data as it moves about the file system. However, all this is hidden from the caller; a moniker is *opaque* as far as it is concerned: the moniker encapsulates whatever needs to be done in order to get back to the appropriate object. In a loose analogy, monikers provide the same sort of abstraction and encapsulation for information references and binding as OLE 1 did for file formats and presentation rendering.

The rest of this chapter is organized as follows. First, we present monikers in detail. Here we discuss the conceptual model of what a moniker is, the interface that clients of monikers will use, as well as the semantics of some important OLE-provided implementations of the moniker interface. We also illustrate in some detail important parts of these implementations, since these provide a concrete understanding of how the overall process of binding to a moniker actually operates. The section following deals with the OLE 2 link objects. It presents the interfaces which are specific to links and discusses link implementations of the general compound document interfaces defined in earlier chapters. It also describes how during the binding process we avoid loading an object a second time if it is already running.

7.1. Moniker Synopsis

A moniker is simply an object that supports the IMoniker interface. IMoniker interface includes the IPersistStream interface; thus, monikers can be saved to and loaded from streams. The persistent form of a moniker contains the class identifier (CLSID) of its implementation which is used during the loading process, and so new kinds of monikers can be created transparently to clients.

The most basic operation in IMoniker interface is that of *binding* to the object to which it points, which is supported by IMoniker::BindToObject(). This function takes as a parameter the interface identifier by which the caller wishes to talk to the object, runs whatever algorithm is necessary in order to locate the object, then returns a pointer of that interface type to the caller.[64] Each moniker class can store arbitrary data its persistent representation, and can run arbitrary code at binding time.

If there is an identifiable piece of persistent storage in which the object referenced by the moniker is stored, then IMoniker::BindToStorage() can be used to gain access to it. Many objects have such identifiable storage (all OLE embedded objects do, for example), but some, such as the objects which are the ranges on a Microsoft Excel spreadsheet do not. (These ranges exist only as a part of Excel's data structures; they are in effect a figment of Excel's imagination and are only reified[65] on demand for clients.)

In most cases, a particular moniker class is designed to be one step along the path to the information source in question. These pieces can be *composed* together to form a moniker which represents the complete path. For example, the moniker stored inside the chart of Figure 73 might be a composite moniker formed from three pieces:

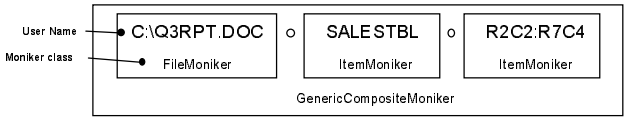

Figure 75. Moniker in link of Figure 73.

This composite is *itself* a moniker; it just happens to be a moniker which is a sequenced collection of other monikers. The composition here is *generic* in that it has no knowledge of the pieces involved *other* than that they are monikers.

Most monikers have a textual representation which is meaningful to the user; this can be retrieved with IMoniker::GetDisplayName(). The API function MkParseDisplayName() goes the other direction: it can

[64] This function also takes some parameters that provide contextual information to the binding process which we shall get to in a moment.
[65] Yes, it's a word; look it up.

turn a textual display name into the appropriate moniker, though beware that in general this is operation is as expensive as actually binding to the object.

Monikers can compare themselves to other monikers using IMoniker::IsEqual(). A hash value useful for storing monikers in lookup tables is available through IMoniker::Hash(). Monikers are not a total order or even a partial order; therefore, monikers cannot be stored in tables that rely on sorting for retrieval; use hashing instead (it is inappropriate to use the display name of a moniker for sorting, since the display name may not reflect the totality of internal state of the moniker).

The earliest time after which the object to which the moniker points is known not to have changed can be obtained with IMoniker::GetTimeOfLastChange(). This is *not* necessarily the time of last change of the object; rather, it is the best cheaply available approximation thereto.

A moniker can be asked to re-write itself into another equivalent moniker by calling IMoniker::Reduce(). This function returns a new moniker that will bind to the same object, but does so in a more efficient way. This capability has several uses:

- It enables the construction of user-defined macros or aliases as new kinds of moniker classes. When reduced, the moniker to which the macro evaluates is returned.
- It enables the construction of a kind of moniker which tracks data as it moves about. When reduced, the moniker of the data in its current location is returned.
- On file systems such as Macintosh System 7 which support an ID-based method of accessing files which is independent of file names, a File Moniker could be reduced to a moniker which contains one of these IDs.

Figure 76 shows a (somewhat contrived) example of moniker reduction. It illustrates the reduction of a moniker which names the net income entry for this year's report in the "Projects" directory of the current user's home directory.

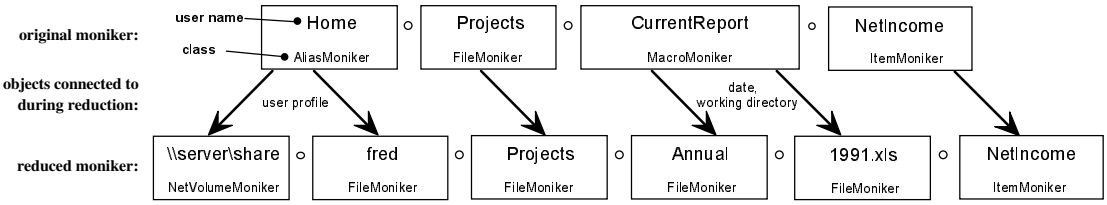

Figure 76. Reduction of a moniker showing the objects connected to during reduction.

(Note that the particular classes of monikers used here are for illustrative purposes only.) As we can see, many monikers in this example are reduced to something completely different, and some bind to something during their reduction, but some do not. For example, to reduce the alias "Home", the reduction must access the information that "Home" was an alias for "\\server\share\fred".

The process of moniker reduction could be completely hidden in the implementations of IMoniker::BindToObject() were it not for the need to do auto-link reconnections (which was mentioned as problem (3) above). We intend to support this functionality through the use of two global tables, the *Alert Object Table* and the *Running Object Table.* The Alert Object Table[66] contains an identification of those link consumers who are awaiting the appearance of their link source, along with an identification of the triggering source; the identification used for each is a moniker. The Running Object Table serves two roles. First, it serves as the place where monikers in the process of binding look to see if they are already running or not. Secondly, when an object is placed in the Running Object Table, the moniker by which it is registered is compared against the set of trigger monikers in the Alert Object Table to see if there is anything awaiting its appearance. This requires that a link source when opened announce itself using exactly the same identifying moniker as those link consumers which await its presence have indicated interest. Thus, the moniker in the link consumer must ahead of time be reduced to the form that the link source knows as its identity.

[66] Due to constraints imposed by implementation resources, the Alert Object Table is not implemented in this release of OLE. However, in the interests of expository completeness (and, frankly, to save the editor's tired fingers), the present chapter is written as if it were. We trust readers will not be too confused.

Pointers to instances of IMoniker interface can be marshalled to other processes, just as any other interface pointer can. Many monikers are of the nature that they are immutable once created and that they maintain no object state outside themselves. Item Monikers are an example of a class of such monikers. These monikers, which can be replicated at will, will usually want to support custom marshalling (see IMarshal interface) so as to simply serialize themselves and deserialize themselves in the destination context (see IPersistStream regarding serialization).

7.2. IMoniker interface and OLE IMoniker Implementations

This section describes the details of IMoniker interface and related interfaces. In addition, it discusses the various kinds of monikers that are provide as part of OLE 2.

Some moniker errors have associated with them some extended information. See IBindCtxt::RegisterObjectParam() for more details.

7.2.1. IMoniker interface

We'll now look in detail at IMoniker interface its supporting functions and structures.

```
interface IMoniker : IPersistStream[67] {
    virtual    HRESULT    BindToObject(pbc, pmkToLeft, iidResult, ppvResult) = 0;
    virtual    HRESULT    BindToStorage(pbc, pmkToLeft, iid, ppvObj) = 0;
    virtual    HRESULT    Reduce(pbc, dwReduceHowFar, ppmkToLeft, ppmkReduced) = 0;
    virtual    HRESULT    ComposeWith(pmkRight, fOnlyIfNotGeneric, ppmkComposite)
    virtual    HRESULT    Enum(fForward, ppenmMoniker) = 0;
    virtual    HRESULT    IsEqual(pmkOtherMoniker) = 0;
    virtual    HRESULT    Hash(pdwHash) = 0;
    virtual    HRESULT    IsRunning(pbc, pmkToLeft, pmkNewlyRunning) = 0;
    virtual    HRESULT    GetTimeOfLastChange(pbc, pmkToLeft, pfiletime) = 0;
    virtual    HRESULT    Inverse(ppmk) = 0;
    virtual    HRESULT    CommonPrefixWith(pmkOther, ppmkPrefix) = 0;
    virtual    HRESULT    RelativePathTo(pmkOther, ppmkRelPath);
    virtual    HRESULT    GetDisplayName(pbc, pmkToLeft, lplpszDisplayName) = 0;
    virtual    HRESULT    ParseDisplayName(pbc, pmkToLeft, lpszDisplayName, pchEaten, ppmkOut) = 0;
    virtual    HRESULT    IsSystemMoniker(pdwMksys);
    };

HRESULT BindMoniker(pmk, reserved, iidResult, ppvResult);
HRESULT CreateBindCtx(reserved, ppbc);

HRESULT MkParseDisplayName(pbc, lpszDisplayName, pchEaten, ppmk);
interface IParseDisplayName : IUnknown {
    virtual    HRESULT    ParseDisplayName(pbc, lpszDisplayName, pchEaten, ppmkOut) = 0;
    };

HRESULT CreateGenericComposite(pmkFirst, pmkRest, ppmkComposite);
HRESULT CreateFileMoniker(lpszPathName, ppmk);
HRESULT CreateItemMoniker(lpszDelim, lpszItem, ppmk);
HRESULT CreateAntiMoniker(ppmk);
HRESULT CreatePointerMoniker(punk, ppmk);

HRESULT MonikerRelativePathTo(pmkSrc, pmkDest, ppmkRelPath, reserved);
HRESULT MonikerCommonPrefixWith(pmkThis, pmkOther, ppmkPrefix);
```

[67] Notice that IMoniker interface derives from IPersistStream. As discussed in architectural overview chapter, this is an acceptable use of inheritance within the Component Object Model (in contrast with *implementation* inheritance). However, if the author of IMoniker interface had the chance to "do it over again," he would not *inherit* from IPersistStream; rather, he would mandate simply that IMoniker implementors must also implement IPersistStream accessible via QueryInterface().

7.2.1.1. IMoniker::BindToObject

HRESULT IMoniker::BindToObject(pbc, pmkToLeft, iidResult, ppvResult)

This is the workhorse function in IMoniker interface. Locate and load the object semantically referred to by this moniker according to the interface indicated by iidResult and return the object through ppvResult. After this call has returned, the semantics of the returned interface, whatever they are, should be fully functional.

In general, each kind of moniker is designed to be used as one piece in a composite which gives the complete path to the object in question. In this composite, any given piece has a certain prefix of the composite to its left, and a certain suffix to its right. If BindToObject() is invoked on the given piece, then most often the implementation of BindToObject() will require certain services of the object indicated by the prefix to its left. Item monikers, for example, require IOleItemContainer interface of the object to their left; see below. The Item Moniker implementation of BindToObject() recursively calls pmkToLeft->BindToObject() in order to obtain this interface. Other implementations of BindToObject() might instead invoke pmkToLeft->BindToStorage() if they need access not to the object itself, but to its persistent storage.

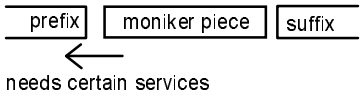

Figure 77. Interface calculus of moniker pieces

In situations where the caller of BindToObject() does not have a moniker for the object on the left, but instead has the object itself, a Pointer Moniker can be used to wrap the object pointer so that the moniker may be bound.

In situations where the moniker in fact does *not* need services of the moniker to its left, yet one is provided by the caller nevertheless, *no* error should occur; the moniker should simply ignore the needless moniker to its left.

If the object indicated by the moniker does not exist, then the error MK_E_NOOBJECT is returned.

In general, binding a moniker can be quite a complicated process, since it may need to launch servers, open files, etc. This often may involve binding to other objects, and it is often the case that binding pieces of the composite to the right of the present piece will require the same other objects. In order to avoid loading the object, releasing it, then having it loaded again later, BindToObject() can use the *bind context* passed through the pbc parameter in order to defer releasing the object until the binding process overall is complete. See IBindCtx::RegisterObjectBound() for details.

The bind context also contains a deadline time by which the caller would like the binding process to complete, or fail with the error MK_E_EXCEEDEDDEADLINE if it cannot. This capability is not often used with BindToObject(); it is more often used with other IMoniker functions such as GetTimeOfLastChange(). Nevertheless, BindToObject() implementations should (heuristically) honour the request. See IBindCtx::-GetBindOptions() for details.

Usually, for most monikers, binding a second time will return the same running object as binding the first time, rather than reloading it again from passive backing store. This functionality is supported with the Running Object Table, which is described in detail later in this chapter. Basically, the Running Object Table is a lookup table keyed by a moniker whose values are pointers to the corresponding now-running object. As objects become running, the register themselves in this table. Implementations of BindToObject() can use this table to shortcut the binding process if the object to which they point is already running. More precisely, if the passed pmkToLeft parameter is NULL (and this is not an error; that is, the moniker does not *require* something to its left), then the moniker should fully reduce itself, then look itself up in the Running Object Table and simply return the pointer to the object found there. If the pmkToLeft parameter is non-NULL, then it is the responsibility of the caller to handle this situation; the BindToOb-

ject() implementation should *not* consult the Running Object Table.[68] The Running Object Table is accessible from the bind context using IBindCtx::GetRunningObjectTable().

Argument	Type	Description
pbc	IBindCtx*	the bind context to be used for this binding operation.
pmkToLeft	IMoniker*	the moniker of the object to the left of this moniker.
iidResult	IID	the interface by which the caller wishes to connect to the object.
ppvResult	void**	on successful return, a pointer to the instantiated object is placed here, unless BINDFLAGS_JUSTTESTEXISTENCE was specified in the binding options, in which case NULL *may* be returned instead.
return value	HRESULT	S_OK, MK_E_NOOBJECT, STG_E_ACCESSDENIED, MK_E_EXCEEDED-DEADLINE, MK_E_CONNECTMANUALLY, MK_E_INTERMEDIATEINTER-FACENOTSUPPORTED, E_OUTOFMEMORY, E_NOINTERFACE

7.2.1.2. BindMoniker

HRESULT BindMoniker(pmk, reserved, iidResult, ppvResult)

Bind a moniker with the specified interface and return the result. This is strictly a helper function in that it uses no functionality which is not also available publicly. It has roughly the following implementation:

```
IBindCtx pbc;
CreateBindCtx(0, &pbc);
pmk->BindToObject(pbc, NULL, iidResult, ppvResult);
pbc->Release();
```

Argument	Type	Description
pmk	IMoniker *	the moniker which is to be bound.
reserved	DWORD	reserved for future use; must be zero.
iidResult	IID	the interface by which the caller wishes to connect to the object.
ppvResult	void **	on successful return, a pointer to the resulting object is placed here.
return value	HRESULT	S_OK, union of IMoniker::BindToObject() & CreateBindCtx() errors

7.2.1.3. IMoniker::BindToStorage

HRESULT IMoniker::BindToStorage(pbc, pmkToLeft, iid, ppvObj)

Return access to the persistent *storage* of the receiver using the given interface, rather than access to the object itself, which is what IMoniker::BindToObject() returns. Consider, for example, a moniker which refers to spreadsheet embedded in a word processing document, such as:

$$[\text{c:\foo\bar.doc}]_{\text{File Moniker}} \circ [\text{summaryTable}]_{\text{Item Moniker}}$$

Calling IMoniker::BindToObject() on this composite will enable us to talk to the spreadsheet; calling IMoniker::BindToStorage() will let us to talk to the IStorage instance in which it resides.

IMoniker::BindToStorage() will most often be called during the right-to-left recursive process of IMoniker::BindToObject() invoked on a Generic Composite Moniker. Sometimes it is the case that monikers in the tail of the composite don't require access to the object on their left; they merely require access to its persistent storage.[69] In effect, these monikers can be bound to without also binding to the objects of the monikers to their left, potentially a much more efficient operation.

Some objects do not have an independently identifiable piece of storage. These sorts of objects are really only a object-veneer on the internal state of their container. Examples include named cell ranges inside an Excel worksheet, and fragments of a Windows Word document delimited by bookmarks. Attempting to

---

[68] The reason behind this rule lies in the fact that in order to look in the Running Object Table, we need the whole moniker in its fully reduced form. If the current moniker is but a piece of a generic composite, then it has to be the composite's responsibility for doing the reduction; the moniker cannot do it correctly do it by itself.

[69] OLE2 embeddings are never of this nature, since an object may be running only if its container is also running.

call IMoniker::BindToStorage() on a moniker which indicates one of these kinds of objects will fail with the error MK_E_NOSTORAGE.

Use of the bind context in BindToStorage() is the same as in BindToObject().

Argument	Type	Description
pbc	IBindCtx*	the binding context for this binding operation.
iid	IID	the interface by which we wish to bind to this storage. Common interfaces passed here include IStorage, IStream, and ILockBytes.
ppvObj	void **	On successful return, a pointer to the instantiated storage is placed here, unless BINDFLAGS_JUSTTESTEXISTENCE was specified in the binding options, in which case NULL *may* be returned instead.
return value	HRESULT	S_OK, MK_E_NOSTORAGE, MK_E_EXCEEDEDDEADLINE, MK_E_CONNECTMANUALLY, E_NOINTERFACE, MK_E_INTERMEDIATEINTERFACENOTSUPPORTED, STG_E_ACCESSDENIED

7.2.1.4. IMoniker::Reduce

HRESULT IMoniker::Reduce(pbc, dwReduceHowFar, ppmkToLeft, ppmkReduced)

The reduction of monikers was overviewed and illustrated in the synopsis above; this is the function that actually carries it out. Return a more efficient or equally efficient moniker that refers to the same object as does this moniker. Many monikers, if not most, will simply reduce to themselves, since they cannot be re-written any further. A moniker which reduces to itself indicates this by returning itself through ppmkReduced and the returning status code MK_S_REDUCED_TO_SELF. A moniker which reduces to nothing should return NULL, and should return the status code S_OK.

If the moniker does not reduce to itself, then this function does *not* reduce this moniker in-place; instead, it returns a *new* moniker.

The reduction of a moniker which is a composite of other monikers repeatedly reduces the pieces of which it is composed until they all reduce to themselves, then returns the composite of the reduced pieces. dwReduceHowFar controls the stopping point of the reduction process. It controls to what extent the reduction should be carried out. It has the following legal values.

```
typedef enum tagMKRREDUCE {
    MKRREDUCE_ONE              = 3<<16,
    MKRREDUCE_TOUSER           = 2<<16,
    MKRREDUCE_THROUGUSER       = 1<<16,
    MKRREDUCE_ALL              = 0
    } MKRREDUCE;
```

These values have the following semantics.

Value	Description
MKRREDUCE_ONE	Perform only one step of reduction on this moniker. In general, the caller will have to have specific knowledge as to the particular kind of moniker in question in order to be able to usefully take advantage of this option.
MKRREDUCE_TOUSER	Reduce this moniker to the first point where it first is of the form where it represents something that the user conceptualizes as being the identity of a persistent object. For example, a file name would qualify, but a macro or an alias would not. If no such point exists, then this option should be treated as MKRREDUCE_ALL.
MKRREDUCE_THROUGUSER	Reduce this moniker to the point where any further reduction would reduce it to a form which the user does not conceptualize as being the identity of a persistent object. Often, this is the same stage as MKRREDUCE_TOUSER.
MKRREDUCE_ALL	Reduce the entire moniker, then, if needed reduce it again and again to the point where it reduces to simply itself.

When determining whether they have reduced themselves as far as requested, IMoniker::Reduce() implementations should not compare for equality against dwReduceHowFar, as we wish to allow for the possibility that intermediate levels of reduction will be introduced in the future. Instead, Reduce() implementations should reduce themselves *at least* as far as is requested.

An important concept in the above is the idea of a moniker that the user thinks of as the name of a persistent object; a persistent identity. The intent is to provide the ability to programmatically reduce a moniker to canonical forms whose display names would be recognizable to the user. Paths in the file system, bookmarks in word-processing documents, and range names in spreadsheets are all examples of user-identities. In contrast, neither a macro nor an alias encapsulated in a moniker, nor an inode-like file ID moniker are such identities.

The bind context parameter is used as in IMoniker::BindToObject(). In particular, implementations of IMoniker::Reduce() should pay attention to the time deadline imposed by the caller and the reporting of the moniker of the object that, if it had been running, would have allowed the reduction to progress further. See IBindCtx below.

Argument	Type	Description
pbc	IBindCtx*	The bind context to use in this operation.
dwReduceHowFar	DWORD	Indicates to what degree this moniker should be reduced; see above.
ppmkToLeft	IMoniker**	On entry, the moniker which is the prefix of this one in the composite in which it is found. On exit, the pointer is either NULL or non-NULL. Non-NULL indicates that what was previously thought of as the prefix should be disregarded and the moniker returned through ppmkToLeft considered the prefix in its place (this is very rare). NULL indicates that the prefix should not be so replaced. Thus, most monikers will NULL out this parameter before returning.
ppmkReduced	IMoniker**	On exit, the reduced form of this moniker. Possibly NULL.
return value	HRESULT	S_OK, MK_S_REDUCED_TO_SELF, MK_E_EXCEEDEDDEADLINE.

7.2.1.5. IMoniker::ComposeWith

HRESULT IMoniker::ComposeWith(pmkRight, fOnlyIfNotGeneric, ppmkComposite)

Return a new moniker which is a composite formed with this moniker on the left and pmkRight on the right. It is using this operation that the pieces of the path to an object are cobbled together to form the overall full path.

There are two distinct kinds of composite monikers: those that know nothing about their pieces other than that they are monikers, and those that know more. We have been terming the former a *generic* composite; we have seen several examples above. An example of the latter might be that of the result of composing a File Moniker containing a relative path on to the end of another File Moniker: the result could be a new File Moniker containing the complete path.[70] There is only a need for one implementation of a Generic Composite Moniker, and this has been provided; see CreateGenericComposite(). Non-generic composition is useful for monikers that are capable of collapsing a path within a storage domain to a more efficient representation in a subsequent Reduce() operation. None of the monikers provided with OLE 2 are capable of this, but an implementation of File Moniker which could collapse down to a inode-like file ID might be an example.

Each moniker class in general will have a (possibly empty) set of other kinds of special monikers that can be composed onto the end of it in a non-generic way; the moniker class has some sort of intimate knowledge about the semantics of these special monikers, more than simply that they are monikers. Each ComposeWith() implementation will examine pmkRight to see if it is such a special moniker for this imple-

[70] In fact, the current implementation of File Monikers does have this behaviour. An alternative to the non-generic composition implementation described here is that the elements in a path are each separate monikers which are then *generically* composed together.

mentation. Often, it will ask pmkRight for its class, but other possibilities exist, such as using QueryInterface(). A very common case of such special monikers are Anti Monikers.

If pmkRight is special, then the ComposeWith() implementation does whatever is appropriate for that special case. If it is not, then fOnlyIfNotGeneric controls what should occur. If fOnlyIfNotGeneric is true, then NULL should be passed back through ppmkComposite and the status MK_E_NEEDGENERIC returned; if fOnlyIfNotGeneric is false, then a generic composite should be returned using CreateGenericComposite(). Most callers of ComposeWith() should set fOnlyIfNotGeneric to false.[71]

In any situation that pmkRight completely annihilates the receiver (i.e.: irrespective of fOnlyIfNotGeneric), and so the resulting composite is empty, NULL should be passed back through ppmkComposite and the status S_OK returned.

The pieces of a moniker that have been composed together can be picked apart using IMoniker::Enum(). On a generic composite, this enumerates the monikers contained within it. On other monikers, which particular pieces are returned is implementation-defined.

Composition of monikers is an associative operation. That is, if A, B, and C are monikers, then

$(A \circ B) \circ C$

is always equal to

$A \circ (B \circ C)$

where $\circ$ represents the composition operation. Each implementation of IMoniker::ComposeWith() must maintain this invariant.

Argument	Type	Description
pmkRight	IMoniker*	the moniker to compose onto the end of the receiver.
fOnlyIfNotGeneric	BOOL	controls what should be done in the case that the way for form a composite is to use a generic one.
ppmkComposite	IMoniker*	on exit, the resulting composite moniker. Possibly NULL.
return value	HRESULT	S_OK, MK_E_NEEDGENERIC

7.2.1.6. IMoniker::Enum

HRESULT IMoniker::Enum(fForward, ppenmMoniker)

Enumerate the monikers of which the receiver is logically a composite. On a generic composite, this enumerates the pieces of which the composite is composed. On other monikers, the semantics of the pieces of which it is a composite are implementation-defined. For example, enumerating the pieces of a File Moniker might pick apart the internally stored path name into its components, even though they are not stored internally as actual separate monikers.[72] Many monikers have no discernible internal structure; they will simply pass back NULL instead of an enumerator.

IEnumMoniker is an enumerator that supports the enumeration of items which are monikers. It is defined as:

```
typedef Enum<IMoniker*> IEnumMoniker;
```

which is shorthand for

```
interface IEnumMoniker : IUnknown {
    virtual    HRESULT    Next(ULONG celt, IMoniker* rgelt[], ULONG* pceltFetched) = 0;
    virtual    HRESULT    Skip(ULONG celt) = 0;
    virtual    HRESULT    Reset() = 0;
    virtual    HRESULT    Clone(IEnumMoniker** ppenm) = 0;
    };
```

See the section in Chapter 4 on “Enumerators” for information on how enumerators should be used, and the description of the semantics of these methods.

---

[71] fOnlyIfNotGeneric is set by recursive ComposeWith() calls from the implementation of Generic Composite Moniker - ComposeWith().
[72] Could, that is, if File Monikers internally used non-generic composition. Currently, they do not.

Argument	Type	Description
fForward	BOOL	If true, then the enumeration should be done in the normal order. If false, then the order should be the reverse of the order enumerated by the normal order.
ppenmMoniker	IEnumMoniker**	On exit, the returned enumerator. May be NULL, signifying that there is nothing to enumerate.
return value	HRESULT	S_OK.

7.2.1.7. IMoniker::IsEqual

HRESULT IMoniker::IsEqual(pmkOtherMoniker)

The most important use of this function is in the implementation of the Running Object Table. As discussed in detail later, this table has two distinct but closely related rôles. First, using a moniker, entries in the Running Object Table indicate those objects which are presently now logically running and to which auto-link reconnections should be made. Second, for those objects which are actually running (have an object pointer), it provides a means given their moniker to actually connect to the appropriate running object.

The moniker implementation should *not* reduce itself before carrying out the compare operation.

Two monikers which can compare as equal in either order must hash to the same value; see IMoniker::-Hash().

Argument	Type	Description
pmkOtherMoniker	IMoniker*	the other moniker with whom we would like to compare the receiver.
return value	HRESULT	S_OK, S_FALSE

7.2.1.8. IMoniker::Hash

HRESULT IMoniker::Hash(pdwHash)

Return a 32 bit integer associated with this moniker. This integer is useful for maintaining tables of monikers: the moniker can be hashed to determine a hash bucket in the table, then compared with IsEqual() against all the monikers presently in that hash bucket.

It must always be the case that two monikers that compare as equal in either order hash to the same value. In effect, implementations of IsEqual() and Hash() are intimate with one another; they must always be written together.

The value returned by IMoniker::Hash() is invariant under marshalling: if a moniker is marshalled to a new context, then Hash() invoked on the unmarshalled moniker in the new context must return the same value as Hash() invoked on the original moniker. This is the only way that a global table of monikers such as the Running Object Table can be maintained in shared space, yet accessed from many processes. The obvious implementation technique this indicates is that Hash() should not rely on the memory address of the moniker, but only its internal state.

Argument	Type	Description
pdwHash	DWORD *	the place in which to put the returned hash value.
return value	HRESULT	S_OK

7.2.1.9. IMoniker::IsRunning

HRESULT IMoniker::IsRunning(pbc, pmkToLeft, pmkNewlyRunning)

Answer as to whether this moniker is in fact running. As usual, the Running Object Table in whose context this question is to be answered is obtained by this moniker from the Bind context. pmkToLeft is the moniker to the left of this object in the generic composite in which it is found, if any.

If non-NULL, pmkNewlyRunning is the moniker which has most recently been added to the Running Object Table; the implementation of IsRunning() can assume that without this object in the R.O.T., that IsRunning() would have reported that it was not running; thus, the only way that it can now be running is if this newly running moniker is in fact itself! This allows for some n^2-to-n reductions in algorthims that use monikers. (If the moniker implementation chose to ignore pmkNewlyRunning, no harm would come: this moniker is in fact in the R.O.T.)

Implementations of this method in various kinds of moniker classes are roughly as follows:

Generic Composite Moniker

```
if (pmkToLeft != NULL)
    return (pmkToLeft->ComposeWith(this)) -> IsRunning(pbc, NULL, pmkNewlyRunning);
if (pmkNewlyRunning != NULL) {
    if (pmkNewlyRunning -> IsEqual(this) == NOERROR)
        return NOERROR;
    }
else if (pRunningObjectTable -> IsRunning(this) == NOERROR)
    return NOERROR;
// otherwise, forward it on to my last element.
return this->Last()->IsRunning(pbc, this->AllButLast(), pmkNewlyRunning)
```

Any moniker whose class does not do any wildcard matching

```
if (pmkToLeft == NULL) {
    if (pmkNewlyRunning != NULL)
        return pmkNewlyRunning -> IsEqual(this);
    else
        return pRunningObjectTable -> IsRunning(this);
    }
else
    return ResultFromScode(S_FALSE);    // If I was running, then Generic Composite would have caught it.
```

A moniker class which has a wild card entry which always matches any instance of the moniker class: if the wild card is present, then all instances of the moniker class to the right of the same other moniker (that is, with the same moniker to their left) are deemed to be running. Such a moniker class might be reasonably used, for example, to match all the addressable ranges in a given spreadsheet.

```
if (pmkToLeft == NULL) {
    if (pmkNewlyRunning != NULL)
        return pmkNewlyRunning->IsEqual(this) == NOERROR
            || pmkNewlyRunning->IsEqual(my wild card moniker) == NOERROR;
    if (pRunningObjectTable -> IsRunning(this) == NOERROR)
        return NOERROR;
    return pRunningObjectTable -> IsRunning(my wild card moniker);
    }
else
    return pmkToLeft->ComposeWith(my wild card moniker) -> IsRunning(pbc, NULL, pmkNewlyRunning);
```

A moniker class which has a wild card entry which matches against some of the objects, but only the ones which are in fact actually currently running. We illustrate here specifically the behaviour of Item Monikers.

```
if (pmkToLeft == NULL) {
    if (pmkNewlyRunning != NULL) {
        if (pmkNewlyRunning->IsEqual(this) == NOERROR)
            return NOERROR;
        if (pmkNewlyRunning->IsEqual(my wild card moniker) != NOERROR)
            return ResultFromScode(S_FALSE);
        goto TestBind;
        }
    }
if (pmkToLeft->ComposeWith(my wild card moniker)->IsRunning(pbc, NULL, pmkNewlyRunning) != NOERROR)
    return ResultFromScode(S_FALSE);
TestBind:
// In general, connect to the container and ask whether the object is running. The use of
// IOleItemContainer here is Item Moniker-specific, but the theme is a general one.
IOleItemContainer *pcont;
pmkToLeft->BindToObject(pbc, NULL, IID_IOleItemContainer, &pcont);
return pcont->IsRunning(szItemString);
```

The arguments to this function are as follows:

Argument	Type	Description
pbc	IBindCtx*	the usual bind context
pmkToLeft	IMoniker*	the moniker to the left of this one in the composite in which it is found.
pmkNewlyRunning	IMoniker*	may be NULL. If non-NULL, then this is the moniker which has been most recently added to the R.O.T. In this case, IMoniker::IsRunning() implementations may assume that without this moniker in the R.O.T. that IsRunning() would return S_FALSE.
return value	HRESULT	S_OK, S_FALSE

7.2.1.10. IMoniker::GetTimeOfLastChange

HRESULT IMoniker::GetTimeOfLastChange(pbc, pmkToLeft, pfiletime)

Answer the earliest time after which the object pointed to by this moniker is known not to have changed.

The purpose of this function is to support the ability to determine whether a given OLE link object or OLE embedded object which contains links is up-to-date or not. This is usually done as user documents are opened; thus, in most cases it will be important that this operation is fast. Implementations should pay particular attention to the deadline parameter passed in the bind context.

If it is not the case that all the objects in a document are known to be up-to-date, the user will usually be prompted with a dialog as to whether they should be updated. If he says yes, then each of the objects which is not known to be up-to-date will be bound to in order to retrieve a new presentation. The point germane to the present discussion is that GetTimeOfLastChange() is part of the mechanism of avoiding binding to objects unnecessarily. GetTimeOfLastChange() itself, therefore, should not bind to the object in order to obtain the most accurate answer. Rather, it should return the best available answer given objects that are already running. Many monikers denote an object *contained* in the object denoted by the moniker to their left. Implementations of GetTimeOfLastChange() in most of these monikers can take advantage of the fact they cannot have changed any later than the object in which they are contained. That is, these monikers can simply forward the call onto the moniker to their left.

The returned time of change is reported using a FILETIME. A FILETIME is a 64-bit value indicating a time in units of 100 nanoseconds, with an origin in 1601 (see FILETIME elsewhere in this specification for details).[73] A resolution of 100 nanoseconds allows us to deal with very fast-changing data; allocating this many bits gives us a range of tens of thousands of years. It is not expected that most change times in objects will be actually be internally recorded with this precision and range; they only need be reported with such.

If the time of last change is unavailable, either because the deadline was exceeded or otherwise, then it is recommended that a FILETIME of {dwLowDateTime,dwHighDateTime} = {0xFFFFFFFF,0x7FFFFFFF} (note the 0x7 to avoid accidental unsigned / signed confusions) should be passed back. If the deadline was exceeded, then the status MK_E_EXCEEDEDDEADLINE should be returned. If the time of change is unavailable, and would not be available no matter what deadline were used, then MK_E_UNAVAILABLE should be returned. Otherwise, S_OK should be returned.

If pmkToLeft is NULL, then this function should generally first check for a recorded change-time in the Running Object Table with IRunningObjectTable::GetTimeOfLastChange() before proceeding with other strategies. Moniker classes that support wildcards will have to take into consideration exactly what does get put in the R.O.T. and look for the appropriate thing; since Generic Composite Monikers know nothing of wildcards, they may even need to do that in the non-NULL pmkToLeft case. See IMoniker::IsRunning().

---

[73] The definition of FILETIME was taken from the Microsoft Windows–32 specification.

Argument	Type	Description
pbc	IBindCtx*	the binding context for this operation.
pmkToLeft	IMoniker*	the moniker to the left of this one in the composite in which it is found.
pfiletime	FILETIME*	the place in which the time of last change should be reported.
return value	HRESULT	S_OK, MK_E_EXCEEDEDDEADLINE, MK_E_UNAVAILABLE, MK_E_CONNECTMANUALLY

7.2.1.11. IMoniker::Inverse

HRESULT IMoniker::Inverse(ppmk)

Answer a moniker that when composed onto the end of this moniker or one of similar structure will annihilate it; that is, will compose to nothing. IMoniker::Inverse() will be needed in implementations of IMoniker::RelativePathTo(), which are important for supporting monikers that track information as it moves about.

This is the abstract generalization of the ".." operation in traditional file systems. For example a File Moniker which represented the path "a\b\c\d" would have as its inverse a moniker containing the path "..\..\..\..", since "a\b\c\d" composed with "..\..\..\.." yields nothing.

Notice that an the inverse of a moniker does not annihilate just that particular moniker, but all monikers with a similar structure, where structure is of course interpreted with respect to the particular moniker. Thus, the inverse of a Generic Composite Moniker is the reverse composite of the inverse of its pieces. Monikers which are non-generic composites (such as File Monikers are presently implemented) will also have non-trivial inverses, as we just saw. However, there will be many kinds of moniker whose inverse is trivial: the moniker *adds* one more piece to an existing structure; its inverse is merely a moniker that *removes* the last piece of the existing structure. A moniker that when composed onto the end of a generic moniker removes the last piece is provided; see CreateAntiMoniker(). Monikers with no internal structure can return one of these as their inverse.

Not all monikers have inverses. The inverse of an anti-moniker, for example, does not exist. Neither will the inverses of most monikers which are themselves inverses. It is conceivable that other monikers do not have inverses as well; a macro moniker might be an example. Monikers which have no inverse cannot have relative paths formed from things inside the objects they denote to things outside.

Argument	Type	Description
ppmk	IMoniker**	the place to return the inverse moniker.
return value	HRESULT	S_OK, MK_E_NOINVERSE.

7.2.1.12. IMoniker::CommonPrefixWith

HRESULT IMoniker::CommonPrefixWith(pmkOther, ppmkPrefix)

Answer the longest common prefix that the receiver shares with the moniker pmkOther. This functionality is useful in constructing relative paths, and for performing some of the calculus on monikers needed by the **Edit / Links** dialog.

Argument	Type	Description
pmkOther	IMoniker*	the moniker with whom we are determine the common prefix.
ppmkPrefix	IMoniker*	the place to return the common prefix moniker. NULL is returned only in the case that the common prefix does not exist.
return value	HRESULT	MK_S_ME, indicating that the receiver as a whole is the common prefix. MK_S_HIM, indicating that pmkOther as a whole is the common prefix. MK_S_US, indicating that in fact the two monikers are equal. S_OK, indicating that the common prefix exists but is neither the

receiver nor pmkOther. MK_S_NOPREFIX indicating that no common prefix exists.

7.2.1.13. MonikerCommonPrefixWith

HRESULT MonikerCommonPrefixWith(pmkThis, pmkOther, ppmkPrefix)

This function is intended solely for the use of moniker *implementors;* clients of monikers "need not apply;" clients should instead compute the common prefix between two monikers by using

```
pmkSrc->CommonPrefixWith(pmkOther, ppmkPrefix);
```

Implementations of IMoniker::CommonPrefixWith() necessarily call MonikerCommonPrefixWith() as part of their internal processing. Such a method should first check to see if the other moniker is a type that it recognizes and handles specially. If not, it should call MonikerCommonPrefixWith(), passing itself as pmkSrc and the other moniker as pmkDest. MonikerCommonPrefixWith() will handle the generic composite cases correctly.

Argument	Type	Description
pmkThis	IMoniker *	the starting moniker for the computation of the relative path.
pmkOther	IMoniker *	the moniker to which a relative path should be taken.
ppmkPrefix	IMoniker **	May *not* be NULL. The place at which the moniker of pmkDest relative to pmkSrc is to be returned.
return value	HRESULT	S_OK, MK_S_HIM, MK_S_ME, MK_S_US, MK_S_NOPREFIX

7.2.1.14. IMoniker::RelativePathTo

HRESULT IMoniker::RelativePathTo(pmkOther, ppmkRelPath)

Answer a moniker that when composed onto the end of this one or one with a similar structure will yield pmkOther. Conceptually, implementations of this function usually work as follows: the longest prefix that the receiver and pmkOther have is common is determined. This breaks the receiver and pmkOther each into two parts, say (P,T_{me}) and (P,T_{him}) respectively, where P is the maximal common prefix. The correct relative path result is then $T_{me}^{-1} \circ T_{him}$.

For any given implementation of this function, it is usually the case that the same pmkOther monikers are treated specially as would be in IMoniker::ComposeWith(). File Monikers, for example, might treat other File Monikers specially in both cases.

See also MonikerRelativePathTo().

Argument	Type	Description
pmkOther	IMoniker*	the moniker to which a relative path should be taken.
ppmkRelPath	IMoniker*	May *not* be NULL. The place at which the relative path is returned.
return value	HRESULT	MK_S_HIM, indicating that the only form of relative path is in fact just the other moniker, pmkOther. S_OK, indicating that a non-trivial relative path exists.

7.2.1.15. MonikerRelativePathTo

HRESULT MonikerRelativePathTo(pmkSrc, pmkDest, ppmkRelPath, reserved)

This function is intended solely for the use of moniker *implementors;* clients of monikers "need not apply;" clients should instead compute the relative path between two monikers by using

```
pmkSrc->RelativePathTo(pmkDest, ppmkRelPath);
```

Implementations of IMoniker::RelativePathTo() necessarily call MonikerRelativePathTo() as part of their internal processing. Such a method should first check to see if the other moniker is a type that it recognizes and handles specially. If not, it should call MonikerRelativePathTo(), passing itself as pmkSrc and the other moniker as pmkDest. MonikerRelativePathTo() will handle the generic composite cases correctly.

Argument	Type	Description
pmkSrc	IMoniker *	the starting moniker for the computation of the relative path.
pmkDest	IMoniker *	the moniker to which a relative path should be taken.
ppmkRelPath	IMoniker **	May *not* be NULL. The place at which the moniker of pmkDest relative to pmkSrc is to be returned.
reserved	BOOL	must be *non-zero (**NOTE!**)*
return value	HRESULT	S_OK, MK_S_HIM

7.2.1.16. IMoniker::GetDisplayName

HRESULT IMoniker::GetDisplayName(pbc, pmkToLeft, lplpszDisplayName)

Most monikers have a textual representation which is meaningful to a human being. This function returns the current display name for this moniker, or NULL if none exists.

Some display names may change over time as the object to which the moniker refers moves about in the context in which it lives. Formula references between two Microsoft Excel 4.0 spreadsheets are an example of this (formula references between cells are conceptually very much like OLE links). A formula referring to cell "R1C1" in another sheet may change to the refer to "R2C1" if a new row is inserted at the top of the second sheet: the reference still refers to the same actual cell, but now the cell has a different address in its sheet. This behaviour leads to the general observation that obtaining the current display name of a moniker may have to access at least the storage of the object to which it refers, if not the object itself. Thus, it has the potential to be an expensive operation. As in other IMoniker functions, a bind context parameter is passed which includes a deadline within which the operation should complete, or fail with MK_E_EXCEEDEDDEADLINE if unable to do so.

A consequence of the possible unavailability of quick access to the display name of a moniker is that callers of this function most likely will want to cache the last successful result that they obtained, and use that if the current answer is inaccessible (this caching is the Microsoft Excel between-sheet behaviour).

In the general case, the display name of a moniker is *not* unambiguous: there may be more than one moniker with the same display name, though in practice this will be rare. There is also no *guarantee* that a display name obtained from a moniker will parse back into that moniker in MkParseDisplayName(), though failure to do so also will be rare. Display names should therefore be thought of as a merely a note or annotation on the moniker which aid a human being in distinguishing one moniker from another, rather than a completely equivalent representation of the moniker itself.

Notice that due to how display names are constructed in composites, a moniker which is a prefix of another necessarily has a display name which is a (string) prefix of the display name of the second moniker. The converse, however, does not necessarily hold.

A moniker which is designed to be used as part of a generic composite is responsible for including any preceding delimiter as part of its display name. Many such monikers take a parameter for this delimiter in their instance creation functions.

Argument	Type	Description
pbc	IBindCtx*	the bind context for this operation.
pmkToLeft	IMoniker*	the moniker to the left of this one in the composite in which it is found. Most monikers will not require this in GetDisplayName().
lplpszDisplayName	LPSTR *	on exit, the current display name for this moniker. NULL if the moniker does not have a display name or the deadline was exceeded.
return value	HRESULT	S_OK, MK_E_EXCEEDEDDEADLINE.

7.2.1.17. MkParseDisplayName

HRESULT MkParseDisplayName(pbc, lpszDisplayName, pcchEaten, ppmk)

Recall from IMoniker::GetDisplayName() that most monikers have a textual name which is meaningful to the user. The function MkParseDisplayName() does the logical inverse operation: given a string, it 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.

Argument	Type	Description
pbc	IBindCtx*	the binding context in which to accumulate bound objects.
lpszDisplayName	LPSTR	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.
ppmk	IMoniker*	the resulting moniker.
return value	HRESULT	S_OK, MK_E_SYNTAX.

In general, parsing a display name is as expensive as binding to the object that it denotes, since along the way various name space managers need to be connected to by the parsing mechanism. As is usual, this manager objects are not released by the parsing operation itself, but are instead handed over to the passed-in binding context. Thus, if the moniker resulting from the parse is immediately bound using this binding context, redundant loading of objects is maximally avoided.

Generally, parsing will be used considerably less often than one might at first glance expect. Most compound document links, for example, are created by a user doing **Copy** followed by **Paste Link** rather than by her typing in the name of the link source, and this link is created programmatically with no need for an intermediate form which is a human-readable textual representation. The primary use of MkParseDisplayName() lies instead in textual programming languages which permit remote references as syntactic elements. The expression language of a spreadsheet is a good example of such a language. MkParseDisplayName() will also be used in the implementation of the standard **Edit / Links** dialog.

The parsing process is an inductive one, in that there is an initial step that gets the process going, followed by the repeated application of an inductive step. At any point after the beginning of the parse, a certain prefix of lpszDisplayName has been parsed into a moniker, and a suffix of the display name remains not understood. This is illustrated in Figure 78.

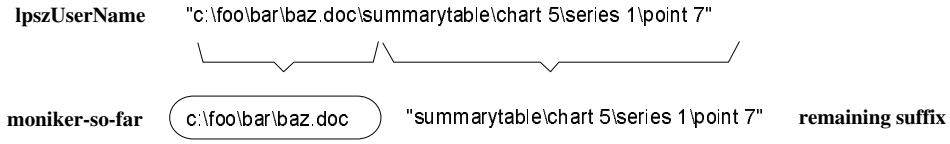

Figure 78. Intermediate stage in parsing a display name into a moniker.

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.[74] 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-

[74] None of the monikers provided with OLE 2 are of this form, but some may be created by particular container applications.

tainer. 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 (see the GetClassObject() function). 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 the OLE 2 parsing machinery 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. MkParseDisplayName() 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. Presently, in OLE 2, this syntax is fixed: parsing a legal display name must begin with an existing file name. However, said file name may be drive absolute, drive relative, working-directory relative, or begin with an explicit network share name. Further, if an existing file name is not found, the Running Object Table is searched for File Monikers which may be registered to support connections to as-yet-unsaved documents; in detail, the display name up to the first '\', '/', or '!' is consulted as a File Moniker in the table. The DDE name space is also searched, for OLE1 compatibility.

Note, however, that this inability to parse into other monikers is the *only* restriction forcing composite monikers to begin with a file moniker.

In the future, it is likely that a mechanism will be provided by which new parsers can be installed. One way this might work is to have each parser register the regular expression which recognizes its prefixes (that is, restrict ourselves to regular expression grammars). At parse time, MkParseDisplayName() would choose the language which maximally matched against the display name.

7.2.1.18. IMoniker::ParseDisplayName

HRESULT IMoniker::ParseDisplayName(pbc, pmkToLeft, lpszDisplayName, pchEaten, ppmkOut)

Given that the composite moniker (pmkToLeft ∘ (the receiver)) is the moniker which has so far been parsed, parse as much of the remaining tail as is appropriate. In general, the maximal prefix of lpszDisplayName which is syntactically valid and which currently *represents an existing object* should be consumed.

The main loop of MkParseDisplayName() finds the next piece moniker piece by calling this function 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.

lpszDisplayName is the as-yet-to-be-parsed tail of the display name. This function is to consume as much of it as is appropriate for a name within the object identified by (pmkToLeft ∘ (the receiver)) and return the corresponding moniker.

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 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.
pmkToLeft	IMoniker*	the moniker to the left of this one in the so-far-parsed display name.
lpszDisplayName	LPSTR	the display name to be parsed.
pcchEaten	ULONG*	the number of characters of the input name that this parse consumed.
ppmkOut	IMoniker*	the resulting moniker.
return value	HRESULT	S_OK, MK_E_SYNTAX.

7.2.1.19. IMoniker::IsSystemMoniker

HRESULT IMoniker::IsSystemMoniker(pdwMksys)[75]

Answer as to whether this moniker is a type of moniker whose particular implementation semantics are conceptually important to the binding process. The values returned through pdwMksys are taken from the following enumeration:

```
typedef enum tagMKSYS {
    MKSYS_NONE = 0,
    MKSYS_GENERICCOMPOSITE = 1,
    MKSYS_FILEMONIKER = 2,
    MKSYS_ANTIMONIKER = 3,
    MKSYS_ITEMMONIKER = 4,
    MKSYS_POINTERMONIKER = 5,
    } MKSYS;
```

All user implementations of this function must simply return MKSYS_NONE through pdwMksys. IMoniker-::GetClassID() (see IPersist) can be used instead by non-system monikers to check for the presence of their own "special" moniker on the right in IMoniker::ComposeWith(). Alternatively, use QueryInterface() to test for the presence of your own private interface (though this approach would also require custom marshalling).

New values of this enumeration may be defined in the future; caller's of this function should be aware of this fact and should therefore explicitly test against known return values that they care about (rather than, for example, assuming that if the returned value is not one of the values listed here then it's the other).

The returned value is *not* a bitfield value; rather it is an integer.

Argument	Type	Description
pdwMksys	DWORD*	the place at which the result is to be returned. May not be NULL.
return value	HRESULT	S_OK

[75] This function is a member of IMoniker interface rather than an independent API function in order that future revisions of this function can be correctly correlated with revisions to system moniker classes.

7.2.2. IParseDisplayName interface

7.2.2.1. IParseDisplayName::ParseDisplayName

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

This is the single function in the IParseDisplayName interface:

```
interface IParseDisplayName : IUnknown {
    virtual     HRESULT     ParseDisplayName(pbc, lpszDisplayName, pcchEaten, ppmkOut) = 0;
    };
```

Its semantics and parameters are as described in IMoniker::ParseDisplayName().

7.2.3. IBindCtx interface

The bind context parameter passed to many of the IMoniker operations serves a few purposes.

Its primary purpose is to accumulate the set of objects that get bound during an operation but which should be released when the operation is complete. This is particularly useful in generic composites: using the bind context in this way avoids binding an object, releasing it, only to have it bound again when the operation moves on to another piece of the composite.

Another purpose of the bind context is to pass a group of parameters which do not change as an operation moves from one piece of a generic composite to another. These are the *binding options*, and are described below. Some of these binding options have a related return value in certain error conditions; the bind context provides the means by which they can be returned.

The bind context is also the only means through which moniker operations should access contextual information about their environment. There should be no direct calls in moniker implementations to API functions that query or set state in the environment; all such calls should instead funnel through the bind context. Doing this allows for future enhancements which can dynamically modify binding behaviour. In OLE 2, the most important piece of contextual information that moniker operations need to access is the Running Object Table; monikers should always access this table indirectly though IBindCtx::GetRunningObjectTable(), rather than using the global function GetRunningObjectTable(). IBindCtx interface allows for future extensions to the passed-in contextual information in the form the ability to maintain a string-keyed table of objects. See IBindCtx::RegisterObjectParam() and related functions.

```
interface IBindCtx : IUnknown {
    virtual     HRESULT     RegisterObjectBound(punk) = 0;
    virtual     HRESULT     RevokeObjectBound(punk) = 0;
    virtual     HRESULT     ReleaseBoundObjects() = 0;

    virtual     HRESULT     SetBindOptions(pbindopts) = 0;
    virtual     HRESULT     GetBindOptions(pbindopts) = 0;

    virtual     HRESULT     GetRunningObjectTable(pprot) = 0;

    virtual     HRESULT     RegisterObjectParam(lpszKey, punk) = 0;
    virtual     HRESULT     GetObjectParam(lpszKey, ppunk) = 0;
    virtual     HRESULT     EnumObjectParam(ppenum) = 0;
    virtual     HRESULT     RevokeObjectParam(lpszKey) = 0;
    };

typedef struct {
    DWORD  cbStruct;                    // the size in bytes of this structure. ie: sizeof(BINDOPTS).
    DWORD  grfFlags;
    DWORD  grfMode;
    DWORD  dwTickCountDeadline;
    } BINDOPTS;

HRESULT CreateBindCtx(reserved, ppbc);
```


7.2.3.1. IBindCtx::RegisterObjectBound

HRESULT IBindCtx::RegisterObjectBound(punk)

Remember the passed object as one of the objects that has been bound during a moniker operation and which should be released when it is complete overall. Calling this function causes the binding context to create an additional reference to the passed-in object with IUnknown::AddRef(); the caller is still required to Release() its own copy of the pointer independently.

The effect of calling this function twice with the same object is cumulative, in that it will require two RevokeObjectBound() calls to completely remove the registration of the object within the binding context.

Argument	Type	Description
punk	IUnknown*	the object which is being registered as needing to be released.
return value	HRESULT	S_OK.

7.2.3.2. IBindCtx::RevokeObjectBound

HRESULT IBindCtx::RevokeObjectBound(punk)

This function undoes the effect of IBindCtx::RegisterObjectBound(): it removes the object from the set that will be released when the bind context in IBindCtx::ReleaseBoundObjects() (actually removes one occurrence of it). This function is likely to be rarely called, but is included for completeness.

Argument	Type	Description
punk	IUnknown*	the object which no longer needs to be released.
return value	HRESULT	S_OK, MK_E_NOTBOUND, E_OUTOFMEMORY

7.2.3.3. IBindCtx::ReleaseBoundObjects

HRESULT IBindCtx::ReleaseBoundObjects()

Releases all the objects currently registered with the bind context though RegisterObjectBound().

This function is (conceptually) called by the implementation of IBindCtx::Release().

Argument	Type	Description
return value	HRESULT	S_OK

7.2.3.4. IBindCtx::SetBindOptions

HRESULT IBindCtx::SetBindOptions(pbindopts)

Store in the bind context a block of parameters that will apply to later IMoniker operations using this bind context. Using block of parameters like this is just an alternative way to pass parameters to an operation. We distinguish the parameters we do for conveyance by this means because 1) they are common to most IMoniker operations, and 2) these parameters do not change as the operation moves from piece to piece of a generic composite.

Argument	Type	Description
pbindopts	BINDOPTS*	the block of parameters to set. These can later be retrieved with GetBindOptions().
return value	HRESULT	S_OK, E_OUTOFMEMORY

BINDOPTS is defined as follows:

```
typedef struct tagBINDOPTS {
    DWORD  cbStruct;                // the size in bytes of this structure. ie: sizeof(BINDOPTS).
    DWORD  grfFlags;
    DWORD  grfMode;
    DWORD  dwTickCountDeadline;
    } BINDOPTS;
```


The members of this structure have the following meanings:

Member Description

grfFlags

A group of boolean flags. Legal values that may be or'd together are the taken from the enumeration BINDFLAGS; see below. Moniker implementations should simply ignore any set-bits in this field that they do not understand (presumably because their meanings were defined in some future OLE extension).

grfMode

A group of flags that indicates the intended use that the caller has towards the object that he ultimately receives from the associated moniker binding operation. Constants for this member are taken from the STGM enumeration, described in the chapter on "Persistent Storage For Objects."

When applied to the BindToObject() operation, by far the most significant flag values are: STGM_READ, STGM_WRITE, and STGM_READWRITE. It is possible that some binding operations might make use of the other flags, particularly STGM_DELETEONRELEASE or STGM_CREATE, but such cases are quite esoteric.

When applied to the BindToStorage() operation, *most* STGM values are potentially useful here.

The default value for grfMode is STGM_READWRITE | STGM_SHARE_EXCLUSIVE.

dwTickCountDeadline

This is an indication of when the caller would like the operation to complete. Having this parameter allows the caller to approximately & heuristically bound the execution time of an operation when it is more important that the operation perform quickly than it is that it perform accurately. Most often, this capability is used with IMoniker::GetTimeOfLastChange(), as was previously described, though it can be usefully applied to other operations as well.

This 32-bit unsigned value is a time in milliseconds on the local clock maintained by the GetTickCount() function.[76] A value of zero indicates "no deadline;" callers should therefore be careful not to pass to the bind context a value of zero that was coincidentally obtained from GetTickCount(). Clock wrapping is also a problem. Thus, if the value in this variable is less than the current time by more than 2^{31} milliseconds, then it should be interpreted as indicating a time in the future of its indicated value plus 2^{32} milliseconds.

Typical deadlines will allow for a few hundred milliseconds of execution. Each function should try to complete its operation by this time on the clock, or fail with the error MK_E_EXCEEDEDDEADLINE if it cannot do so in the time allotted. Functions are not *required* to be absolutely accurate in this regard, since it is almost impossible to predict how execution might take (thus, callers cannot rely on the operation completing by the deadline), but operations which exceeded their deadline excessively will usually cause intolerable user delays in the operation of their callers. Thus, in practice, the use of deadlines is a **heuristic** which callers can impose on the execution of moniker operations.

If a moniker operation exceeds its deadline because a given object or objects that it uses are not running, and if one of these had been running, then the operation would have completed more of its execution, then the monikers of these objects should be recorded in the bind context using RegisterObjectParam() under the parameter names "ExceededDeadline", "ExceededDeadline1", "ExceededDeadline2", etc.; use the first name in this series that is currently unused. This approach gives the caller some knowledge as to when to try the operation again

The enumeration BINDFLAGS, which contains the legal values for the bitfield BINDOPTS::grfFlags, is defined as follows:

```
typedef enum tagBINDFLAGS {
    BINDFLAGS_MAYBOTHERUSER = 1,
    BINDFLAGS_JUSTTESTEXISTENCE = 2,
    } BINDFLAGS;
```

[76] This implies that if an IBindCtx is ever remoted to a separate machine that the proxy for the IBindCtx would have to perform appropriate translation to local tickcount time. In practice, however, IBindCtx will rarely if ever be remoted.

These flags have the following interpretation.

Value	Description
BINDFLAGS_MAYBOTHERUSER	If not present, then the operation to which the bind context containing this parameter is applied should not interact with the user in any way, such to ask for a password for a network volume that needs mounting. If present, then this sort of interaction is permitted. If prohibited from interacting with the user when it otherwise would like to, an operation may elect to use a different algorithm that does not require user interaction, or it may fail with the error MK_MUSTBOTHERUSER.
BINDFLAGS_JUSTTESTEXISTENCE	If present, indicates that the caller of the moniker operation to which this flag is being applied is not actually interested in having the operation carried out, but only in learning of the operation could have been carried out had this flag not been specified. For example, this flag give the caller the ability to express that he is only interested in finding out whether an object actually exists by using this flag in a BindToObject() call. Moniker implementations are free, however, to ignore this possible optimization and carry out the operation in full. Callers, therefore, need to be able to deal with both cases. See the individual routine descriptions for details of exactly what status is returned.

7.2.3.5. IBindCtx::GetBindOptions

HRESULT IBindCtx::GetBindOptions(pbindopts)

Return the current binding options stored in this bind context. See IBindCtx::SetBindOpts() for a description of the semantics of each option.

Notice that the caller provides a BINDOPTS structure, which is filled in by this routine. It is the caller's responsibility to fill in the cbStruct member correctly.

Argument	Type	Description
pbindOpts	BINDOPTS*	the structure of binding options which is to be filled in.
return value	SCODE	

7.2.3.6. IBindCtx::GetRunningObjectTable

HRESULT IBindCtx::GetRunningObjectTable(pprot)

Return access to the Running Object Table relevant to this binding process. Moniker implementations should get access to the Running Object Table using this function rather than the global API GetRunningObjectTable(). The appropriate Running Object Table is determined implicitly at the time the bind context is created.

Argument	Type	Description
pprot	IRunningObjectTable**	the place to return the running object table.
return value	SCODE	

7.2.3.7. IBindCtx::RegisterObjectParam

HRESULT IBindCtx::RegisterObjectParam(lpszKey, punk)

Register the given object pointer under the name lpszKey in the internally-maintained table of object pointers. The intent of this table is that it be used as an extensible means by which contextual information can be passed to the binding process. String keys are compared case-sensitive.

Like IBindCtx::RegisterObjectBound(), this function creates an additional reference to the passed-in object using IUnknown::AddRef(). The effect of calling this function a second time with the same lpszKey is to replace in the table the object passed-in the first time.

By convention, moniker implementors may freely use object parameters whose names begin with the string representation of the class id of the moniker implementation in question.

This facility is also used as means by which various errors can convey information back to the caller. Associated with certain error values are the following object parameters:

Error	Parameters
MK_E_EXCEEDEDDEADLINE	Parameters named "ExceededDeadline", "ExceededDeadline1", "ExceededDeadline2", etc., if they exist, are monikers who appearance as running would make it reasonable for the caller to attempt the binding operation again.
MK_E_CONNECTMANUALLY	The parameter named "ConnectManually" is a moniker whose display name should be shown to the user requesting that he manually connect it, then retry the operation. The most common reason for return this value is that a password is needed. However, it could be that a floppy needs to be mounted. The existence of this error return is a concession to OLE implementation schedules: clearly, in the best case, this stuff should be completely handled inside the moniker implementations themselves. Later versions of the OLE-provided monikers will be enhanced so that they simply don't ever return this value. Other moniker writers are discouraged from using it.
E_CLASSNOTFOUND	The parameter named "ClassNotFound", if present, is a moniker to the storage of the object whose class was not able to be loaded in the process of a moniker operation. When the moniker is being used in an OLE compound document situation, a sophisticated client may wish to BindToStorage() on this moniker, then attempt to carry out a Treat As... or Convert To... operation as described in the "Persistent Storage For Objects" chapter. If this is successful, then the binding operation could be tried again. Such a methodology improves the usability of link operations.

New moniker authors can freely use parameter names that begin with the string form of the CLSID of their moniker; see StringFromCLSID().

The arguments to this function are as follows:

Argument	Type	Description
lpszKey	LPSTR	the name under which the object is being registered.
punk	IUnknown *	the object being registered.
return value	HRESULT	S_OK, E_OUTOFMEMORY

7.2.3.8. IBindCtx::GetObjectParam

HRESULT IBindCtx::GetObjectParam(lpszKey, ppunk)

Lookup the given key in the internally-maintained table of contextual object parameters and return the corresponding object, if one exists.

Argument	Type	Description
lpszKey	LPSTR	the key under which to look for an object.
ppunk	IUnknown **	The place to return the object interface pointer. NULL is returned on failure (along with S_FALSE).
return value	HRESULT	S_OK, S_FALSE

7.2.3.9. IBindCtx::EnumObjectParam

HRESULT IBindCtx::EnumObjectParam(ppenum)

Enumerate the strings which are the keys of the internally-maintained table of contextual object parameters.

Argument	Type	Description
ppenum	IEnumString **	the place to return the string enumerator. See Chapter 4 for a description of IEnumString.
return value	HRESULT	S_OK, E_OUTOFMEMORY

7.2.3.10. IBindCtx::RevokeObjectParam

HRESULT IBindCtx::RevokeObjectParam(lpszKey)

Revoke the registration of the object currently found under this key in the internally-maintained table of contextual object parameters, if any such key is currently registered.

Argument	Type	Description
lpszKey	LPSTR	the key whose registration is to be revoked.
return value	HRESULT	S_OK, S_FALSE

7.2.3.11. CreateBindCtx

HRESULT CreateBindCtx(reserved, ppbc)

Allocate and initialize a new BindCtx using an OLE-supplied implementation.

Argument	Type	Description
reserved	DWORD	reserved for future use; must be zero.
ppbc	IBindCtx*	the place in which to put the new BindCtx.
return value	HRESULT	S_OK, E_OUTOFMEMORY

7.2.4. Generic Composite Moniker class

Recall that in general monikers are a composite list made up of other pieces. All monikers which are a generic composite of other monikers are instances of Generic Composite Moniker class whose implementation is provide with OLE; there is no need for two Generic Composite Moniker classes.

The implementations of Generic Composite Moniker::Reduce() and Generic Composite Moniker::BindToObject() are particularly important as they manage the interactions between the various elements of the composite, and as a consequence define the semantics of binding to the moniker as a whole.

Generic composite monikers of size zero or of size one are never exposed outside of internal Generic Composite Moniker method implementations. From a client perspective, a Generic Composite Moniker always contains at least two elements.

7.2.4.1. CreateGenericComposite

HRESULT CreateGenericComposite(pmkFirst, pmkRest, ppmkComposite)

Allocate and return a new composite moniker. pmkFirst and pmkRest are its first and trailing elements respectively. Either of pmkFirst and pmkRest may be a generic composite, or another kind of moniker. Generic composites get flattened into their component pieces before being put into the new composite. This function will be called by implementations of IMoniker::ComposeWith() when they wish to carry out a generic compose operation.

Argument	Type	Description
pmkFirst	IMoniker*	the first element(s) in the new composite. May not be NULL.

pmkRest	IMoniker*	the trailing element(s) in the new composite. May not be NULL.
ppmkComposite	IMoniker*	through here is returned the new composite.
return value	HRESULT	S_OK, E_OUTOFMEMORY

7.2.4.2. Generic Composite Moniker–IMoniker::Reduce

Reduction of a generic composite conceptually reduces each of its pieces in a left-to-right fashion and builds up a a composite of the result. If any of the pieces of the composite did not reduce to themselves (and thus, the generic composite overall did not reduce to itself), then this process is repeated.

An optimized implementation of this function might use a more complicated but equivalent algorithm that avoids unnecessarily re-reducing monikers that the composite already knows reduce to themselves.

7.2.4.3. Generic Composite Moniker–IMoniker::BindToObject

Binding to a generic composite works in a right-to-left manner. Conceptually, the generic composite merely forwards the bind request onto its last piece, along the way informing that piece of the moniker to its left in the composite. The last piece, if it needs to, recursively binds to the object to its left. In practice, binding to a generic composite itself has to handle the recursive call on the left-hand object, as was described in IMoniker::BindToObject().

7.2.5. File Moniker class

A File Moniker can be thought of as a wrapper for a path name in the native file system. It's implementation of IMoniker::GetDisplayName(), for example, is trivial: it just returns the path. When bound to, it determines the class of the file by using the API GetClassFile(), makes sure that the appropriate class server is running, then asks it to open the file. On DOS, network file servers are accessed by connecting a local drive letter to the remote drive. File monikers designating such drives internally store the network name to which they are connected instead of the drive letter. This conversion is done as soon as possible, often in the process of parsing a display name.

7.2.5.1. CreateFileMoniker

HRESULT CreateFileMoniker(lpszPathName, ppmk)

Creates a moniker from the given path name. This path may be an absolute path or a relative path. In the latter case, the resulting moniker will need to be composed onto another File Moniker before it can be bound. In either case, it will often be the case that other monikers are composed onto the end of *this* moniker in order to access sub-pieces of the document stored in the file.

Argument	Type	Description
lpszPathName	LPSTR	the path name of the desired file.
ppmk	IMoniker*	the newly created moniker.
return value	HRESULT	S_OK, MK_E_SYNTAX, E_OUTOFMEMORY

7.2.5.2. File Moniker–IMoniker::BindToObject

The class of an object designated by a File Moniker is determined in a platform-specific way; see GetClassFile(). Having found the correct class, File Moniker::BindToObject() will instantiate an instance of it using the interface IPersistFile (which is described in the chapter on "Persistent Storage for Objects"). This interface contains an Load() function, which the File Moniker invokes to open the file and initialize the object.

7.2.5.3. File Moniker–IMoniker::BindToStorage

The present implementation of File Moniker currently supports BindToStorage(..., IID_IStorage, ...) in the case that the designated file in fact a Docfile. Future implementations will also support IStream and ILockBytes.

7.2.5.4. File Moniker–IMoniker::GetDisplayName

File monikers render their display names according to the syntax of the platform on which they are currently found. A File Moniker serialized on one platform then deserialized on another will possibly have different display names on each platform.

7.2.5.5. File Moniker–IMoniker::ParseDisplayName

File monikers designate objects that live in files; however, they have no knowledge of the name space contained *within* that file. A File Moniker for the path "C:\FOO.XLS", for example, knows how to connect to the spreadsheet that is the file, but it does not know anything of the syntax of range-address expression language of the sheet. Consequently, when asked to parse a display name, a File Moniker needs to delegate this operation to the class object that it designates. For this purpose it uses the IParseDisplayName interface. If the class refuses to handle the parse, the parsing is delegated to the object.

7.2.6. Item Moniker class

Item monikers provide a bridge from the generality of IMoniker interface to the simple and common situation in which an OLE object which is a container of other objects also provides a space of names for those objects. Examples include Microsoft Excel, which has "named ranges," and Microsoft Word for Windows, which has "bookmarks."

Item Moniker is a *class*, not a *interface*; that is, it is an *implementation* of IMoniker provided by the OLE libraries, not an interface that others implement. This implementation supports IMoniker interface by converting IMoniker invocations into a series of calls on part of the interface IOleItemContainer; the support required on the container's part is very much like the OLESERVERDOC::GetObject() of OLE 1. The implication is that, for the most part, OLE 2 object implementors do not have to deal much with monikers: they can continue to deal just with "items" in a string form, then wrap them in an Item Moniker as needed to support other interfaces.

Thus, Item Monikers provide a straightforward and easily-implemented generalization of the OLE 1

document[, item]

referencing mechanism to

document[, item[, item[, item]]]

in the case where each item except possibly the last in fact specifies another embedded object which supports IOleItemContainer interface. Each "item" here is actually an Item Moniker.

A client creates an Item Moniker using CreateItemMoniker(). When this new moniker is composed onto the end of a moniker that binds to an IOleItemContainer, the resulting composite moniker will bind to the appropriate contained object.

The following is the IOleItemContainer interface used by Item Monikers:

```
interface IOleItemContainer : IOleContainer {
    virtual    HRESULT    GetObject(lpszItem, dwSpeedNeeded, pbc, riid, ppvObject) = 0;
    virtual    HRESULT    GetObjectStorage(lpszItem, pbc, riid, ppvStorage) = 0;
    virtual    HRESULT    IsRunning(lpszItem) = 0;
    };
```


7.2.6.1. CreateItemMoniker

HRESULT CreateItemMoniker(lpszDelim, lpszItem, ppmk)

Allocate and return a new Item Moniker. It is intended that the resulting moniker be then composed onto the end of a second moniker which binds to something that supports IOleItemContainer interface. The resulting composite moniker when bound will extract the object of the indicated name from within this container.

lpszItem is the item name which will be later passed to IOleItemContainer::GetObject(). lpszDelim is simply another string that will prefix lpszItem in the display name of the Item Moniker.

lpszItem may be NULL; this is the wildcard entry for Item Monikers. See the description of the Item Moniker–IMoniker::IsRunning() implementation in the description of that function definition. Containers who wish to use wild-card Item Monikers to match their elements should register as running:

```
pmkContainer -> wildCardItemMoniker
```

See also IOleItemContainer::IsRunning().

Argument	Type	Description
lpszDelim	LPSTR	a string that will prefix lpszItem in the display name of this moniker. Often an exclamation mark: "!". See also the discussion of syntax in MkParseDisplayName().
lpszItem	LPSTR	the item name to pass to IOleItemContainer::GetObject().
ppmk	IMoniker**	the place to put the new Item Moniker.
return code	HRESULT	S_OK, E_OUTOFMEMORY

7.2.6.2. Item Moniker–IMoniker::BindToObject

Item monikers merely require IOleItemContainer interface of the object to their left, which they obtain by invoking BindToObject() on the moniker of the object to their left. Once they have the container, they merely invoke IOleItemContainer::GetObject() with the internally stored item name.

7.2.6.3. IOleItemContainer::GetObject

HRESULT IOleItemContainer::GetObject(lpszItem, dwSpeedNeeded, pbc, riid, ppvObject)

IOleItemContainer::GetObject() should first check to see if the given item designates an embedded object. If so, then it should load and run the object, then return it. If not, then it should check to see if the item designates a local object within the container. This latter case is just like OLESERVERDOC::GetObject() in OLE 1.

dwSpeedNeeded is an indication of how willing the caller is to wait to get to the object. This value is set by the implementation of Item Moniker; the value it uses is derived from the dwTickCountDeadline parameter in the Bind Context that it receives. dwSpeedNeeded is one of the following values:

```
typedef enum tagBINDSPEED {
    BINDSPEED_INDEFINITE    = 1,  // the caller is willing to wait indefinitely
    BINDSPEED_MODERATE      = 2,  // the caller is willing to wait a moderate amount of time.
    BINDSPEED_IMMEDIATE     = 3,  // the caller is willing to wait only a very short time
    } BINDSPEED;
```

If BINDSPEED_IMMEDIATE is specified, then the object should be returned only if it is already running or if it is a pseudo-object (an object internal to the item container, such as a cell-range in a spreadsheet or a character-range in a word processor); otherwise, MK_E_EXCEEEDEDDEADLINE should be returned. BINDSPEED_MODERATE would include those things indicated by BINDSPEED_IMMEDIATE, plus, perhaps, those objects which are always running when loaded: in this case, *load* (not load & run) the designated object, ask if it is running, and return it if so; otherwise, fail with MK_E_EXCEEEDEDDEADLINE as before. BINDSPEED_INDEFINITE indicates that time is of no concern to the caller.

The actual bind context parameter is also here passed in pbc for the use of more sophisticated containers. Less sophisticated containers can simply ignore this and look at dwSpeedNeeded instead. In effect, what

the implementation of Item Moniker does is convert the deadline in the bind context into an appropriate dwSpeedNeeded, in the hope that the latter is easier to take a decision on for most containers.

Argument	Type	Description
lpszItem	LPSTR	the item in this container that should be bound to.
dwSpeedNeeded	DWORD	a value from the enumeration BINDSPEED. See above.
pbc	IBindCtx*	the actual deadline parameter involved in this binding operation. For the use of more sophisticated containers. Most can ignore this, and instead use dwSpeedNeeded.
riid	REFIID	the interface with which a connection to that object should be made.
ppvObject	void**	the bound-to object is returned here.
return value	HRESULT	S_OK, MK_E_EXCEEEDEDDEADLINE, MK_E_NOOBJECT, E_NOINTERFACE, E_OUTOFMEMORY

7.2.6.4. Item Moniker–IMoniker::BindToStorage

For storage binding, Item Monikers merely require IOleItemContainer interface of the object to their left. The implementation of Item Moniker::BindToStorage() binds to the object to its left using IOleItem-Container interface, then invokes IOleItemContainer::GetObjectStorage() with the internally stored item name.

7.2.6.5. IOleItemContainer::GetObjectStorage

HRESULT IOleItemContainer::GetObjectStorage(lpszItem, pbc, riid, ppvStorage)

If lpszItem designates an item in this container that has an independently identifiable piece of storage (such as does an embedded object), then return access to that storage using the indicated interface.

pbc is the bind context as received by the Item Moniker BindToStorage() call. Most container implementations can simply ignore this value; it is passed for the benefit for more sophisticated containers.

Argument	Type	Description
lpszItem	LPSTR	the item access to whose storage is being requested.
pbc	IBindCtx*	as in IOleItemContainer::GetObject(). Can be ignored by most containers.
riid	REFIID	the interface by which the caller wishes to access that storage. Often IID_IStorage or IID_IStream are used.
ppvStorage	void**	the place to return the access to the storage
return value	HRESULT	S_OK, MK_E_EXCEEDEDDEADLINE, MK_E_NOOBJECT, E_OUTOFMEMORY, E_NOINTERFACE, MK_E_NOSTORAGE

7.2.6.6. IOleItemContainer::IsRunning

HRESULT IOleItemContainer::IsRunning(lpszItem)

Answer whether the given item in this item container is in fact running or not. See IMoniker::IsRunning() for a sketch of how this function is used in Item Monikers.

Argument	Type	Description
lpszItem	LPSTR	the item access to whose running status is being requested.
return value	HRESULT	S_OK, S_FALSE, MK_E_NOOBJECT

7.2.6.7. Item Moniker–IMoniker::ParseDisplayName

Item Moniker::ParseDisplayName() uses IParseDisplayName in the same way as a File Moniker does. Note that it requests this interface from a *different object* than the one that supplies the IOleItemContainer

interface used in BindToObject(), etc.: it asks for IOleItemContainer of the object designated by the moniker to its left, whereas it asks for IParseDisplayName of the object that *it* (the Item Moniker) designates.

7.2.7. Anti Moniker class

An Anti Moniker is a moniker that when composed onto the end of a generic composite moniker removes the last piece. Composing an Anti Moniker onto the end of another kind of moniker should, generally, annihilate the whole other moniker.

Being composed onto the end of another moniker is pretty much the only interesting thing one can do to an anti-moniker: they cannot be bound, their display name is useless, etc. They exist to support implementations of IMoniker::Inverse(); see that function for usage scenarios.

Moniker implementations that use Anti Monikers as inverses should check for Anti Monikers on the right in their implementations of IMoniker::ComposeWith() and collapse down to nothing if so.

7.2.7.1. CreateAntiMoniker

HRESULT CreateAntiMoniker(ppmk)

Create and return a new anti-moniker.

Argument	Type	Description
ppmk	IMoniker**	the place to return the new anti-moniker
return value	HRESULT	S_OK, E_OUTOFMEMORY

7.2.8. Pointer Moniker class

A pointer moniker is a kind of moniker that wraps an existing object pointer in a moniker so that it may participate as a piece in the moniker binding process. Think of pointers as a referencing mechanism into the “active space:” a process’s memory. Most moniker implementations are by contrast references into “passive space:” the representation of an object on disk. Pointer monikers provide a means by which a a given use of a moniker can transparently reference *either* active *or* passive space.

Implementations of functions in IMoniker interface for Pointer Monikers work roughly as follows. BindToObject() turns into a QueryInterface() on the pointer; BindToStorage() returns MK_E_NOSTORAGE; Reduce() reduces the moniker to itself; ComposeWith() always does a generic composition; Enum() returns NULL; IsSystemMoniker() returns MKSYS_NONE; IsEqual() uses the identity test paradigm on pointers after first checking that the other moniker for the right class; Hash() returns a constant; GetTimeOfLastChange() returns MK_E_UNAVAILABLE; Inverse() returns an anti-moniker; RelativePathTo() returns the other moniker; GetDisplayName() returns NULL; and ParseDisplayName() binds to the punk pointer using IParseDisplayName interface and works from there.

Instances of this kind of moniker refuse to be serialized; that is, IPersistStream::Save() will return an error. These monikers can, however, be *marshalled* to a different process in an RPC call; internally, this marshals and unmarshals the pointer using the standard paradigm for marshalling interface pointers.

7.2.8.1. CreatePointerMoniker

HRESULT CreatePointerMoniker(punk, ppmk)

Wrap a pointer in a Pointer Moniker so that it can be presented to interfaces that require monikers for generality, but specific uses of which can usefully deal with a moniker which cannot be saved to backing store.

Argument	Type	Description
punk	IUnknown*	the pointer that we are wrapping up in a moniker.
ppmk	IMoniker**	the returned Pointer Moniker.
return value	HRESULT	S_OK, E_OUTOFMEMORY

7.3. OLE 2 Link Objects

Now that we understand how monikers provide a generalized abstraction of a reference to data, we will examine in detail the workings of the most common place in which monikers are actually used: OLE 2 linked compound-document objects.

As mentioned earlier, OLE 2 provides for tracking links as they move about relative to their sources. In order to support the most general such support it is necessary to ask the moniker of the link source to generate a tracking representation of itself (which would be another moniker, of course, perhaps called a "tracking moniker"). Though this most-general support has been designed, and will be implemented in the future, it is not implemented in OLE 2. Instead, in OLE 2 knowledge of one particularly important link-tracking algorithm is incorporated into the OLE-provided link object: in addition to storing the moniker given to it with IOleLink::SetSourceMoniker(), it also stores a *relative* moniker formed using its *own* moniker, the source moniker, and IMoniker::RelativePathTo(). The relative moniker has priority over the original, absolute moniker: the link object always tries to bind the relative moniker first, using the absolute moniker on if the relative one fails to bind. Using a relative moniker in addition to the absolute moniker in this way covers the following link tracking scenarios:

1. the link source and the link consumer have been copied or moved but retain the same relative structure. A very important common case of this is that of two documents in the same directory. Another case is that of a link between two objects both embedded in a third document. In these situations, the relative moniker succeeds in binding.
2. the link source does not move, but the consumer does (in a manner other than that of the previous case). Here, the relative moniker fails to bind, but the absolute one works.

Link tracking scenarios other than these are not supported by OLE 2. When at a later time more sophisticated tracking is supported, the following will need to be done:

1. A new interface containing functions by which a moniker can be asked to generate a tracking representation of itself will be defined. This interface will be obtained from a moniker using QueryInterface().
2. As a helper, a new moniker implementation will be provided: Tracking Moniker. Instances of this kind of moniker contain a prioritized sequence of monikers which are tried in turn when the Tracking Moniker is reduced or bound.
3. Generic Composite Moniker will likely need to be generalized in a small way in order to obtain the desired compositional behavior with Tracking Moniker and like moniker implementations.
4. The implementation of the OLE link object will need to be revised in order to cause it to use the new tracking support.

All of these steps can easily be done without breaking then-existing OLE 2 applications.

7.3.1. IOleLink interface

From a container's perspective, the architectural difference between an embedding and a link is that a link supports IOleLink interface whereas an embedding does not. IOleLink interface contains functionality by which the moniker inside the link and the link's update options are manipulated.

```
interface IOleLink : IUnknown {
    virtual     HRESULT     SetUpdateOptions(dwUpdateOpt) = 0;
    virtual     HRESULT     GetUpdateOptions(pdwUpdateOpt) = 0;
    virtual     HRESULT     SetSourceMoniker(pmk, rclsid) = 0;
    virtual     HRESULT     GetSourceMoniker(ppmk) = 0;
    virtual     HRESULT     SetSourceDisplayName(lpszDisplayName) = 0;
    virtual     HRESULT     GetSourceDisplayName(lplpszDisplayName) = 0;
    virtual     HRESULT     BindToSource(bindflags, pbc) = 0;
    virtual     HRESULT     BindIfRunning() = 0;
    virtual     HRESULT     GetBoundSource(ppUnk) = 0;
    virtual     HRESULT     UnbindSource() = 0;
    virtual     HRESULT     Update(pbc) = 0;
    };
```


7.3.1.1. IOleLink::SetUpdateOptions

HRESULT IOleLink::SetUpdateOptions(dwUpdateOpt)

This function sets the link-update options for the link object. This controls exactly when the data and / or presentation cache on the consuming end of the link is updated. dwUpdateOpt is taken from the enumeration OLEUPDATE, defined as follows:

```
typedef enum tagOLEUPDATE {
    OLEUPDATE_ALWAYS = 1,
    OLEUPDATE_ONCALL = 3,
    } OLEUPDATE;
```

These flags have the following semantics:

Value	Description
OLEUDPATE_ALWAYS	update the link object whenever possible. This option supports the Automatic link-update option in the Links dialog box. This is the default value.
OLEUPDATE_ONCALL	update the link object only when IOleObject::Update() is called. This option supports the Manual link-update option in the Links dialog box.

The arguments to this function have the following meanings.

Argument	Type	Description
dwUpdateOpt	DWORD	flags taken from the enumeration OLEUPDATE.
return value	HRESULT	S_OK, E_INVALIDARG

7.3.1.2. IOleLink::GetUpdateOptions

HRESULT IOleLink::GetUpdateOptions(pdwUpdateOpt)

Retrieve update options previously set with IOleLink::SetUpdateOptions().

Argument	Type	Description
pdwUpdateOpt	DWORD *	a place to return flags taken from the enumeration OLEUPDATE.
return value	HRESULT	S_OK

7.3.1.3. IOleLink::SetSourceMoniker

HRESULT IOleLink::SetSourceMoniker(pmk, rclsid)

Stores inside of the link object a moniker which indicates the source of the link. This moniker becomes part of the persistent state of the object. In addition to storing this moniker, in order to support link source tracking, link objects also store a relative moniker computed as:

```
pmkOfThisLinkObject->RelativePathTo(pmk).
```

When in the running state (i.e.: the source moniker has been bound and connected), a link object registers itself on its link source to receive rename notifications. When one is received, the link object updates its source moniker to the new name. The primary reason for doing this is to handle as best we can the situation where a link is made to a newly created document that has never been saved, though doing this does provide better link tracking in general. For example, newly created Excel spreadsheets are named “SHEET1”, “SHEET2”, etc. Only when they are saved for the first time do they acquire a persistent identity which is appropriate to store in links to them. So long as the sheet is saved before its link consumer is closed the link will track correctly.

Recall that from the container’s perspective, a link is just an embedding that also happens to support the IOleLink interface. In particular, a link object may be at different times in both the loaded and the running state. When in the loaded state, the link object still needs to be able to carry out a limited amount of class-specific, such as verb enumeration, data format enumeration, etc. In order to be able to carry this out, the link object keeps as part of its persistent state an internal cache of the CLSID of the object to which it was last connected. The parameter rclsid here is the initial value of the cache. The cache is updated whenever the link connects. Further, SetSourceMoniker() does a BindIfRunning(), so if the link source

indicated by pmk is currently running, then rclsid has basically no effect. See also IOleLink::BindTo-Source().

Argument	Type	Description
pmk	IMoniker *	the new moniker for the link.
rclsid	REFCLSID	the value to set for the cached class of the link source that is kept in the link object. Most often either CLSID_NULL or the previous value of the CLSID (obtainable with IOleObject::GetUserClassID()) is passed.
return value	HRESULT	S_OK

7.3.1.4. IOleLink::GetSourceMoniker

HRESULT IOleLink::GetSourceMoniker(ppmk)

Retrieve the indication of the current link source.

Argument	Type	Description
ppmk	IMoniker **	the place at which the moniker currently in the link should be placed.
return value	HRESULT	S_OK

7.3.1.5. IOleLink::SetSourceDisplayName

HRESULT IOleLink::SetSourceDisplayName(lpszDisplayName)

As described above in IMoniker::GetDisplayName(), monikers used to indicate the source of embedded link objects have a display name by which they can be shown to the user. Conversely display names can be parsed into moniker using MkParseDisplayName(). Most often, the indication of a link source is provided directly in a moniker, such as the moniker passed through the clipboard in a **Copy / Paste Link** operation. Less frequently, it is originates in a textual form, such as the text box in the **Edit / Links...** dialog.

Monikers originating in textual form of course need to be parsed into monikers in order to be stored as the source of a link. A key question is whether this is done before or after the display name / moniker is passed to the link object. Both scenarios are supported.

- If the caller wishes to do the parsing, then he calls MkParseUserName() and passes the resulting moniker to IOleLink::SetSourceMoniker().
- If instead it wishes the link object itself to do the parsing, then it should call IOleLink::Set-SourceDisplayName(). This allows the possibility that the link object can optimize the parsing with a subsequent binding.

In the latter case, then by the first time the link needs to be bound the display name will be parsed and the resulting moniker stored in its place.[77] Until such parsing takes place, the link object will not participate in auto-link reconnections (see the Alert Object Table); thus, most callers will either want to themselves call MkParseDisplayName() or will want to let the link object do the parsing but run the link immediately after setting the display name in order to cause the parsing to happen.

Argument	Type	Description
lpszDisplayName	LPSTR	the display name of the new link source. May not be NULL.
return value	HRESULT	S_OK, MK_E_SYNTAX

7.3.1.6. IOleLink::GetSourceDisplayName

HRESULT IOleLink::GetSourceDisplayName(lplpszDisplayName)

This returns the display name of the source of the link using the most efficient means available.

The present implementation carries this out by simply asking the internal moniker for its display name. This is sometimes expensive, though very rarely (and never with any of the OLE-supplied monikers).

---

[77] In fact, the present implementation parses immediately, but that may be optimized in future releases.

Thus, clients who for whom this is a time-critical operation should consider caching the display name themselves.

Argument	Type	Description
lplpszDisplayName	LPSTR*	the place to return the name of the link source. May not be NULL.
return value	HRESULT	S_OK + any error returned from IMoniker::GetDisplayName().

7.3.1.7. IOleLink::BindToSource

HRESULT IOleLink::BindToSource(grfLinkBind, pbc)

Causes the link to bind the moniker contained within itself. When the user double-clicks a link and the server must be located, this is the workhorse function which is invoked to cause the connection to happen, though normally this is invoked by being called internally by DoVerb().

pbc is the bind context to use for the bind operation.

When binding a link, it may be the case that the current class of the link source is not the same as it was the previous time that the link tried to connect. Imagine, for example, a link to a Lotus spreadsheet object that the user subsequently converted (using the Change Type dialog) to an Excel sheet. grfLinkBind controls the behaviour of the binding operation in this scenario. It contains values taken from the enumeration OLELINKBIND:

```
typedef enum tagOLELINKBIND {
    OLELINKBIND_EVENIFCLASSDIFF = 1,
    } OLELINKBIND;
```

If OLELINKBIND_EVENIFCLASSDIFF is not provided, then this function will return OLE_E_CLASSDIFF if the class is different than the previous time that this link successfully bound. If OLELINKBIND_EVENIFCLASSDIFF *is* given, then the bind process will proceed even if the class has changed.

When OleRun() is invoked on an embedding which is in fact a link object, it conceptually just invokes this function as

```
plink->BindToSource(0, 0)
```

If OleRun() returns OLE_E_CLASSDIFF, then the client will have to call BindToSource() directly.

Argument	Type	Description
grfLinkBind	DWORD	values from the enumeration OLELINKBIND.
pbc	IBindCtx*	the bind context to use for the bind operation. May be NULL.
return value	HRESULT	S_OK, OLE_E_CLASSDIFF, MK_E_NOOBJECT, MK_E_EXCEEDEDDEADLINE, MK_E_SYNTAX

7.3.1.8. IOleLink::BindIfRuning

HRESULT IOleLink::BindIfRunning()

This binds the link to its source only if said source is in fact presently running.

Argument	Type	Description
return value	HRESULT	S_OK, MK_E_SYNTAX, others

7.3.1.9. IOleLink::GetBoundSource

HRESULT IOleLink::GetBoundSource(ppUnk)

This function retrieves the object to which the this link is currently connected, if any is present. In the event that no source is currently connected, then S_FALSE is returned by the function and NULL is returned through *ppunk.

Argument	Type	Description
ppUnk	IUnknown*	the place to return the currently-connected source of this object. May not be NULL. In the event that no source is currently connected, NULL is returned.
return value	HRESULT	S_OK, S_FALSE

7.3.1.10. IOleLink::UnbindSource

HRESULT IOleLink::UnbindSource()

If the link object is presently connected to it source, then break that connection.

Argument	Type	Description
return value	HRESULT	S_OK

7.3.1.11. IOleLink::Update

HRESULT IOleLink::Update(pbc)

Carry out the same functionality as is described in IOleObject::Update(), but in addition record any bound objects in the passed in bind context. IOleObject::Update() on an object which also supports IOleLink should just call IOleLink::Update(NULL) on itself. Non-NULL uses of pbc allow complex binding scenarios to be optimized by callers.

The OLE-provided Link Object implementation of IOleLink::Update() requires that the link be made running; that is, that the source moniker be bound. In the event that the source is unavailable, then the implementation of Update() is careful not to loose the presentations that it already has cached from the previous time it connected. That is, a failed connect attempt will not cause any presentations to be lost.

Argument	Type	Description
pbc	IBindCtx*	the bind context to use for binding operations carried out during the update. May be NULL.
return value	HRESULT	S_OK, ...

7.3.1.12. Link Object – IOleObject::IsUpToDate

The implementation of IsUpToDate() in links is a bit tricky. The problem is two-fold:

1) how to avoid comparing the local clock against the remote clock at the other end of the link, since the two may be very much out of synchronization, and
2) how to handle the fact that we must treat equal change times reported from the source conservatively as "out of date" due to possible lack of precision in the source's clock. For example, if the source data is changing at the rate of 20 times per second, but the only clock that the source has available with which to report change times has a resolution of one second, then each group of 20 changes will report exactly the same change time.

The solutions to these problems are embodied in the OLE-provided link object implementation; however, it is instructive nevertheless that others understand how they are addressed.

Consider Figure 79. Whenever a link object updates from its source, it stores the remote time (rtUpdate) beyond which the data in that update is known not to have changed; this is the time returned by GetTimeOfLastChange() on the source moniker. In addition to this time, the link object also stores the local time (ltChangeOfUpdate) at which it *first obtained a particular value* of rtUpdate. That is, when rtUpdate is revised as a result of an Update(), if the new value is different than the old, then ltChangeOfUpdate is set to

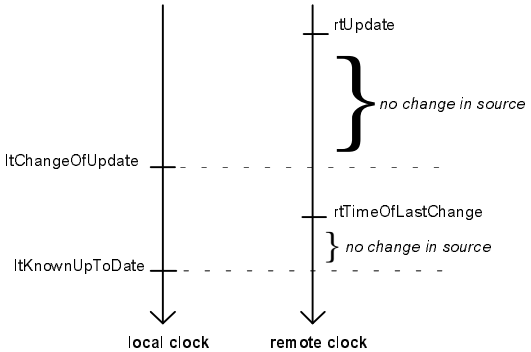

Figure 79. Out-of-date detection for links

the current local time; if it is the same, then ltChangeOfUpdate is left alone. Finally, the link object stores the local time (ltKnownUpToDate) at which it last new itself to be up to date. For auto-links, this time is updated as part of their save sequence. Manual links update this time only at Update() time. When IsUpToDate() is invoked, it retrieves GetTimeOfLastChange(), the value indicated by rtTimeOfLast-Change in the diagram. Given this structure, a link is deemed to be up to date if (approximately):

(rtTimeOfLastChange - rtChangeOfUpdate) < (ltKnownUpToDate - ltUpdate)

More completely, backwards movement of clocks needs to be considered, and a case of equal remote times is taken as out of date (per the problem indicated above) only if less than two seconds has elapsed on our local clock (that is, we assume that remote clocks have a precision of at least two seconds).

7.3.2. Running Object Table

In general when binding to an object we want to open it if it is currently passive, but if not, then we want to connect to the running instance. A link to a Lotus 123 for Windows spreadsheet, for example, when first bound to should open the spreadsheet, but a second bind should connect to the already-open copy. The key technical piece that supports this is the Running Object Table.

The Running Object Table is a globally accessible table on each workstation. It keeps track of the objects that are currently running on that workstation so that if an attempt is made to bind to one a connection to the currently running instance can be made instead of loading the object a second time. The table conceptually is a series of tuples, each of the form:

(pmkObjectName, pvObject)

The first element is the moniker that if bound should connect to the running object. The second element is the object that is publicized as being available, the object that is running. In the process of binding, monikers being bound with nothing to their left consult the pmkObjectName entries in the Running Object Table to see if the object that they (the moniker being bound) indicate is already running.

Access to the Running Object Table is obtained with the function GetRunningObjectTable(). This returns an object with the interface IRunningObjectTable (note as described earlier, however, that moniker implementations should not use this API, but should instead access the Running Object Table from the bind context they are passed).

As entries are placed into the Running Object Table, they are matched against the Alert Object Table to see if any auto-link reconnections need to be done.[78]

```
interface IRunningObjectTable : IUnknown {
    virtual    HRESULT    Register(reserved, pUnkObject, pmkObjectName, pdwRegister) = 0;
    virtual    HRESULT    Revoke(dwRegister) = 0;
    virtual    HRESULT    IsRunning(pmkObjectName) = 0;
    virtual    HRESULT    GetObject(pmkObjectName, ppunkObject) = 0;
    virtual    HRESULT    NoteChangeTime(dwRegister, pfiletime) = 0;
    virtual    HRESULT    GetTimeOfLastChange(pmkObjectName, pfiletime) = 0;
    virtual    HRESULT    EnumRunning(ppenumMoniker) = 0;
    };

SCODE    GetRunningObjectTable(reserved, pprot);
```

7.3.2.1. GetRunningObjectTable

HRESULT GetRunningObjectTable(reserved, pprot)

Return a pointer to the Running Object Table for the caller's context.

Argument	Type	Description
reserved	DWORD	reserved for future use; must be zero.
pprot	IRunningObjectTable**	the place to return the running object table.
return value	HRESULT	S_OK

[78] Recall that the Alert Object Table is not provided in this OLE release.

7.3.2.2. IRunningObjectTable::Register

HRESULT IRunningObjectTable::Register(reserved, pUnkObject, pmkObjectName, pdwRegister)

Register the fact that the object pUnkObject has just entered the running state and that if the moniker pmkObjectName is bound to, then this object should be used as the result of the bind (with an appropriate QueryInterface()).

The moniker pmkObjectName should be fully reduced before registration. See IMoniker::Reduce() for a more complete discussion. If an object goes by more than one fully reduced moniker, then it should register itself under all such monikers. Here, "fully reduced" means reduced to the state MKRREDUCE_THROUGUSER.

OLE compound document objects should announce themselves as running by calling this function as soon as all of the following are true:

1. The object is in the running state.
2. The object knows its full moniker (see IOleObject::SetMoniker()). This is true if both of the following are true:
   2a. A moniker for the object relative to its container has been assigned to the object. Recall that this is part of the persistent state of the object.
   2b. The object knows the current moniker of its container (almost always through its container calling IOleObject::SetMoniker()). Recall that the moniker of the object's container is not part of the persistent state of the object.
3. There is any possibility that a link to the object or something that it contains exists.

Normally, if a link has *ever* been made to an object, then it must be assumed that the link to the object still might exist. The consumer of the link might be on a floppy disk somewhere, for example, which may later reappear. The exceptions are some rare situations where a link is created but almost immediately destroyed before the link source is saved.

The moniker with which the OLE object should register itself as running is its full moniker as described in IOleObject::GetMoniker().

Registering a second object under the same moniker sets up a second independent registration, though MK_S_MONIKERALREADYREGISTERED is returned instead of S_OK. This is done without regard to the value of pUnkObject in the second registration; thus, registering the exact same (pmkObjectName, pUnkObject) pair a second time will set up a second registration. It is not intended that multiple registration under the same moniker be a common occurrence, as which registration actually gets used in various situations is nondeterministic.

The arguments to this function are as follows:

Argument	Type	Description
reserved	DWORD	reserved for future use; must be zero.
pUnkObject	IUnknown*	the object which has just entered the running state.
pmkObjectName	IMoniker*	the moniker which would bind to the newly running object.
pdwRegister	DWORD*	a place to return a value by which this registration can later be revoked. May not be NULL.
return value	HRESULT	S_OK, MK_S_MONIKERALREADYREGISTERED, E_NOMEMORY

7.3.2.3. IRunningObjectTable::Revoke

HRESULT IRunningObjectTable::Revoke(dwRegister)

Undo the registration done in IRunningObjectTable::Register(), presumably because the object is about to cease to be running. Revoking an object that is not registered as running returns the status code MK_S_NOT_RUNNING. Whenever any of the conditions that cause an object to register itself as running cease to be true, the object should revoke its registration(s). In particular, objects must be sure to extant

registrations of themselves from the Running Object Table as part of their release process; there is no means by which entries in the Running Object Table can be removed automatically by the system.

Argument	Type	Description
dwRegister	DWORD	a value previously returned from IRunningObjectTable::Register().
return value	HRESULT	S_OK, MK_S_NOT_RUNNING.

7.3.2.4. IRunningObjectTable::IsRunning

HRESULT IRunningObjectTable::IsRunning(pmkObjectName)

This function should, in general, only be called by implementations of IMoniker::IsRunning(); clients of monikers should invoke this on their monikers, rather than asking the R.O.T. directly.

Inquire by looking up in this Running Object Table as to whether an object with this moniker is currently registered as running. Success or failure is indicated using the return codes S_OK or S_FALSE. The R.O.T. compares monikers by sending IsEqual() to the monikers already in the table with moniker on the right as an argument.

Argument	Type	Description
pmkObjectName	IMoniker*	the moniker that we want to see is running
return value	HRESULT	S_OK, S_FALSE.

7.3.2.5. IRunningObjectTable::GetObject

HRESULT IRunningObjectTable::GetObject(pmkObjectName, ppunkObject)

If the object designated by pmkObject name is registered as actually running, then return the object so registered. The R.O.T. compares monikers by sending IsEqual() to the monikers already in the table with moniker on the right as an argument.

This is the function moniker implementations should use to test if they are already running (and get the pointer to the object if so).

Argument	Type	Description
pmkObjectName	IMoniker*	the moniker in whom interest is being expressed.
ppunkObject	IUnknown**	the place to return the pointer to the object. A returned value of NULL indicates that the object is not registered.
return value	HRESULT	S_OK, MK_S_NOT_RUNNING

7.3.2.6. IRunningObjectTable::NoteChangeTime

HRESULT IRunningObjectTable::NoteChangeTime(dwRegister, pfiletime)

Make a note of the time that a particular object has changed in order that IMoniker::GetTimeOfLast-Change() can report an appropriate change time. This time so registered is retrievable with IRunningOb-jectTable::GetTimeOfLastChange(). Objects should call this as part of their data change notification process.

Argument	Type	Description
dwRegister	DWORD	the token previously returned from IRunningObjectTable::Register(). The moniker whose change time is noted is the one specified in pmkObjectName in that call.
pfiletime	FILETIME *	on entry, the time at which the object has changed.
return value	HRESULT	S_OK

7.3.2.7. IRunningObjectTable::GetTimeOfLastChange

HRESULT IRunningObjectTable::GetTimeOfLastChange(pmkObjectName, pfiletime)

As with IMoniker::IsRunning(), this function should, in general, only be called by implementations of IMoniker::GetTimeOfLastChange(); clients of monikers should invoke this on their monikers, rather than asking the R.O.T. directly.

Look up this moniker in the running object table and report the time of change recorded for it if same is present. The R.O.T. compares monikers by sending IsEqual() to the monikers already in the table with moniker on the right as an argument. Implementations of IMoniker::GetTimeOfLastChange(), when invoked with pmkToLeft == NULL, will want to call this function as the first thing they do.

Argument	Type	Description
pmkObjectName	IMoniker *	the moniker in which we are interested in the time of change.
pfiletime	FILETIME *	on exit, the place at which the time of change is returned.
return value	HRESULT	S_OK, MK_S_NOT_RUNNING

7.3.2.8. IRunningObjectTable::EnumRunning

HRESULT IRunningObjectTable::EnumRunning(ppenumMoniker)

Enumerates the objects currently registered as running. The returned enumerator is of type IEnumMoniker, which enumerates monikers.

```
typedef Enum<IMoniker*>      IEnumMoniker;
```

The monikers which have been passed to IRunningObjectTable::Register() are enumerated.

Argument	Type	Description
ppenumMoniker	IEnumMoniker**	the place at which to return the enumerator.
return value	HRESULT	S_OK, E_OUTOFMEMORY

7.3.3. Auto-link Reconnection & the Alert Object Table

Warning to readers: the following discusses what the Alert Object Table does and gives an overview of how its implementation would work. However, the Alert Object Table specifically and auto-link reconnection in general are not supported in this release of OLE. The description of their design and support has been retained in this specification in order to educate the curious.
*It is explicitly **not** guaranteed that this functionality will at all be supported at any particular time in the future.[79] Further, if it is supported at some future time, that it is **not** guaranteed that it will be supported exactly as described herein.*

The problem of correctly reconnecting auto-links when their link sources become open was discussed at the beginning of this chapter. We now look at that problem in depth. Consider the picture shown in Figure 80, which illustrates a link from a (whole) chart stored in the file QUARTERS.XLC into a report document REPORT.DOC. This link is an automatic (as opposed to manual) link. Suppose the user opens REPORT.DOC, then double clicks on the link. QUARTERS.XLC is opened in its editor, and if the use makes any changes to the chart, then they are immediately reflected in the report. However, if instead of opening the chart by double clicking the link the user opens it by using the **File/Open** command in its editor, then changes are *not* reflected in the report: the connection to the report is not automatically made when the chart is opened. This is very surprising behaviour to users. The primary component of OLE 2 that addresses this problem is the Alert Object Table.

[79] Though clearly such support would be valuable.

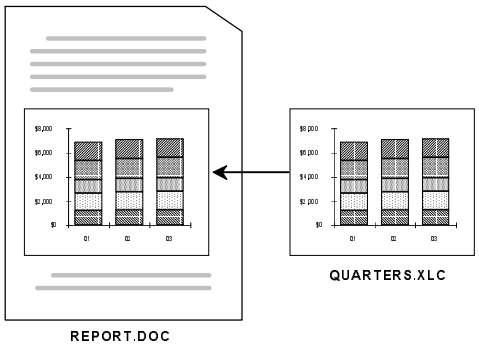

Figure 80. Link from chart to report.

Suppose that both documents are presently closed, and the user then opens REPORT.DOC. At this point in time, OLE has to be told that the link inside the report is awaiting the appearance of a certain object, namely the chart. If and when the chart later appears, the link object will notified and a connection automatically established. While the link is awaiting the appearance of its source, it is said to be in the "alert" state. The Alert Object Table contains a list of objects and the monikers whose appearance each of them is awaiting.

Notice that embedded objects cannot go into the Alert state, since they don't contain monikers whose appearance can be awaited. However, an embedded object may *contain* a link object, and in general the embedding can be of an arbitrary depth before the link is reached. Furthermore, while in the alert state, such a nested link object may not even be loaded off of disk at all; only the outermost embedding need have been loaded (in order to get its picture, which contains the pictures of all the nested embeddings and the link). Thus, the process of notifying an alert object may require that it and intermediate objects above it be loaded before the reconnection can occur. We accomplish this by having the alert object register itself using a moniker; in order to do a notification we bind this moniker, which will do any needed intermediate loading.

In order that we can compare the trigger monikers of the objects in the Alert Object Table with the monikers of the objects currently registered as running, both monikers need to be reduced to the same level: MKREDUCE_THROUGHUSER.

7.3.3.1. OLEOBJPRIV & OLEOBJMON structures

Normally, the internal state of links and embeddings is completely hidden to the outside: clients access this state solely through various interfaces on the object. However, in order that auto-link reconnection can be done it is necessary that some of this internal state be accessible when the object is not even loaded from the disk. Accordingly, the state needed to support reconnection needs to be in a publicly accessible location in the storage of the link. This state is manipulated using the OLEOBJPRIV structure and the functions WriteOleObjPrivStg() and ReadOleObjPrivStg(). It is intended that these functions are *only* to be used by implementations of object handlers and by the implementation of StgEnumObjects().

Reminder: These structures are not implemented.

```
typedef struct tagOLEOBJMON {
    IMoniker *      pmkObject;              // for links and embeddings
    IMoniker *      pmkSourceRel;           // non-NULL for links only
    IMoniker *      pmkSourceAbs;           // non-NULL for links only
    } OLEOBJMON;

typedef struct tagOLEOBJPRIV {
    OLEOBJMON;                              // anonymous member; its members merge into current scope
    LPSTR           lpszDisplayNameCache;   // non-NULL for links only. NULL if cache is empty.
    DWORD           dwUpdateOpt;            // relevant for links only
    } OLEOBJPRIV;
```

This structure definition makes use of an anonymous member in order that the internally stored monikers may be separated into a separate typedef that can be reused in functions such as RegisterAsAlert(). The members of the OLEOBJMON become members of the OLEOBJPRIV. The net effect is to create an OLEOBJPRIV which appears to be defined as:

```
typedef struct tagOLEOBJPRIV {
    IMoniker *      pmkObject;              // for links and embeddings
    IMoniker *      pmkSourceRel;           // non-NULL for links only
    IMoniker *      pmkSourceAbs;           // non-NULL for links only
    LPSTR           lpszDisplayNameCache;   // non-NULL for links only. NULL if cache is empty.
    DWORD           dwUpdateOpt;            // relevant for links only
    } OLEOBJPRIV;
```


All objects, both links and embeddings, which have a moniker that has been assigned to them with IOleObject::SetObjectMoniker() store that moniker in the pmkObject member of the OLEOBJPRIV. The remaining members of the OLEOBJPRIV are applicable only to links. For a link object, it is always the case that at least one of the members pmkSourceAbs and lpszDisplayNameCache is non-NULL.

The members of OLEOBJPRIV have the following meanings:

Member	Description
pmkObject	a moniker for the embedded or linked object. When passed to Read/WriteOleObjPrivStg(), this must be the moniker of the object, this moniker must be the moniker of the object *relative to its container;* when passed to RegisterAsAlert(), this must be a full moniker for the object. See also StgEnumObjects().
pmkSourceRel	a moniker indicating the source of the link relative to the moniker of the object which is the link target.
pmkSourceAbs	the absolute moniker indicating the source of the link.
lpszDisplayNameCache	the cached display name of the link source.
dwUpdateOpt	the update options of the link object.

7.3.3.2. WriteOleObjPrivStg

HRESULT WriteOleObjPrivStg(pstg, pobjpriv)

Write the OLEOBJPRIV for a link or an embedding to the IStorage instance in which that object is stored. pobjpriv->pmkObject is the moniker of the object relative to its container. This function should only be used by implementations of embedding or link handlers.

Reminder: This function is not implemented.

Argument	Type	Description
pstg	IStorage*	the storage into which the handler state is to be written.
pobjpriv	OLEOBJPRIV*	the handler state which is to be written. May not be NULL, though some members of this structure may be (especially if the object is not a link.)
return value	HRESULT	S_OK.

7.3.3.3. ReadOleObjPrivStg

HRESULT ReadOleObjPrivStg(pstg, pobjpriv)

Read an oleobjpriv previously written with WriteOleObjPrivStg(). This function should only need to be called by implementations of embedding and link handlers, and by the implementation of StgEnumObjects(). This function returns S_OK if successful; it returns E_FAIL if no OLEOBJPRIV has ever been written to this IStorage.

Reminder: This function is not implemented.

Argument	Type	Description
pstg	IStorage*	the storage from which the handler state is to be read.
pobjpriv	OLEOBJPRIV*	the handler state which is to be read. Caller allocated (with members in an undefined state); filled in by the callee.
return value	HRESULT	S_OK, E_FAIL.

7.3.3.4. StgEnumObjects

HRESULT StgEnumObjects(pstgRoot, pmkRoot, grfFilter, grfMode, ppenumGrovel)

Reminder: This function is not implemented.

Answer an enumerator that will enumerate all the OLE objects transitively contained within the passed IStorage instance. The enumerator returned by this utility function provides an easy way for a container implementor to put all of his transitively-contained link objects into the alert state. Recall from the chapter on Persistent Storage for Objects that each embedded object has associated with it an instance of IStorage; if an object is also a container then the IStorage instances of nested objects are children of the container's IStorage. The enumerator walks the tree of these storage object underneath the IStorage passed as pstg, looking for objects.

Objects are identified by the presence of a class tag in the IStorage (see WriteClassStg() / IStorage::SetClass()); there may be intermediate non-object IStorage instances used by containers, but these are skipped in the enumeration.

Link objects are identified by the presence of their source moniker being stored in a well-known location inside their persistent representation. The enumerator digs out this trigger moniker and returns it (and other necessary state) during the enumeration.

The enumerator is of type IEnumGrovel, which is defined as

```
typedef Enum<GROVELOBJECT> IEnumGrovel;
```

where

```
typedef struct {
    OLEOBJPRIV;                 // anonymous member; its members merge into current scope.
    IStorage *      pstg;       // the storage of the link object.
    } GROVELOBJECT;
```

(Like the definition of OLEOBJPRIV, the definition of GROVELOBJECT makes use of anonymous structure members.)

In detail, the process of enumerating links is done as follows. The enumeration walks the tree of IStorage objects contained beneath pstgRoot. It looks at each such IStorage and from it retrieves the class id and the handler state. For each link object that it finds, the enumeration returns in a GROVELOBJECT the OLEOBJPRIV together with the IStorage instance in which the link was found.

The pmkObject member returned in the GROVELOBJECT (i.e.: the OLEOBJPRIV::pmkObject member found in the anonymous OLEOBJPRIV member of GROVELOBJECT) is different than the pmkObject member that is persistently stored inside the IStorage. Whereas the latter is the moniker of the object relative to its container, the moniker returned in the GROVELOBJECT is the full moniker to the object. This transformation is accomplished by composing the chain of relative-monikers found in the objects on the path from pstgRoot down to the link object onto the end of the moniker pmkRoot.

A container can put its transitively-contained links into the alert state by calling RegisterAsAlert() using the information returned in the GROVELOBJECT. They can be taken out the alert state by calling RevokeAsAlert() instead.

The parameter grfFilter can control whether links, embeddings, or both are enumerated.

```
typedef enum tagSTGGROVEL {
    STGGROVEL_LINKS         = 1,    // without this flag, links are skipped in the enumeration.
    STGGROVEL_EMBEDS        = 2,    // without this flag, embeddings are skipped in the enumeration.
    } STGGROVEL;
```

The parameter grfMode can be used to control how nested storage objects are opened in the process of this groveling. Legal values from the enumeration STGM which may be passed here are:

STGM_READ, STGM_READWRITE, STGM_SHARE_(any)

Other STGM values are illegal in this function (in particular, STGM_TRANSACTED). If a STGM_SHARE_* value is not *explicitly* passed in grfMode, then STGM_SHARE_DENY_WRITE is used.

Once again, readers are reminded that this function is not implemented; this description is included for educational purposes only.

Argument	Type	Description
pstgRoot	IStorage*	the storage object whose tree structure we are to grovel.
pmkRoot	IMoniker*	the moniker which forms the prefix of the moniker to the alert objects.
grfFilter	DWORD	flags from the enumeration STGGROVEL.

grfMode	DWORD	flags from the enumeration STGM (described in the storage chapter).
ppenumGrovel	IEnumGrovel**	the place at which the enumerator is returned.
return value	HRESULT	S_OK

7.3.3.5. RegisterAsAlert

HRESULT RegisterAsAlert(pobjmon, pdwRegister)

Reminder: this function is not implemented.

Register the fact that one object is awaiting the appearance of a second object. When the second object appears, the first object is connected to and IOleAlert::OnAppear() invoked. The first object, the one which awaits the second, is indicated by the moniker pobjmon->pmkObject.

The object which the first object awaits is indicated using pobjmon->pmkSourceRel and pobjmon->pmkSource-Abs. These contain a relative and an absolute moniker to the source, respectively. The relative moniker is interpreted relative to pobjmon->pmkObject. At most one of these may be non-NULL. If only one of the source monikers is non-NULL, then the object is registered as awaiting that one moniker. If instead both are non-NULL, the object is registered as awaiting *either* source. However, the relative moniker takes priority: if the absolute moniker appears first, then before the object is awakened from its alert state, the relative moniker is bound with BINDFLAGS_JUSTTESTEXISTENCE and an infinite deadline in order to determine whether it in fact exists or not. If it does, then the registration on the absolute moniker is discarded.[80]

A common situation for which optimization is done the situation where the result pmkRelFull of

```
IMoniker * pmkRelFull;
(pobjmon->pmkObject)->ComposeWith(pobjmon->pmkSourceRel, &pmkRelFull)
```

is in fact *equal* to pobjmon->pmkSourceAbs. In this case, pmkSourceAbs and pmkRelFull indicate the same link source, and the awaiting of only one object need be registered.

It is important that the monikers pmkSourceAbs and pmkSourceRel *not* be reduced by the caller this function. Reduction of monikers is in general an expensive operation, comparable to that of binding, since it may need to load arbitrarily large amounts of data (in most situations, though, reduction is cheap). Objects are put in the alert state precisely because a user does not at a particular time want to pay the cost of doing a full bind. Thus, to reduce the monikers now defeats our whole purpose.

Furthermore, it is the case that only a fully reduced moniker can be compared against the monikers in the Running Object Table. The matching process in the Alert Object Table gets out of this dilemma by incrementally *but cheaply* trying to reduce the trigger monikers using a deadline binding option. When reduced under a deadline, a moniker can fail to reduce because some intermediate object needed by the binding process is not currently running. In this case, the Alert Object Table registers the original trigger moniker as awaiting the appearance of this intermediate object; that is, the trigger moniker itself enters the alert state. When the intermediate object appears, the trigger moniker is notified, and the reduction is tried again. Eventually, the trigger moniker fully reduces, and will then be compared against current and future entries in the Running Object Table.

Argument	Type	Description
pobjmon	OLEOBJMON*	the monikers indicating the object and the source(s) it awaits.
pdwRegister	DWORD*	the place at which a registration token is returned.
return value	HRESULT	*TBD.*

7.3.3.6. RevokeAsAlert

HRESULT RevokeAsAlert(dwRegister)

Reminder: this function is not implemented.

[80] In summary: auto-link reconnection in the face of link-tracking is a **really** hard problem. It gets particularly hairy when an extensible tracking architecture is considered.

Undo the effects of a previous RegisterAsAlert().

Argument	Type	Description
dwRegister	DWORD	a value returned previously from RegisterAsAlert().
return value	HRESULT	S_OK, ALERT_E_NOREGISTRATION

7.3.4. IOleAlert interface

Reminder: this interface is not implemented.

IOleAlert interface is the interface with which the Alert Object Table binds to an object in order to inform it of the appearance of the moniker which it was registered as awaiting. This interface is very simple, containing but one function.

```
interface IOleAlert : IUnknown {
    virtual     void  OnAppear(pmkTrigger) = 0;
    };
```

7.3.4.1. IOleAlert::OnAppear

void IOleAlert::OnAppear(pmkTrigger)

Reminder: this function is not implemented.

Informs the alert object that the object that it was registered as awaiting has now appeared.

Argument	Type	Description
pmkTrigger	IMoniker*	the moniker of the object that has now appeared.