OLE Controls/COM Objects for the Internet

Internet Awareness for Objects, Controls, and Containers

FOURTH DRAFT 18 April, 1996

Distribution: Draft

© Copyright Microsoft Corporation, 1995-1996. All Rights Reserved.

NOTE: THIS DOCUMENT IS AN EARLY RELEASE OF THE FINAL SPECIFICATION. IT IS MEANT TO SPECIFY AND ACCOMPANY SOFTWARE THAT IS STILL IN DEVELOPMENT. SOME OF THE INFORMATION IN THIS DOCUMENTATION MAY BE INACCURATE OR MAY NOT BE AN ACCURATE REPRESENTATION OF THE FUNCTIONALITY OF THE FINAL SPECIFICATION OR SOFTWARE. MICROSOFT ASSUMES NO RESPONSIBILITY FOR ANY DAMAGES THAT MIGHT OCCUR EITHER DIRECTLY OR INDIRECTLY FROM THESE INACCURACIES. MICROSOFT MAY HAVE TRADEMARKS, COPYRIGHTS, PATENTS OR PENDING PATENT APPLICATIONS, OR OTHER INTELLECTUAL PROPERTY RIGHTS COVERING SUBJECT MATTER IN THIS DOCUMENT. THE FURNISHING OF THIS DOCUMENT DOES NOT GIVE YOU A LICENSE TO THESE TRADEMARKS, COPYRIGHTS, PATENTS, OR OTHER INTELLECTUAL PROPERTY RIGHTS.

Contents

1. Introduction and Requirements
    1.1. Non-Requirements
2. Control Instantiation
    2.1. Author Time Instantiation
    2.2. Saving a Control at Publish Time
    2.3. Run Time Instantiation
3. Storage of Control Persistent Data
    3.1. A Review of Embedding and Linking á la OLE Documents
    3.2. Persistent Embedding and Linking (*IPersist** Interfaces)
    3.3. Use of *IPersistMoniker*
    3.4. Relationship to the HTML OBJECT Tag
4. Data Path Properties
    4.1. Exposing Data Path Properties and Corresponding Data Format
        4.1.1. Author-Time Discovery of Path Properties
    4.2. Assigning Path Properties
        4.2.1. Storing and Comparing Path Properties
        4.2.2. Integrated Author Tool Browsing and Control Property Pages
    4.3. Container Supply of Monikers via *IBindHost*
        4.3.1. Overview of *IBindHost*
        4.3.2. Generic Siting with *IObjectWithSite*
    4.4. Cooperative and Asynchronous Data Retrieval
        4.4.1. Aborting a Data Transfer
        4.4.2. Transfer Priority
        4.4.3. Controls that use WinINet or Sockets Directly
    4.5. Object Persistence and Path Properties
    4.6. Container Flexibility in Moniker Choice
    4.7. Exposing Path Properties from Nested Controls
5. Communicating Control "Readiness"
    5.1. Use of E_PENDING
    5.2. The *OnReadyStateChange* Event and *ReadyState* Property
6. Other Considerations
    6.1. Component Categories for Describing Internet-Aware Objects
7. Summary
    7.1. Added Standard Properties, Methods, Events, and Interfaces
    7.2. Requirements for Internet-Aware Controls and Objects
    7.3. Requirements for Internet-Aware Containers/Clients/Controllers
8. Standard Internet-Aware Objects
9. Interface Reference
    9.1. The *IBindHost* Interface
        9.1.1. *IBindHost::CreateMoniker*
        9.1.2. *IBindHost::MonikerBindToStorage*
        9.1.3. *IBindHost::MonikerBindToObject*
    9.2. The *IServiceProvider* Interface
        9.2.1. *IServiceProvider::QueryService*
    9.3. The *IObjectWithSite* Interface
        9.3.1. *IObjectWithSite::SetSite*
        9.3.2. *IObjectWithSite::GetSite*
    9.4. The *IPersistMemory* Interface
        9.4.1. *IPersistMemory::Load*
        9.4.2. *IPersistMemory::Save*
    9.5. The *IPersistPropertyBag* Interface

© Microsoft Corporation 1995-1996, All Rights Reserved.

9.5.1. *IPersistPropertyBag::InitNew*
9.5.2. *IPersistPropertyBag::Load*
9.5.3. *IPersistPropertyBag::Save*
9.6. The *IPropertyBag* Interface
9.6.1. *IPropertyBag::Read*
9.6.2. *IPropertyBag::Write*
9.7. The *IErrorLog* Interface
9.7.1. *IErrorLog::AddError*

© Microsoft Corporation 1995-1996, All Rights Reserved.

References

The following documents are referred to in this document as they contain relevant information:

Full Name	*Referred to As*
Asynchronous Moniker Design Specification	**Asynchronous Monikers**
Component Categories	**Component Categories**
Compound Files on the Internet	**Internet Files**
Inserting Multimedia Objects into HTML3, link via http://www.w3.org/pub/WWW/MarkUp	**HTML Standards**
OLE Control and Control Container Guidelines v2.0	**Control Guidelines**
URL Monikers	**URL Monikers**

© Microsoft Corporation 1995-1996, All Rights Reserved.

1 Introduction and Requirements

There is no question that OLE Controls, or COM objects in general, are useful as part of a document or page on an Internet site. The real questions and problems in need of solutions have to do with how an control is specified within a document, and how a control can behave well in a "slow-link" environment such as the Internet, including the means through which the control retrieves its data in an incremental or progressive fashion, working well in concert with other controls that may also be retrieving their data in the same manner.

It should be immediately apparent that many controls, such as small buttons and label controls, that do not have significant amounts of persistent data at all, are already suitable for use in the Internet environment. Such controls, even those that already exist, need virtually no changes in order to work well inside browsers. The primary concern in this specification are those controls that do have significant amounts of persistent data.

This document offers solutions for making a control work well in the Internet environment with the ultimate goal of delivering optimal quality of service to end users. As browser speed is one of the primary factors in users' perception of quality, this specification aims to provide solutions that allow a document or page to become visible as soon as possible, interactive very shortly thereafter, while allowing controls to retrieve large data blocks in the background.

Thus the problems under consideration and the sections in this document that cover them are the following:

- What is the exact instantiation mechanism for a control? That is, how does a container get the class code bits into memory? (Section 2)
- What options are available for the storage of a control's persistent data, both properties and large data BLOBs (such as a bitmap image or video data)? (Section 3)
- When a control's data includes potentially huge binary streams (bitmap, video, sound, etc.), how does the control access this data asynchronously and incrementally in cooperation with the container (including the ability for the container to display progress UI)? (Section 4)
- How does a control notify the container that the control has finished retrieving asynchronous data and is ready for interaction with the user? (Section 5)

In addition to these topics, Sections 6 covers miscellaneous topics including component categories, Section 7 supplies a summary of "Internet-awareness" requirements and options including new dispIDs and GUIDs, Section 8 describes standard Internet-aware picture, sound, and video objects, and Section 9 provides reference material for interfaces described in this document.

In addressing the problem above, one must recognize that there are three different points in the lifetime of a Web document where these concerns arise:

1. **Author Time**: the author of the document typically saves the contents of all the controls directly in the document such that the working document may be entirely self-contained. This is the same idea as "design time" in the original OLE Controls specification, and a container's *UserMode* ambient property is set to FALSE at this time.
2. **Publish Time**: the author has completed creation of the document and wishes to finalize it for publication. This is not so much an operational mode as it is a *transition* between design and "ready-to-run" states. Anyway, at this point the author can break the data for any and all controls out into separate documents/storage locations, assigning to each control the name of its storage location.
3. **Run Time**: a user is viewing the document where the controls must become interactive at some point. Those controls whose data exists at some other location must progressively retrieve their data (using the names assigned at publish time if needed) without blocking the entire process, that is, cooperating with the container. In this state a container's *UserMode* property is set to TRUE.

© Microsoft Corporation,1995-1996, All Rights Reserved.

It is also a requirement of this specification to preserve as much of the existing Windows, COM, and OLE programming model as possible, that is, to avoid creating new technology wherever possible, especially to avoid adding additional properties or interfaces that all controls would have to support regardless of their feature set. This ensures that this specification does not significantly raise the amount of baseline technology for controls, meaning that only controls that need Internet features need implement support for those features at all. It also leverages a great deal of work that has already been done on OLE Controls, both in design and implementation.

NOTE: everything in this document applies to generic COM objects as well as controls, although the term "control" is used most often. In fact, the distinction between controls and generic COM objects is gradually becoming non-existent. Controls are, at the time of writing, now considered to be nothing more than a COM object with the *IUnknown* interface as a minimum, with a whole host of optional interfaces that provide specific features as the control requires (such controls no longer use the "Control" registry key for identity, using instead a new component category). This allows a control vendor to implement only as much code as is absolutely necessary for the control's functionality, placing the small added burden on the container to degenerate gracefully when a certain interface or feature is not available.

Readers are therefore advised to think of "controls" as the term is used in this document as referring to any generic usage of OLE technologies in creating component software. The solutions presented here are extensions to the totality of OLE, not just extensions for the specific model of a complete OLE Control that implements all possible control features. It should also be noted that this document describes only minimal extensions to the OLE model that has succeeded in the marketplace for several years now. As such, this document does not contain any exceptional amount of "new" technology, merely extensions to make the existing technology work great in a new environment.

The extensions in this specification also apply identically, without modification to the new additions to the OLE Controls model that will be finalized in 1996 to support windowless controls.

For more information on this view of controls, see the **Control Guidelines** document listed in the references.

This document specifies functionality that is not necessarily present in Sweeper components as of the date on this document. Differences between this specification and Sweeper components are described in text using the same style as this paragraph.

2 Non-Requirements

This document concentrates on runtime/browsing issues and only contains mention of a few issues related to authoring and publishing phases of development. This section gives some examples of requirements in those domains that are not relevant to the rest of this document.

In addition, exact requirements for design time and publish time considerations are still in flux as there is no solid implementation against which to compare the feasibility of possible solutions. Therefore, within this document, descriptions for functionality in the design and publish time scenarios are nothing more than proposals and do not reflect any existing product.

In any situation where the source and consumer of any data are physically separate, there is the possibility for "broken links" where the consumer loses track of the source. This is a problem when it comes to standard OLE compound documents where an end user may move a source or a consumer document independent of the other, possibly breaking any links between the two. This problem exists because file systems are read-write as far as the user is concerned.

There is ongoing work to solve the link-tracking problem for OLE compound documents where monikers are involved. As solutions become available in that space, they will also become available to any other architecture that employs monikers.

As moniker usage is part of this specification, some degree of link tracking will automatically come into play without any extra work on the part of controls.

Furthermore, for Internet browsing purposes, link tracking is generally a non-requirement because it is assumed that *only* authors/designers of Web pages and Web sites will have the capability to rearrange the relative positions between source and consumer documents. In other words, the ultimate end user sees the Web as a read-only file system. Because of that, it is the author's/designer's responsibility—with the help of an authoring tool built on the system-level link tracking services—to ensure that a document with external links contains accurate specifications of those external links. The same issues apply to hyperlinks, but no solutions are readily available given the widely distributed and heterogeneous nature of the Web. This specification simply doesn't attempt to address link tracking.

This document does offer some assistance to authoring tools for managing external references made within controls. At the present time, this specification doesn't offer any solutions for going into all levels of nested controls where external references may exist at any level. This specification handles a large majority of cases. A complete solution is therefore not a requirement for this specification.

It is also entirely an author/designer responsibility to manage data stored in external locations such that the data can be removed when no documents container links to those sources. For example say someone is authoring a document with a picture control whose data comes from a source specified with a URL moniker. At a later time, someone (the author or another person) deletes that picture control from the document. The author is then responsible to determine if the data source itself should be deleted if there are no controls referencing that data any longer.

It is thus a non-requirement for this specification to offer any solutions to this resource management/reference counting issue. Solutions are the domain of content-management and authoring tools.

3 Control Instantiation

How does a container bring control class code (binary executable code) into memory and instantiate a control of that class? There are two cases of concern here:

4. The control's class code is on the client machine already (or somewhere in the namespace addressable via the registry on that machine). Therefore the document only needs the CLSID of that class code as COM/OLE provides the mapping from CLSID to code bits using the registry. A container generally creates a control directly using the CLSID with *CoCreateInstance*.
5. The control's class code is located somewhere other than the client machine, possibly on another Internet site, and must be brought to the client machine before a control of that class can be instantiated. The exact details of this "Code Download" scenario will be described in future specifications.

This document only concerns itself with case #1 above, as the purpose of "code download" described in case #2 is to essentially install a control on the client machine such that case #1 works. Therefore the next three sections describe the issues for instantiation at author, publish, and run time.

4 Author Time Instantiation

This section is proposal only.

A designer of a document or a page will use the user interface facilities of the authoring tool to decide which object or control to insert into a document, ultimately ending up with a CLSID for the object or control. The authoring tool is responsible for providing the browsing capabilities to locate objects and controls wherever they may be, in the local namespace or on some other Web location. See the user interface guidelines for more information on standards related to browsing available object and control classes.

© Microsoft Corporation,1995-1996, All Rights Reserved.

Once a control's CLSID is known, the authoring tool simply uses standard COM/OLE instantiation means such as *CoCreateInstance, OleCreate,* etc., to instantiate the control. In this case the authoring tool will need to include the control's CLSID when saving the document persistently. Handling other possibilities is still under consideration.

Otherwise the authoring tool treats each control instance in exactly the same manner: calling methods, setting properties, attaching event handlers, specifying layout, etc.

5 Saving a Control at Publish Time

This section is proposal only.

At publish time, the authoring tool must create a document, in HTML or another format, that describes the exact source of each control's class code, as it will be referenced at run-time, in addition to each control's instance data.

If the code is assumed to be on the client machine along with the browser then the authoring tool need only store the CLSID for the control class (the CLASSID attribute in HTML) in addition to the control's properties and persistent data, as described in Section 3 below. Such would be the assumption for standard system-provided control classes as well as implementations that the document will manage in its own code repository (part of a future specification).

If the control's class code will be found at another Internet site at run-time, then the authoring tool must save the URL for that site. This can be done either saving the URL text directly (as the CODE attribute in HTML).

If the document incorporates custom control classes which are on the author's machine at publish time but may not be on each client machine at run-time, then the authoring tool must facilitate the placement of the class code (according to license restrictions) on an appropriate Internet sites, converting the CLSIDs of those the control instances in the document into the appropriate monikers and hen saving the monikers in the document.

In addition to each control instance (CLSID/moniker plus the control's instance data) the authoring tool, at publish time, also saves its own "extended properties" which a control instance itself never deals with, such as Name, Width, Height, Align, etc. Documents stored in their own formats use whatever data formats they desire; documents saved in HTML must follow what's described in **HTML Standards.**

6 Run Time Instantiation

An Internet browsing tool, when it wishes to view a document, will generally activate the necessary viewer for that document type which may be itself (in HTML cases, for instance) or some other view via DocObjects. The viewer then loads the document, recreating the control instances therein according to the prioritization scheme determined at author time.

"Recreating a control" means creating a previously saved *instance* of a control such that some persistent data for that control exists in the document or somewhere else. In the OLE model, instance data always has an associated CLSID—thus the container knows which class code it must have to instantiate an uninitialized control using the CLSID directly (*CoCreateInstance, OleLoadFromStream, etc.*). Once the container creates that instance, it then initializes the instance with whatever persistent data was previously saved through the *Load* member of some *IPersist** interface. Section 3 below covers exactly where this persistent data comes from and which interfaces are involved.

In some cases both the class code for a control and its instance data resides in other locations named with monikers (or URLs, which can be turned into URL monikers as it the case with the CODE attribute in HTML). In this case the container must first retrieve the control's class code, then proceed with creation and initialization as described above.

As with initial creation (see 2.1 above) the container now has an interface pointer to an initialized control which is generally ready for use (depending on the internal state of the control as discussed in Section 5). At this time a container may have other properties in its storage that lay outside the control's own persistent image (such as PARAM fields in HTML) in which case it sends these property values to the control after the control has returned from *IPersist*::Load*.

7 Storage of Control Persistent Data

Note to readers: this section is primarily background material to set the stage for Section 4. Those interested in only the details of writing good Internet-aware controls and containers can skip directly to that section.

For the purposes of this discussion, any given control may potentially have the following types of persistent data as illustrated in Figure 1:

- **CLSID (16 bytes)**: identifies the class code that can read other data that follows.
- **Properties (usually less than 10K-20K bytes):** the control has a set of named values, accessible generally through OLE Automation (that is, vtable interfaces with type information, *IDispatch,* or dual interfaces). These may be standard or custom properties, of course.
- **BLOBs (arbitrary sizes)**: the control has any number of large binary data blocks, each of which exists in any format (standard or custom) and can be arbitrarily large. Bitmaps, videos, sounds, and other sizable data fall into this category where controls are concerned. "Native data" as used to describe the persistent data of a "compound document object" in OLE Documents is another example.

Note: the description of these three elements is for conceptual purposes only. These elements do *not* describe any kind of actual stream, file, or data format.

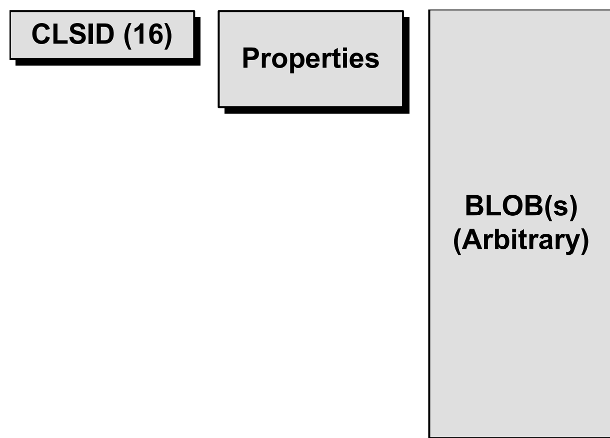


Figure 1: Possible elements of a control's persistent data

There may be controls that have no persistent data in which case *none* of these elements exist in a document. In that case the container has the CLSID of the control (directly from the CLSID attribute in HTML or indirectly from CODE) and the control needs no other initialization after creation.

When a control has any persistent data at all, the CLSID element always exists along with one or both of the other two elements. How these elements are stored in relation to the container document involves the concepts of "embedding" and "linking" which originated in OLE Documents (the OLE compound document architecture) which is reviewed in Section 3.1 below. Because of certain limitations inherent in the compound document model, Section 3.2 extends the "embedding" and "linking" concepts *as persistence mechanisms only* to allow more flexibility in control implementations. Additional sections provide notes regarding the use of *IPersistMoniker,* HTML, and progressive property retrieval.

Many readers will already be asking themselves how a control might be able to split its properties element from its BLOB elements such that the CLSID and properties are stored in one location and the BLOBs in other locations. This topic requires special treatment which is the topic of Section 4.

© Microsoft Corporation,1995-1996, All Rights Reserved.

8 A Review of Embedding and Linking á la OLE Documents

"Object Linking and Embedding" as OLE was originally called in its version 1.0 days, introduced the idea of "embedded compound document objects" and "linked compound document objects." These concepts, defined below and illustrated in Figure 2, carried forward into OLE 2:

6. **Embedding**: All of the object's native data (and CLSID) is completely stored in-line, that is, embedded, within the document itself. The container also stores any container-owned information, such as a presentation cache, with the object's native data.
7. **Linking**: The container still stores any container-owned information and the presentation cache in the document but the object's native data (and CLSID) is stored in an outside location. The location is named with some moniker which is itself serialized in the document (usually the object's CLSID is cached here as well for various optimization purposes). Linked objects also have the user interface restriction that they cannot be in-place activated.

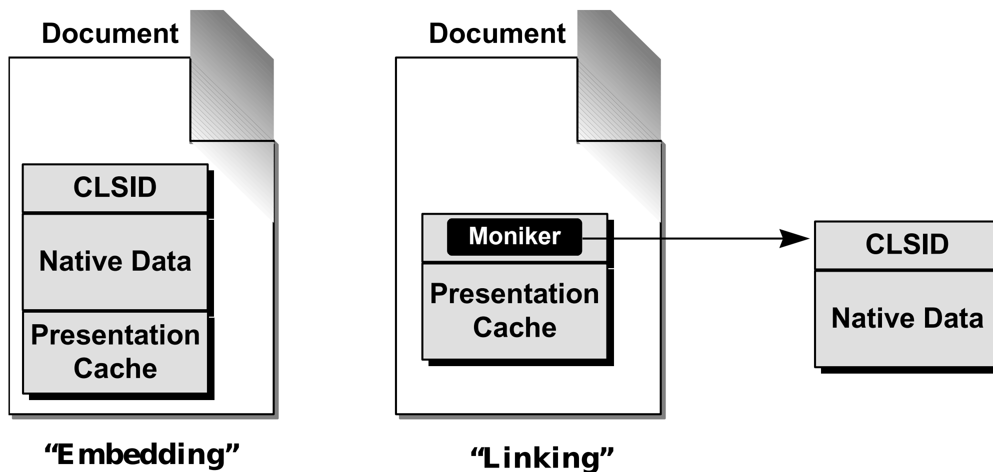


Figure 2: Embedding and Linking as persistence mechanisms in OLE Documents

Because of the demands of the presentation cache, a container in OLE Documents must always provide an instance of *IStorage* to each embedded or linked objects. In the embedding case, the class code for that object must itself implement *IPersistStorage*. In the linking case, OLE provided the "default handler" that handled the moniker and the cache in the container document (through *IPersistStorage*) but the class code itself has to provide whatever interfaces were appropriate for the binding behavior of the moniker. For example, if a File moniker is used, then the class code must implement *IPersistFile* in order to learn from the moniker where the native data resides on the file system.

Note also that an embedded (or even linked) object may internally, in its own native data, store references to external data sources, essentially doing a simpler form of "linking" unbeknownst to the container. The system-provided "Package" object is a perfect example of this. A Package can contain an "embedded" file or a "command line" along with an icon and label (which make up its presentation in the document). When a file is embedded in the package, the file image itself makes up the package's "native" data. When the package contains a command line, the command line is embedded along with the icon and label, but the "native" data is ultimately in some other file in the namespace as described by the command line. We can call this a "link" to that other file, although the link is entirely internal to the package object and is entirely hidden from the container.

While this architecture works fine with in the confines of a single high-speed file system, where storage space is cheap and network latency and transfer speed are usually not an issue, it does not apply all that well to an environment like the Internet. In particular there are the following problems:

- There is almost always some sizable data stored inside the document, usually the presentation cache. Unless the container chooses to eliminate the cache entirely, the document size becomes significant which is undesirable in a slow-speed network.
- An embedded object can only use *IPersistStorage* as a persistence mechanism which is simply too “heavyweight” for many types of controls that store all their properties in only use a few hundred bytes or less, such as simple labels, buttons, checkboxes, group boxes, scrolling marquees, etc. This leads to wasted storage space.
- A link source must generally implement *IPersistFile* to support moniker binding which works only in the domain of UNC path names whereas the Internet requires the ability to use URLs.
- Any external references in the object’s native data are known only to the object which precludes any container participation in managing or assigning those locations nor does it have any chance to help manage the data stored there.

OLE Controls as first introduced in 1994 gave controls the ability to implement the *IPersistStreamInit* persistence mechanisms for the “embedding” case, eliminating the caching issue in the process and providing a lightweight alternative to *IPersistStorage.* However, OLE Controls was not originally concerned with complete flexibility in persistence mechanisms nor alternate forms of “linking” that addressed all of these issues. Such are the topics of the next section with the exception of the last item above, which is covered in Section 4.

9 Persistent Embedding and Linking (*IPersist** Interfaces)

This specification then extends the concepts of embedding and linking to work outside the particulars of OLE Documents (*IPersistStorage,* caching, *IPersistFile*), specifically to work with all available persistence mechanisms as well as with URL monikers and the possibility of asynchronous retrieval of linked data. The concepts of embedding and linking are then as follows, illustrated in Figure 3 as well:

8. **Persistent Embedding**: The object’s/control’s CLSID, properties, and BLOBs are completely stored in-line, that is, embedded, within the document itself.
9. **Persistent Linking:** The object’s/control’s CLSID, properties, and BLOBs are stored in another location identified with some single moniker.

> NOTE: The “linking” architecture of OLE Documents is itself more than just a persistence mechanism as it also involves various user interface standards, such as the stipulation that a linked object cannot be in-place activated. Any user interface guidelines concerned with OLE Documents are not of interest to what this section calls “Persistent Embedding” and “Persistent Linking” as the mechanisms here are solely concerned with the location of data and have nothing to do with user interface models. A control can thus work with persistently linked data while still being in-place activated.
>
> In addition, in OLE documents the idea of “linking” generally means that the source itself (of the linked object) supplies the exact moniker to name the object. In “persistent linking” the container provides the moniker to the object and the object binds that moniker to some piece of storage in which the object reads or writes its data.

© Microsoft Corporation,1995-1996, All Rights Reserved.

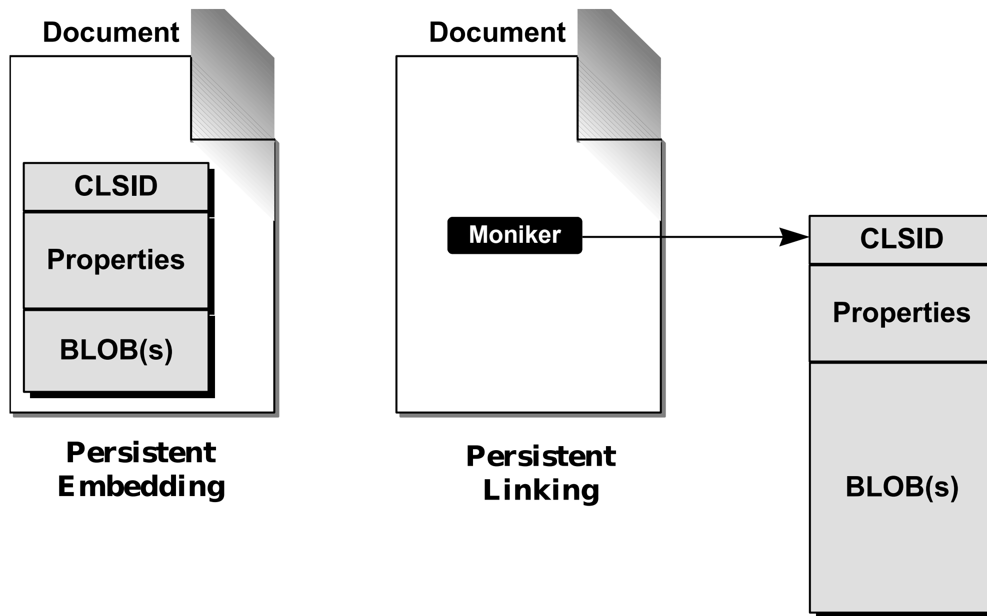


Figure 3: Persistent Embedding and Persistent Linking

Again, Section 4 covers the case where one or more of the control's properties name other external storage locations such that the BLOBs do not have to be stored with the control's properties in either embedding or linking case. For the purposes of this discussion we assume that any in-line BLOBs in the embedding case are still small such that the byte count of the properties and the BLOBs still fall within a reasonable size. One can simply see these small portions of binary data as additional control properties.

The extensions that enable these generic persistent mechanisms are as follows:

10. A control may use any of *IPersistStreamInit, IPersistStorage, IPersistMemory,* and *IPersistPropertyBag* as a **Persistent Embedding** mechanism. A control can choose to support as many of these as it desires.
11. A container that wishes to use **Persistent Embedding** must be able to provide controls with storage space appropriate for the preferred interfaces on the control: *IStream* for *IPersistStreamInit, IStorage* for *IPersistStorage,* memory (*void* *) for *IPersistMemory*, and *IPropertyBag* for *IPersistPropertyBag*. Containers will determine what method is used to save an control initially (according to its own preference for these methods) and expect that a control can be reinitialized through the same means.
12. A control may use any of *IPersistStreamInit, IPersistStorage, IPersistMemory,* and *IPersistFile* as a **Persistent Linking** mechanism, again choosing to support as many interfaces as desired (*IPersistMoniker* is also an option although the feasibility of this interface is questionable at this time). It is strongly recommended that Internet-aware controls implement at least one other interface than *IPersistMemory* and *IPersistFile* because both of these require that all the data exists locally which is unsuitable for use with asynchronous data transfers.
13. A container that wishes to use **Persistent Linking** must use a moniker that supports the possible persistence mechanisms that a control might support. In the immediate term this is limited to the URL moniker.
14. A container is allowed to make a copy of a control's persistent data at any time through any *IPersist*::Save* member functions regardless of which *IPersist** interface was used to initialize the control. When doing so, the container must not pass TRUE as the *fRemember* argument of *IPersistStorage::Save, IPersistMoniker::Save,* and *IPersistFile::Save* and must pass FALSE as the *fClearDirty* argument to *IPersistStreamInit::Save, IPersistPropertyBag::Save,* and *IPersistMemory::Save*. The container must also call the *SaveCompleted* members of *IPersistStorage*

and *IPersistFile* (and perhaps *IPersistMoniker*) with NULL arguments after calling their *Save* members.

The following table summarizes all these *IPersist** interfaces:

Storage Location	Persistence Interface	Comments
Storage Element	*IPersistStorage*	Standard in OLE Documents; container provides an *IStorage* pointer to the storage element in which the control can create any structure it desires. This will be potentially common for author-time scenarios, typically rare for publish/run time scenarios.
Expandable Stream	*IPersistStreamInit, IPersistStream*	Most suitable for small and fully embedded controls; all the data including paths goes into one stream which can be easily placed inline in the document. *IPersistStreamInit* is a superset and replacement for *IPersistStream*. *IPersistStreamInit* should be used preferentially.
Fixed-Size Memory Block	*IPersistMemory*	Alternative for *IPersistStreamInit* but allows the container to specify a fixed-size memory allocation as the storage medium with the restriction that the control does not attempt to access data outside that boundary.
“Property Bag” (container-supplied)	*IPersistPropertyBag*	Alternative for *IPersistStreamInit* in which the control tells the container to save and load individual properties (described in a *VARIANT*) through *IPropertyBag*. The implementor of the property bag can deal with each property in any way it wants.
File	*IPersistFile*	The object is given a UNC path name and is told to read or write its data to that file.
External: named with a moniker	*IPersistMoniker*	The object is given a moniker and told to read and write its data to whatever storage mechanism (*IStorage, IStream, ILockBytes, IDataObject*) it desires when dealing with that external data. The storage mechanism may also be asynchronous in which case the *IPersistMoniker* implementation understand the necessary considerations.

IPersistMoniker is a useful design but may not be feasible to work with in the Internet environment for a number of reasons, primarily that in asynchronous cases a download of the data has already begun before the object receives a call to IPersistMoniker::Load, in which case it really can’t control the storage mechanism that is used. See Section 3.3

For details on *IPersistMemory*, *IPersistPropertyBag*, and related interfaces see Section 9. For details on *IPersistMoniker* see **Asynchronous Monikers**.

These extensions are basically concerned with the protocol through which a container communicates persistence intentions (*IPersist*::Load* and so on) to controls in its document, which also addresses the needs of author, publish, and run time scenarios. At author time, the designer may want to keep all the data for all controls in-line so there is only one document to manage, or some controls may be told to save and load their data in another site already. At publish time the author might wish to store the bulky portions of data (controls with large BLOBs) in other locations, thus breaking the document up into its final distribution on the Internet. At run-time, then, the controls in the document will access their data using as instructed to do so by the container.

An Internet-aware object or control is one that understands the options available here and implements support for whatever mechanisms it needs and can support best. Such a control need not support all storage cases and can choose to only support those it sees fit to support or can support reasonably well. As a minimum, controls that have *any* persistent data should implement at least *IPersistStreamInit* **or** *IPersistStorage* (whichever is most suitable), adding other interfaces as features

© Microsoft Corporation,1995-1996, All Rights Reserved.

demand them. For example, controls whose persistent data is made up entirely of name/value property pairs will likely implement *IPersistPropertyBag* as well). For more details on why a control might choose to implement *IPersistMoniker*, see Section 3.3 below.

If a control for some reason *only* implements *one* persistence interface out of *IPersistStreamInit, IPersistStorage, IPersistMemory,* and *IPersistPropertyBag*, it **must** mark itself with the appropriate component category to say that support for that interface in a container is mandatory. See Section 6.1 for more details on these categories. Controls that implement *IPersistStorage* and can work in down-level containers as simple compound document embeddings can mark themselves with the "Insertable" registry key such that they appear in the Insert Object UI of such containers (a category is not used since down-level containers don't see categories).

Note that it takes less than 200 bytes to persistently save values for *all* of the standard properties defined for a full-featured OLE Control, such as text, caption, colors, drawing styles, and the font; if the control wishes to include a small metafile for immediate rendering, adding a few hundred bytes more bytes should not be a big problem

On the other side of the picture, containers should support—for **Persistent Embedding**—as many different persistence interfaces as reasonable with *IPersistStreamInit* and *IPersistStorage* as a baseline, adding support for *IPersistMemory* for optimization purposes and *IPersistPropertyBag* for Save-As-Text capabilities. Authoring tools and containers must also pay attention to component categories when it does not implement support for any one interface such that it can avoid inserting a control into a document when its persistence needs cannot be met.

In this **Persistent Embedding** case as well, the container chooses which interface it will attempt to use first with any given control. Some containers may look for *IPersistStorage* before *IPersistStreamInit*; others will try *IPersistPropertyBag* before all others, or will place *IPersistMemory* before *IPersistStreamInit* as the former might be more efficient for the container. Because the size of embedded data is of high concern with Internet-aware containers, *IPersistStreamInit* and *IPersistMemory* should generally be given priority over *IPersistStorage* and *IPersistPropertyBag* as they generally produce the smallest amount of data.

Note that it is perfectly reasonable for a container to have a control save its data into any location specified in any of these interfaces after which the container copies the resulting binary data to another location altogether. The container then saves a reference to that other location such that when loading the control it can recreate the necessary storage structure and hand it back to the control. This option, by the way has always been available in OLE since version 1.0.

For **Persistent Linking c**ontainers need not be concerned with the interfaces directly but must understand its own usage and storage of monikers, specifically URL and other asynchronous monikers. Because the monikers themselves internally query for and call the various *IPersist** interfaces, the container does not have to understand those persistence interfaces directly. Such intelligence is encapsulated in the monikers.

In this case the moniker itself chooses the priority as the container merely needs only to choose how it will save the moniker. As described in **Asynchronous Monikers** such monikers will look for interfaces in the order of *IPersistMoniker, IPersistStreamInit, IPersistStorage, IPersistMemory*, and *IPersistFile*, the latter two requiring that all the data is available before any members of either interface can be called.

Use of IPersistMoniker is questionable and has limitations. See Section 3.3.

In both embedding and linking cases, the container is responsible for handling the details of asynchronous storage as described in "Compound Files on the Internet" and "Asynchronous Monikers Design Specification." In short, all persistence interfaces other than *IPersistMoniker* are considered **synchronous** in that the control expects all the data to be available when it gets a call to *IPersist*::Load*. In the case of *IPersistFile* and *IPersistMemory* this is explicit because you can't create the file or the memory block unless you have the data.

In the case of *IPersistStream[Init]* and *IPersistStorage*, a container can pass either a "synchronous" storage object or a "blocking" storage object. In the "synchronous" case the container will retrieve all the data before calling *IPersist*::Load* so nothing has changed from all historical uses of these interfaces. In the "blocking" case, the data might not actually be

available, but any call the control makes to *IStorage::OpenStorage, IStorage::OpenStream,* or *IStream::Read* (and so on) will simply not return until the data is actually available. From the control's point of view, the storage or stream objects are simply slow—the control won't usually care since it just waits for the call to return. But from the container's point of view, simultaneously loading multiple controls using asynchronous "blocking" storage may be a perfect way to manage multiple data transfers.

10 Use of *IPersistMoniker*

> *This section is proposal only—at the time of writing there exists no working implementation of this interface. Controls have no need to use this interface at this time.*

As described in Section 3.2, the *IPersistMoniker* interface is the primary interface through which an asynchronous moniker will attempt to have a control initialize in the **Persistent Linking** case. In general, *IPersistMoniker* is the successor of *IPersistFile* that allows persistence to and from any abstract location that can be named with a moniker as opposed to only a filename. The interface is defined as follows (see **Asynchronous Monikers** for complete details):

```
interface IPersistMoniker : public IPersist
    {
    HRESULT IsDirty(void);
    HRESULT InitNew([in] IMoniker* pmkStore, [in] DWORD grfMode
        ,[in] IBindCtx* pbindctx);
    HRESULT Load([in] IMoniker *pmkStore, [in] DWORD grfMode
        , [in] IBindCtx *pBindCtx);
    HRESULT Save([in] IMoniker *pmkStore, [in] BOOL fRemember
        , [in] IBindCtx *pBindCtx);
    HRESULT SaveCompleted([in] IMoniker *pmkStore);
    HRESULT GetCurMoniker([out] IMoniker **ppmkStore);
    }
```

In short, this interface is just like *IPersistFile* (plus the *InitNew* member) except that where *IPersistFile* takes a filename string, *IPersistMoniker* takes a moniker and a bind context. The semantics of the *Load* and *Save* members are as follows:

```
pmkStore->BindToStorage(pBindCtx, …, IID_<xxx>, &pInterface);

if (load)
    pInterface->[Read](…)   //May be asynchronous
else if (save)
    pInterface->[Write](…)
```

That is, the control is asked to save or load itself from a moniker using the supplied bind context, and the control is responsible to call *IMoniker::BindToStorage* to get the appropriate storage-related interface to which reads (or writes) its data. This may involve asynchronous considerations as described in the **Internet Files** and **Asynchronous Monikers** documents. When the control obtains an asynchronous *IStorage* or *IStream*, it has to ensure that it can handle (a) *IMoniker::BindToObject* returning a NULL interface pointer that will be passed to *IBindStatusCallback::OnDataAvailable* when it becomes available, (b) *IStream::Read* and other reading calls that might return E_PENDING, and (c) data that becomes available in segments through *IBindStatusCallback::OnDataAvailable*. The considerations here are the same as for any client of an asynchronous monikers, so again, see **Asynchronous Monikers** for more details. Section 4 also discusses this to some extent in terms of a control's use of "data paths."

Thus implementing *IPersistMoniker* is not a requirement, even for asynchronous transfer of persistently linked data that goes on outside the control before any of its other *IPersist*::Load* members are called. This is because asynchronous monikers themselves (that is, the URL moniker presently) will take whatever data it sees at the named location and package that data in some storage object appropriate for other *IPersist** interfaces. That is, if the moniker does not find *IPersistMoniker* on the object, then it will query for *IPersistStreamInit*. If that interface exists, the moniker will wrap up the data in an asynchronous-blocking *IStream* and pass it to the control. If the interface is not there, the moniker will try other interfaces like *IPersistStorage, IPersistMemory,* and *IPersistFile* in turn, wrapping the data in an asynchronous-blocking *IStorage* if needed, or retrieving all the data and placing it in memory or a file before handing it to the control.

© Microsoft Corporation,1995-1996, All Rights Reserved.

If this design turns out to be feasible, there are only a few reasons why anyone actually would need to use IPersistMoniker. Those reasons, however, are not useful unless this design is feasible, thus they are not included here.

11 Relationship to the HTML OBJECT Tag

It is useful at this point to describe the relationships between the persistent embedding and persistent linking scenarios described above and the attributes described in **HTML Standards**, namely the CLASSID, DATA, and PARAM attributes within an OBJECT tag.

In the persistent embedding case the authoring tool that is writing the HTML document first asks the control to save all of its data through some persistence interface. After the *IPersist*::Save* call, the authoring tool has some bucket of control-created bits and the CLSID for the control itself. When writing the HTML OBJECT tag, the authoring tool always writes a CLASSID attribute, then includes a DATA attribute describing the control's data in some way. In all cases, the data must have a CLSID in the first 16 bytes in order to identify the exact format of the data.

In some cases the stream data might be too large in which case the authoring tool might use *IPersistPropertyBag* to be handed each property in turn which it would then write into the HTML document as individual PARAM attributes. Other PARAM attributes might also be later added manually to this set by someone editing the HTML directly.

The MSHTML control currently has a bug whereby it may call IPersistStreamInit::InitNew followed later by IPersistPropertyBag::Load. This occurs if there is no DATA attribute in the OBJECT tag.

12 Data Path Properties

The impetus for much of this work in this document finds its origins in the concept of "progressive rendering" ("progressive disclosure") for something like a picture or a picture control. In such cases the picture should be able to display an initial low-detail picture, followed by mid-detail and high-detail renderings.

This specification is, however, concerned with the more general problem of retrieving any significant amounts of data in a cooperative fashion from possibly many distributed locations *after* the control has already been instantiated and is possibly interacting with the container and the user in other ways. This capability has been called "Progressive Rendering," "Progressive Property Disclosure," and "Progressive Downloading." The term "Progressive Data Retrieval" will be used to refer to all of these more special cases at once, because it makes no assumptions about the exact type of data that is being retrieved—might be properties, might be images, might be anything.

In all the following sections, it is *very* important to *keep in mind that the control is ultimately responsible for specifying the exact format and structure of any linked data and decides exactly how it will be pulled from the named source*. The container has nothing to do with this other than being able to control the relative priority of the retrieval and being notified when the loading state of a control changes. So, for example, a container doesn't have to care about how a picture control chooses to progressively render low-, mid-, and high-detail images (from a GIF, for instance). The picture control would internally know where and how each image is stored in its data, and retrieves those in turn. The container would then be notified through *IAdviseSink::OnViewChange* when the control has enough data to do something meaningful in *IViewObject2::Draw* (which is all moot for an in-place active control whose window is visible since the control would just update itself, but this will be important later on for windowless controls).

The point of all this is that the container chooses *where* to tell controls to store and retrieve their data; the controls choose *how* to store and retrieve that data in a cooperative, asynchronous, and progressive manner. The asynchronous aspect may apply to none, some, or all of a control's data.

This section is thus about “Data Path Properties,” and describes the architecture for having a control save and load pieces of its data (BLOBs especially) in locations other than where the control’s properties are kept. That is, For any number of reasons it may be inappropriate or impossible for a control to store absolutely all of its data in the same location as its properties. For example:

- The control is being embedded and its BLOB data is too large to store in an embedded manner, such as in an HTML document.
- The control wants to retrieve BLOBs in an asynchronous manner regardless of where the control’s properties are stored. A video control is a good example where this would be important.
- The control may be sharing BLOBs with other controls such that it is inappropriate and/or wasteful to duplicate the BLOB in each control’s persistent state (this is similar to the original linking concept in OLE Documents). Sharing a company logo across many Web pages is a good example, and excellent performance can be achieved in this case when a URL cache is in operation—the BLOB itself may have already been downloaded from another page and can simply be pulled from the cache immediately.
- The control may want to allow BLOBs to be modified such that whenever the control retrieves the data it can retrieve the most recent information. A weather-map control that always shows the latest satellite image is a good example.
- The control may simply want to separate properties from BLOBs for personal reasons.

To address these particular concerns, a control may use one or more “Data Path Properties” to reference external sources of data. As illustrated in Figure 4, Each “path” (as they’re called for short) is a value of some kind which are stored like any other properties in the control’s persistent data. When the control needs to retrieve data from a source, it has the container create a moniker from the string representation of the property value and calls *IMoniker::BindToStorage* to retrieve some sort of storage pointer (usually *IStream* but not limited to that). In binding a URL moniker the control can choose whether the binding itself will be synchronous or asynchronous, and whether or not the data transfer itself will be synchronous or asynchronous.

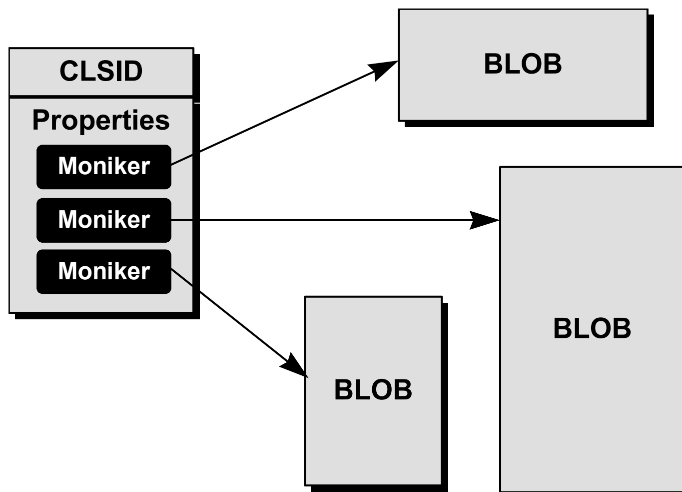


Figure 4: Data Path Properties are values which can be parsed into monikers that reference sources of data external to the control’s embedded properties

In short, each path property is a control-managed “link” to an outside data source. As described in an earlier section, this capability has always been available in OLE Documents, except that such links were always internal to the control. Such

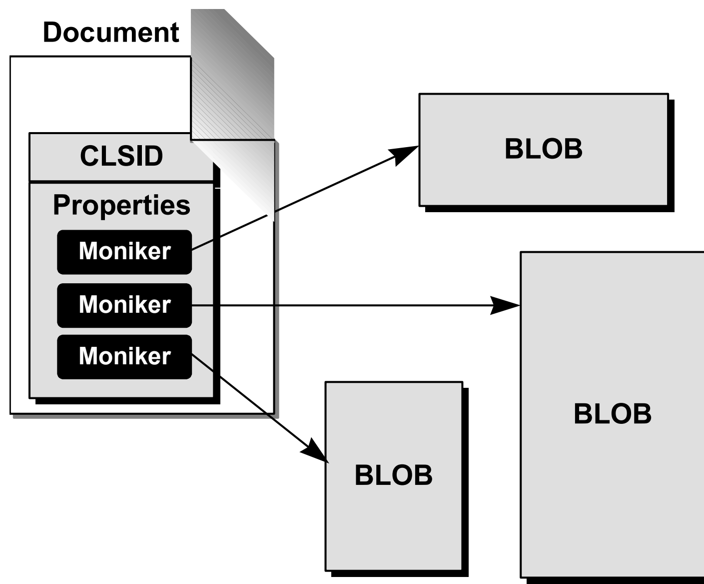
© Microsoft Corporation,1995-1996, All Rights Reserved.

hidden use of external resources is not acceptable in the Internet environment where authoring tools need to manage the interconnections between many Internet sites.

Data Path Properties therefore address the following concerns:

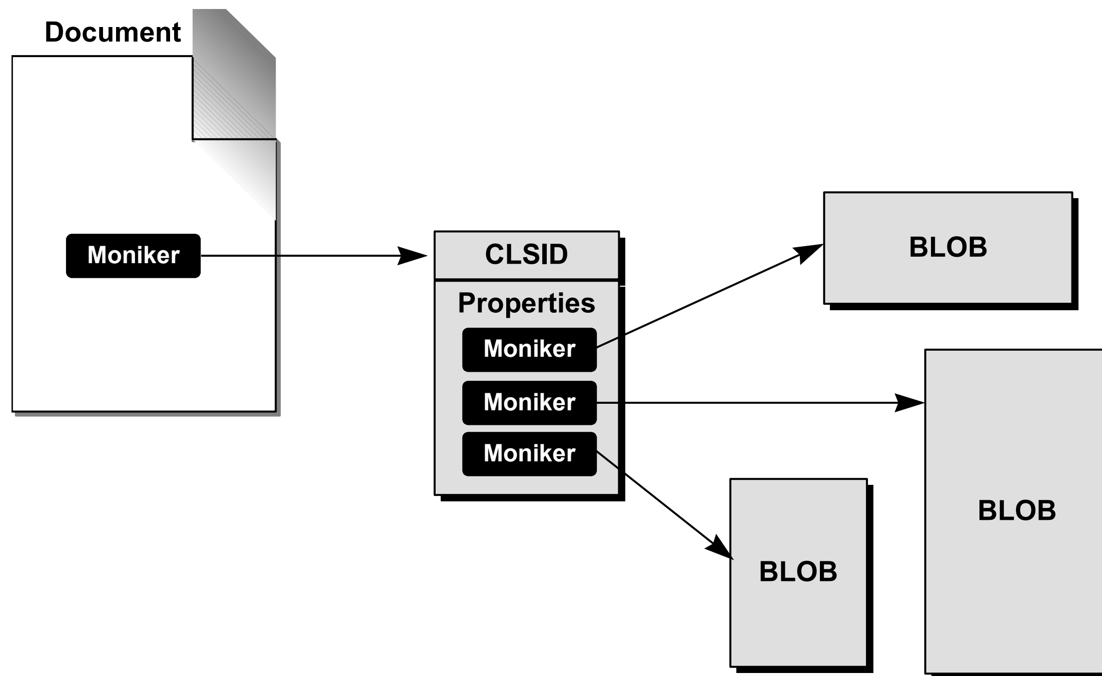
- How to expose the existence of path properties from a control
- How to specify exactly what type of data a particular path should refer to such that an authoring tool can provide appropriate browsing UI or automatically assign a path property referring to the correct format of data
- How to assign values to path properties, which may occur through either the authoring tool (automatically or manually) or the control itself (manually through its own property pages).
- How to save each path property in the control's persistent data
- How to cooperatively retrieve data referred to with a path property which might include container participation
- How to communicate a control's "readiness" to a container at run-time such that the container can manage dependencies on the control's internal state.

Linking capabilities using path properties are equally usable in both the Persistent Embedding and Persistent Linking cases as described in Section 3.2. For clarity, the combinations of Persistent Embedding and Persistent Linking with Data Path Properties are illustrated in Figures 5 and 6.



Persistent Embedding with Data Paths

Figure 5 A control that uses data paths is embedded in a document but each path references an external source of data



Persistent Linking with Data Paths

Figure 6: A control that uses data paths is linked to a document and itself contains addition paths that reference external sources of data.

With data paths, a control can allows its persistent data to be distributed around a file system or the Internet. For example, a sophisticated motion-picture control may separately reference its AVI data, its audio data, and a lengthy text transcript of the dialogue. The control could then have *MoviePath, AudioPath,* and *TranscriptPath* properties. In a similar manner, a picture control might store three separate detail images and use the properties *LowDetailImagePath, MediumDetailImagePath,* and *HighDetailImagePath* (and maybe the motion picture control uses these three as well to provide progressive rendering of a static image). Anything is possible. This specification places no restrictions on the exact data formats stored in locations given to path properties—the control ultimately determines the formats it can employ.

The following sections provide the details of Data Path Properties, starting with how a control exposes its path properties and the data formats associated with those paths. This is followed with a discussion on the assignment of path values, how a control saves a path property, and how a control uses a property to retrieve data asynchronously. Section 5 describes how the control communicates its "readiness" to a container and how the container handles different readiness states.

13 Exposing Data Path Properties and Corresponding Data Format

Earlier versions of this specification along with a preview article that appeared in Microsoft Systems Journal described path properties as a GUID-types BSTR. After review with the appropriate groups involved in designing authoring tools, the GUID-typing scheme has been replaced with the custom-attribute means of identifying any property of any type as a path. The new proposal is higher performance and easier for authoring tools to accommodate.

The following characteristics apply to publicly exposed path properties and controls that provide them:

15. A path property can be a property of any type, most commonly a *BSTR* (allowing easy manipulation through Visual Basic code such as *MyObj.TranscriptPath*) but other types are allowed, such as *IUnknown* or *IDispatch*. Choice of the exact property type is the decision of the control developer.

© Microsoft Corporation,1995-1996, All Rights Reserved.

16. All path properties regardless of type are tagged with a special custom attribute GUID that identifies the property as being a "path." This simplifies the process for an authoring tool to find path properties with a quick check through the control's type information. This standard GUID is defined as *GUID_PathProperty* with the value of *{0002DE80-0000-0000-C000-000000000046}*.
17. The custom attribute described in #2 can express one or more MIME type values to describing the actual data type that this path should reference. The syntax is the same as that for the **Accept:** fields of HTTP headers (see below, and see the HTTP specification for complete details).
18. A path property can be assigned any *dispID* as the control developer sees fit.
19. A control marks each path property with the **[bindable]** attribute. This is because a container may allow path assignment through its own user interface and may be displaying path properties in its own property browser. **[bindable]** is thus necessary to keep the property browser synchronized with the control's state. With this attribute, the control must call *IPropertyNotifySink* methods if the container has connected an implementation of that sink interface through the control's connection points. Controls can choose to mark path properties as **[requestedit]** and **[displaybind]** if desired, but these are not requirements.
20. The control's **coclass** entry in its type information is tagged with another special custom attribute GUID that specifies "this control has one or more path properties." This standard GUID is defined as *GUID_HasPathProperties* with the value *{0002DE81-0000-0000-C000-000000000046}*. The value of this attribute is the number of path properties that exist in the control. This attribute thus allows authoring tools to quickly and efficiently determine if a control has path properties at all and quickly describes how many path properties exist.

The HTTP **Accept:** syntax is summarized as follows:

```
<field>  =  Accept: <entry> *[ , <entry>
<entry>  =  <content type> *[ ; <param> ]
<param>  =  <attr> = <float>
<attr>   =  q / mxs / mxb
<float>  =  <ANSI-C floating point text representation>
```

as in:

Accept: text/plain, text/html, image/*
Accept: text/x-dvi; q=.8; mxb=100000; mxt=5.0, text/x-c

The "custom attribute" feature of type information is supported only in the updated OLEAUT32.DLL that is included with the Sweeper SDK. An unofficial version of MKTYPLIB.EXE is included in the SDK to compile ODL files written to use custom attributes (see below for syntax). Note, however, that this version of MKTYPLIB will not be shipped "officially" at any time in the future. The MIDL compiler that will ship with the Win32 SDK for Windows NT 4.0 will be the official replacement of MKTYPLIB as it now handles both ODL and IDL files and can produce type libraries in addition to interface marshaling code.

Previous versions of this specification defined GUID-types such as OLE_DATAPATH_BMP for typing data path properties and proposed an interface called IProvideClassInfo3 to simplify manipulation of paths. Both of these proposals have been replaced with the current specification. However, because this specification requires new OLEAUT32.DLL features, the Sweeper SDK headers that are provided with this document still container a definition for OLE_DATAPATH as a GUID-typed alias for BSTR. For the time being, controls in development can use OLE_DATAPATH as a property type that still provides a container a way to determine that the property is a path of some sort. The WebImage sample control shows how to use this. The files datapath.h, datapath,idl, and datapath.odl define OLE_DATAPATH and

other obsoleted interfaces for the reasons of backwards compatibility. When good authoring tools become available that exploit the specification using custom attributes, controls that use OLE_DATAPATH should be updated as such tools are not expected to work with OLE_DATAPATH.

A "custom attribute" as now supported in type information is defined using the keyword **custom** with a (GUID, value) pair as arguments for the attribute. For example:

```
#define GUID_PathProperty 0002DE80-0000-0000-C000-000000000046

[id(1), bindable, displaybind, propget, custom(GUID_PathProperty
    , "text/plain; q=0.5, text/html, text/*")]
    BSTR TranscriptPath(void);
```

defines the *BSTR* property *TranscriptPath* as path property with the types of plain text, HTML text, and other text (using HTTP **Accept:** syntax). This information is retrieved from the type information through the new type library interface *ITypeInfo2* as described in the next section.

As an example of defining path properties in this way, consider a control that has an AVI *MoviePath* property, a RIFF *AudioPath* property, and an ANSI text *TranscriptPath* property would include them in its type library this way (in ODL syntax; exact MIME types have been omitted since the code below is demonstrating the definition of multiple properties):

```
#define GUID_PathProperty      0002DE80-0000-0000-C000-000000000046
#define GUID_HasPathProperties 0002DE81-0000-0000-C000-000000000046


[<attributes>]
library
    {
    importlib <path to header that includes GUID_PropertyPath and GUID_HasPathProperties>

    [<attributes>]
    interface IMyObjectProperties
        {
        //Other stuff here…

        [id(1), bindable, displaybind, propget, custom(GUID_PathProperty, "…")]
            BSTR MoviePath(void);

        [id(1), bindable, displaybind, propput, custom(GUID_PathProperty, "…")]
            void MoviePath([in] BSTR pathVideo);

        [id(2), bindable, propget, custom(GUID_PathProperty, "…")]
            BSTR AudioPath(void);

        [id(2), bindable, propput, custom(GUID_PathProperty, "…")]
            void AudioPath([in] BSTR pathAudio);

        [id(3), bindable, propget, custom(GUID_PathProperty, "…")]
            BSTR TranscriptPath(void);

        [id(3), bindable, propput, custom(GUID_PathProperty, "…")]
            void TranscriptPath([in] BSTR pathText);

        //Other stuff here…
        }


    [<other attributes>, custom(GUID_HasPathProperties, "3")]
    coclass MyObject
        {
        interface IMyObjectProperties;
        ...
        }

    }
```

© Microsoft Corporation,1995-1996, All Rights Reserved.

14 Author-Time Discovery of Path Properties

Authoring tools and other containers will be interested in knowing exactly which control properties are, in fact, data path properties as well as the type of data that should be referenced through each property. This allows the authoring tool to effectively enumerate data paths and retrieve their current values which is useful when the authoring tool needs to update any such properties to perform link management.

First, to determine if any particular control has *any* path properties, the authoring tool should:

21. Retrieve the control's type information for its **coclass** either through *IProvideClassInfo::GetClassInfo* or from the type library directly. These methods result in an *ITypeInfo* interface for that part of the type library.
22. Call *ITypeInfo::QueryInterface(IID_ITypeInfo2, &pITypeInfo2)* to retrieve the interface necessary to read custom attributes. If this interface does not exist, the container has to assume that no path properties exist.
23. Call *pITypeInfo2->GetCustData(GUID_HasPathProperties, &va)* to retrieve the value of the custom attribute into the *VARIANT* in *va*. If this function fails, or if the value in *va* is empty or zero, then the control has no path properties.

Having this knowledge in hand, the authoring tool can then enumerate the properties in any of the control's incoming interfaces and check if those properties are data paths using the following procedure. These steps assume that *pTI2* is the *ITypeinfo2* pointer to one of the control's **interface** or **dispinterface** entries (available through the *ITypeInfo* of the control's **coclass**):

24. Get the count of the functions in the interface using *pTI2->GetTypeAttr* which returns a TYPEATTR whose *cFuncs* field has the number of function. Call *pTI2->ReleaseTypeAttr* when finished.
25. Iterate in a for loop where the index counter (*index*) begins at zero and counts to *cFuncs-1* (*cFuncs* is from step 1) through the functions.
26. For each function, call *pTI2->GetFuncCustData(index, GUID_PathProperty, &va)* to attempt to read the custom path property attribute's value into the *VARIANT va*.
27. If the *GetFuncCustData* call fails, or if the contents of *va* are empty, then this property is not a data path. On success, the property *is* a data path and the value in *va* will be the description of the MIME type.
28. To determine the dispID of the property, call *pTI2->GetFuncDesc* and look in the *FUNCDESC* structure for the *memid* field, which is the dispID. Be sure to call *pTI2->ReleaseFuncDesc* when done with the *FUNCDESC* structure.
29. For optimization purposes, one can count the number of path properties found so far and exit the loop early if that number matches the value that was read from the *GUID_HasPathProperties* in the steps used to determine if the control had any paths.
30. Once the loop is complete, the authoring tool will have a list of dispIDs (which may have some duplicates) for all the control's data path properties. The authoring tool can use *pTI2->GetNames* with these dispIDs to retrieve text names for those properties.

What then does a tool do with such knowledge of paths and their data types? There are three general scenarios:

31. If the authoring tool is interactive, it might be able to display data browsing UI based on the data format associated with any given path property. That is, if the tool knows that a path property uses the MIME type given in the value of the *GUID_PathProperty* attribute and that tool has a UI that can browse for the that type, then the tool can enable a "…" button in its property browser UI that invokes a File/Open

type of dialog when pressed. The user then selects a file and the name of that data file is assigned to the path property.

32. If the authoring tool is automatic, it uses other available information (say a data repository) to determine the source of any particular piece of data. Knowing the format that the control needs for a data path allows such a tool to find some appropriate data, create a path name for it, and assign that name to the control.
33. When the authoring tool publishes a document to an Internet site it will want to check for any dependencies on outside data—that is, it will want to check for any control properties that reference data outside the document. The authoring tool can then check if that path will be valid at run-time and if not, it can choose to relocate the data to an accessible location and assign a new path to the control before saving that control. This is necessary, for example, when a control references a file on the local file system when the document is being placed on an Internet site.

Of course, the sophistication of the authoring tool is almost limitless. With the tagging and MIME-typing of data path properties, this specification guarantees that any such tool can always determine the data format associated with a path. It's then a matter of whether the tool recognizes that GUID.

15 Assigning Path Properties

The values of path properties—the strings that identify storage locations—are assumed to be assigned only at author-time. In this case there are two possibilities:

34. The authoring tool obtains the name from the user or creates it automatically, assigning it to the control through a "set property" operation (via dispinterface or dual interface depending on the nature of the control). Note that if a transfer is already underway for this property, the control must abort the transfer as described in Section 4.4.1 below.
35. The control obtains the name itself from its own property page UI, in which case it makes the change unilaterally. Any path properties marked with **[requestedit]** means the control is required to call any connected *IPropertyNotifySink* interface's *OnRequestEdit* function to decide if the change is actually allowed. If *OnRequestEdit* returns *S_FALSE* then the control must not change its path in this way (the authoring tool is essentially saying that the document or Internet site state is read-only as far as document/page contents are concerned).

In both cases, a change in the value of any path property will generate a call to any connected *IPropertyNotifySink* interface's *OnChanged* member, since path properties are marked with **[bindable]**. In both cases an authoring tool that displays its own property browsing UI would update itself to reflect the current state of the control. This also gives the authoring tool the chance to modify the path when the user changes it through the control's property pages directly. When the user makes the change, the control will notify the authoring tool through *OnChanged*. The authoring tool then reads the new value of the path property, modifies it as necessary, and sends the modified value back to the control. Therefore in both cases #1 and #2 above the authoring tool has the final say in the exact value assigned to any path property.

The most preferable case is where the value is simply a container-relative name. Relative filenames and URLs are identical in form: "../pictures/tree.bmp" serves equally well in both local file system and Internet environments. Relative item names, like "!Picture 6" identifies the path as pointing to some other part of the same container document. In URL cases the name may be absolute which indicates that there's no relative path between the container document and the location described in the data path.

In any case, a control must always obtain fabricate some moniker from a data path name in order to bind to the storage so referenced. Because the name may only be relative, the container is always given the chance to create the actual moniker itself. The details of this are covered in the next section.

Use of relative names allows an authoring tool to move the root document—and possibly an entire site—around, without invalidating any external links stored in data paths. For example, consider the author-time case where the document being

© Microsoft Corporation,1995-1996, All Rights Reserved.

created is stored in "c:\pages\mypage.doc" and in that document is a bitmap control with an *ImagePath* property. The authoring tool assigns the name "frog.bmp" to this property. When the control wishes to access the bitmap named in the *ImagePath* property, it passes "frog.bmp" to the authoring tool and asks for a moniker in return. The authoring tool sees two filenames and can combine them into one "c:\pages\frog.bmp" name. The authoring tool then returns this name in a moniker such that the control can bind that moniker and retrieve the data.

Now when the author saves the document to an Internet site at Publish time, the location of the document changes to something like "http://www.bogons.com/bogazoid/mypage.htm". In addition, the authoring tool copies "frog.bmp" to that same site where it becomes "http://www.bogons.com/bogazoid/frog.bmp". At run-time, the control reloads the path property name "frog.bmp" and asks the container to make it a moniker. The container simply combines the document URL with the relative name "frog.bmp" to generate the exact URL for the bitmap.

Now in some cases the control may have obtained an absolute filename (or URL) from its own property pages at author-time. So, for instance, the user types in "c:\pages\frog.bmp" instead of just "frog.bmp." Initially the control will internally save the absolute string, but because it sends an *OnChanged* notification (data path is **[bindable]**), the authoring tool retrieves this new value and computes the relative path to that same file from the document's current location (using the file moniker's *RelativePathTo* member, for example). The relative path in this case will end up as just "frog.bmp" which the authoring tool simply assigns back to the control.

In the case where the user gives a control an absolute URL, there's nothing the authoring tool can do at author-time when the document it stored on a local file system. At publish time, however, the authoring tool first determines the new URL of the document. It can then cycle through all known path properties of each control before it saves that control. If the authoring tool finds an absolute URL in a property, that tool can attempt to compute the relative path between the document's URL and the path's URL. Once it has computed this relative path it can then assign the relative name to the path property and proceed with saving the control.

When the authoring tool supports relative path names that are not either filenames or URLs, but are instead item names that identify a piece of storage in the document itself, such as "#Picture 6," the authoring tool must specify, in its own documentation, what prefix character marks the name as an item of some kind. This is a note to the container's moniker creation mechanism in *IBindHost::CreateMoniker* as described in the next section.

16 Storing and Comparing Path Properties

Controls should always ensure that it can produce a string for any path property. This is because all interfaces that deal with path properties (either *IDispatch* for the property itself or *IBindHost,* in 4.3 below, for converting the string into a moniker) deal with paths as strings.

However, controls should not compare two path properties as strings. Instead, the comparison should be carried out using two monikers created from those strings through *IBindHost::CreateMoniker* (see 4.3.1 below). With monikers, the comparison between *pmk1* and *pmk2* is the call *pmk1->IsEqual(pmk2)* or *pmk2->IsEqual(pmk1).*

17 Integrated Author Tool Browsing and Control Property Pages

A sophisticated container or authoring tool will likely provide its own browsing UI for various types of data, typically the most common types of data that would be assigned to path properties (images, sound, video). To maintain as much of a consistent user interface as possible, a container should make such browsing UI available to a control's property pages such that the property page can invoke that UI instead of showing its own, which would likely be inconsistent.

This integration is achieved through a container-side interface called *IOleBuilderManager*. An exact specification for this interface is not, however, available at this time and will be provided at a later date. The basic idea, however, is that through this interface a control's property page can (a) test for the availability of container-side browsing UI and (b) invoke that UI if it is available. This would allow the page to enable a "Browse..." button for a particular path property, invoking the container's UI when that button is pressed.

18 Container Supply of Monikers via *IBindHost*

Working with data paths requires, of course, the ability of the control to access data referenced by its path properties. The key part of such data access is providing some means by which the control turns a data path name into a moniker that it can bind. Because the container/authoring tool is the one that generally knows the context that modifies any relative path name, the container, in this relationship, is responsible for generating an appropriate moniker when the control asks for one.

This step happens through a container's (or authoring tool's) implementation of the interface *IBindHost,* which is implemented as a service available through a container site's implementation of *IServiceProvider*, and is the topic of the next section. The control accesses the bind host implementation by calling *QueryInterface(IID_IServiceProvider...)* on whatever site pointer it already has and calling *IServiceProvider::QueryService(SID_SBindHost)*. Use of the service provider means that the bind host is generally a separate object from the site, although as an implementation convenience it could be implemented on the site directly..

Of course, in order for a control to have a site through which it can access these services the control must provide some means through which the container can pass its site pointer to the control in the first place. Many controls implement *IOleObject* whose *SetClientSite* member accomplishes exactly this. However, not all controls need to implement *IOleObject,* so a section below describes a simple siting mechanism using *IObjectWithSite* to achieve the same ends.

This specification requires that a control using data paths supports a siting mechanism. If the control doesn't support IOleObject, it ***must*** *support IObjectWithSite.* A container that supports data paths must provide a site through one of these mechanisms and must generally implement *IBindHost* and provide access to it via the site's *IServiceProvider.* If a control that uses paths finds itself without a container site it can still degenerate gracefully by calling *MkParseDisplayNameEx* itself, as necessary.

> *Current Sweeper container components do not support IObjectWithSite; controls should for the time use IOleObject::SetClientSite as a siting mechanism. Support for IObjectWithSite will be added later.*

19 Overview of *IBindHost*

The creation of a moniker from a potentially relative path name is what is considered a container-side "service" that a control accesses through *IBindHost::CreateMoniker.* In addition, a control asks its container to download data via *IBindHost::MonikerBindToStorage* as described in Section 4.4 below.

The *IBindHost* interface is defined as follows:

```
interface IBindHost : IUnknown
    {
    HRESULT CreateMoniker([in] LPCOLESTR pszName, [in] IBindCtx *pBC
        , [out] IMoniker** ppmk, [in] DWORD dwReserved);

    HRESULT MonikerBindToStorage([in] IMoniker *pMk, [in] IBindCtx *pBC
        , [in] IBindStatusCallback *pBSC, [in] REFIID riid
        , [out, iid_is(riid)] void **ppvObj);

    HRESULT MonikerBindToObject([in] IMoniker *pMk, [in] IBindCtx *pBC
        , [in] IBindStatusCallback *pBSC, [in] REFIID riid
        , [out, iid_is(riid)] void **ppvObj);
    }
```

The *CreateMoniker* member takes a text string of some path—which may be absolute or relative—and returns a moniker that references the actual absolute location. That is, the container combines its known document location with the path supplied in *pszName* to create the resulting moniker. How the container does this depends on the document path and the type of name found in *pszName*, which is either something that the URL moniker can handle (including filenames) or an item name referring to some portion of the container document itself.

© Microsoft Corporation,1995-1996, All Rights Reserved.

In the item name case, a container can specify a prefix character, like "#" or ">", that it will use to differentiate an item name from some other kind of name. This fact should be known to users of the container at author-time such that those users can enter the correct names in whatever UI is used to assign path values. When the container detects its prefix character, it knows that it has to parse the name according to its own rules.

In any other case, the container creates a moniker from the text string directly using *MkParseDisplayNameEx* This handles the cases where URLs are given as well as display names of File!Item monikers and other similar composites. If for some reason *IBindHost::CreateMoniker* fails, the calling control can degenerate to calling *MkParseDisplayNameEx* itself, but this should only be used as a last resort.

Either way, the container now has a moniker for the data path as well as a moniker naming its own document. To return a full moniker from *ParseDisplayName* it then calls *pmkDocument->ComposeWith(pmkPath*, &*pmkReturn*) where *pmkReturn* receives the moniker to return to the caller.

This rule assumes that the document moniker class implements *ComposeWith* in an intelligent manner. If the document moniker is a File moniker, for example, and the path moniker is a File moniker, *ComposeWith* will return one File moniker with the combination. The same applies for any case where the path moniker is a URL and the document is either a file or a URL. It is even possible to mix a URL document moniker with a File moniker for the path, provided that the File moniker is not absolute (where the combination would make no sense).

When the path moniker is some item or composite, then the File or URL moniker *ComposeWith* implementations simply create another composite.

The bottom line is that *IBindHost::CreateMoniker* will create an precise moniker that is the combination of the document's own moniker (which is obtainable through *IOleClientSite::GetMoniker(OLEWHICHMK_CONTAINER)* and a moniker created from the path name. The control can use this moniker to bind, but this is not the moniker that the control should save. See Section 4.5 for more details on persistence.

20 Generic Siting with *IObjectWithSite*

A control that does not need *IOleObject* but still requires access to its container site, specifically for the purposes of accessing container-side services, must instead implement *IObjectWithSite*:

```
interface IObjectWithSite : IUnknown
    {
    HRESULT SetSite([in] IUnknown *pUnkSite);
    HRESULT GetSite([in] REFIID riid, [out, iid_is(riid)] void **ppvSite);
    }
```

In the absence of *IOleObject* a container can attempt to provide the *IUnknown* pointer to its site through *IObjectWithSite::SetSite. IObjectWithSite::GetSite* is included as a hooking mechanism, which allows a third party to intercept calls from the control to the site.

When a control needs to create a moniker from a data path, it obtains the bind host through the site's *IServiceProvider* interface and proceeds from there.

> *Current Sweeper container components only support siting through IOleObject. Support for IObjectWithSite will be added later.*

21 Cooperative and Asynchronous Data Retrieval

Once a control has a moniker for a data path property, it will eventually want to call that moniker's *IMoniker::BindToStorage* to retrieve an interface pointer through which the control can retrieve data (or write it as described in the next section).

In order to deliver optimal quality of service to browser users, a control must read data from an external source in a cooperative manner as much as possible. That is to say, the control should support reading linked data in an asynchronous manner, paying attention to container prioritization and allowing the container to participate in the transfer as it wishes. In order to provide the container with authority over the binding operation, the control should use the *IBindHost::MonikerBindToStorage* and *IBindHost::MonikerBindToObject* member functions in order to bind to any moniker.

A control will first, of course, check if the moniker it obtained from the container is, in fact, an asynchronous moniker at all by querying it for *IMonikerAsync* (see **Asynchronous Monikers,** which specified this interface as an alias for *IUnknown*). If this interface is *not* present, the moniker is synchronous and the control does not need to create an *IBindStatusCallback* callback interface in order to receive asynchronous notifications from the moniker bind operaion.

If the moniker is asynchronous, then the control operates as any other client of such a moniker does, by creating an *IBindStatusCallback* callback interface for asynchronous download, as described in the document **Asynchronous Moniker.** The specific requirement for a control is that instead of binding to its moniker directly, it should bind to the moniker through its container, via *IBindHost::MonikerBindToStorage* or *IBindHost::MonikerBindToObject.* This call gives the container a chance to register any callbacks or other bits in the bind operation as desired. For example, a container that wishes to watch the progress of the transfer through *IBindStatusCallback::OnProgress* will register its own callback for the bind operation, paying attention to *OnProgress* callbacks and delegating all other callbacks to the control that initiated the bind operation. .

If the container does not implement *IBindHost*, the control can always bind to its moniker(s) directly through *IMoniker::BindToStorage* and *IMoniker::BindToObject.* These should only be used in degenerate cases when there is no *IBindHost* present.

A control passes its *IBindStatusCallback* to its container directly via *IBindHost::BindToXXX*. However, if the control wishes to participate in format negotiation (e.g. HTTP), it must create a bind context and register its FORMATETC enumerator on this bind context before calling *IBindHost::BindToXXX.* When a control initiates an asynchronous transfer for a data path, usually within *IPersist*::Load,* it should always obtain the data through *IBindStatusCallback::OnDataAvailable* exclusively. This is so the control can immediately release the moniker and the bind context and return immediately from *IPersist*::Load* so the container can continue processing. Such code inside *IPersistStreamInit::Load* where we want to obtain another asynchronous stream, for example, would appear as follows:

```
HRESULT CImpIPersistStream::Load(IStream *pstm)
    {
    IMoniker *pmkPath;
    IStream  *pstmAsync;
    IBindCtx *pbc;

    /*
     * Load properties with pstm->Read which is either blocking or synchronous.
     * Assume that one property is a path moniker which we recreate with
     * OleLoadFromStream into pmkPath.
     *
     * Also assume that m_pbsc is our IBindStatusCallback implementation, and
     * m_pEnumFE is our FORMATETC enumerator.
     */

    //pIBindHost is a pointer to the container's IBindHost
    CreateAsyncBindCtx(0, NULL, m_pEnumFE, &pbc);
    pIBindHost->MonikerBindToStorage(pmkPath, pbc, m_pbsc, IID_IStream, &pstmAsync);


    //Do this only for async monikers
    if (NULL!=pstm)
        pstmAsync->Release();

    pbc->Release();
    pmkPath->Release();

    //Data shows up in IBindStatusCallback::OnDataAvailable

    return S_OK;
```

© Microsoft Corporation,1995-1996, All Rights Reserved.

```
    }
```

So when retrieving data, the path moniker that names the external storage location is either synchronous or asynchronous. If the moniker used in binding is a synchronous moniker, then read calls from the control will effectively block the execution thread as is expected—this is how the OLE programming model already works with regards to moniker binding.

Of course, an object that does not wish to block and can return E_PENDING as necessary from some of its interface members (see Section 5) may elect to perform the otherwise blocking read operations in another thread, thereby allowing the original thread to continue. This still allows *IPersist*::Load* to return right away without waiting for the data to arrive.

The tricky part in handling asynchronous data is communicating the control's "readiness" state (including E_PENDING return codes). These topics are covered in Section 5 later in this document.

22 Aborting a Data Transfer

Any control that is currently reading or writing data in an asynchronous manner must pay attention to *IBindStatusCallback::OnStopBinding*. If the control has all its data already then this notification merely says that the transfer is complete. If the control has not received all its data, then the transfer was aborted for some other reason—perhaps the container scrolled the object out of view and stopped the transfers itself.

The control itself can call *IBinding::Abort* if it needs to stop the transfer for some reason. The control receives the *IBinding* pointer through *IBindStatusCallback::OnStartBinding* and must save it for later use. One such use is for handling the case where a container destroys a control while a transfer is underway—the control has to stop the transfer as part of its destruction procedure.

Also, if a container or authoring tool assigns a new value to a path property while the control is currently retrieving data asynchronously for that path, the control must stop the transfer through *IBinding::Abort* as soon as is reasonable and save the new path value.

Because a control cannot predict exactly how and when one or more data transfers will be aborted, it must be careful where reentrancy is concerned, especially inside the implementation of *IBindStatusCallback::OnStopBinding* (but in all other members as well). That is, the control should maintain an internal state machine to protect itself from operations that occur during or after an aborted transfer.

23 Transfer Priority

As the container owns the "document" as a whole, the container establishes which control will be given the right to transfer data first, second, and so on. When multiple transfers are going on at the same time, a container can control the relative priority of each transfer through its own implementations of *IBindStatusCallback* which are used to control each bind operation (as managed through *IBindHost::MonikerBindToXXX*). A container can establish its desired relative priority in *IBindStatusCallback::GetPriority*. Once the transfer begins and the callbacks receive *IBindStatusCallback::OnStartBinding,* the container will have the *IBinding* interface through which it can set the priority there as well. Controls should not interfere with this process.

24 Controls that use WinINet or Sockets Directly

There are a certain class of controls that may want to bypass the mechanisms of the URL moniker in order to retrieve bits for some data path directly from the network through a transport layer like WinINet or Sockets. In such cases, the control is not willing to call *IBindHost::MonikerBindToStorage* but instead wants to use the absolute URL directly.

To accomplish this, the control still asks the container for the full moniker in the first place since this step is necessary to resolve any relative path names. Once the control has the moniker, it can retrieve the full URL simply by calling *IMoniker::GetDisplayName*.

25 Object Persistence and Path Properties

When a control is asked to save it's persistent data through some *IPersist*::Save* call, it is being asked to save its properties, including its current path properties and the BLOBs in the locations specified with those path properties. That is, once the control has saved its properties into the storage appropriate for the *IPersist** interface in use, it also *synchronously* saves its BLOBs by (1) obtaining the moniker for the path property from the container as described in section 4.3, (2) calling *IMoniker::BindToStorage* with the container's bind context (as described in Section 4.4), and (3) writing data to the storage obtained in step 2. When *IPersist*::Save* returns, the object has completely written any data it needed to save.

> *In browsing cases, the primary consideration of this document, saving is not an issue for the most part. This is primarily an author/publish time issue and the discussion above is therefore suggestive and not definite.*

A control should save its path property values as strings, not as monikers. In short, a control should simply save whatever values are assigned to its data path properties (as opposed to serializing any monikers that might have been created using these strings).

26 Container Flexibility in Moniker Choice

> *This section is proposal only.*

The discussion in the previous section made no stipulations as to what *class* of moniker must be assigned to any path property. This is intentional: a container can instruct a control to store its persistent data in any location of the container's choosing, as long as the container can create path name that, when turned into a moniker, references the exact piece of storage that the control needs to load. In the case of creating a complex Web site, most of these monikers will be URL monikers. But there are many other moniker classes that can be used effectively, such as File monikers, generic composite monikers, or even custom monikers.

This design, therefore, allows the container full flexibility in choosing storage locations depending on the authoring state of the document. At author time, the container (generally at the authors behest) may choose to keep the persistent data of all controls embedded in the document. In this case the container merely passes each control a name identifying some location in the document or on the local file system, such as a single Item moniker or a File moniker. The container then provides the necessary binding support for whatever moniker it supplies through *IBindHost::CreateMoniker*. In the case where the authoring tool has the data stored in the document itself, then supplying a File!Item moniker to the control means that the control's call to *IBindHost::MonikerBindToStorage* returns very quickly.

At publish time, the container may then choose to break whatever portions of control data it desires out of the document, saving that data in other locations and reassigning paths. Since the container will have some idea of the *size* of each control's persistent data, it may choose to leave some of that data embedded in the document anyway, leaving the controls concerned with simple File!Item monikers (and such). If the container moves some data to other Web sites, it would create URL monikers for those sites and assign them the path properties of the appropriate controls. As far as the control is concerned, the data has moved but the data is still named with some moniker. The control continues to use the same *IBindHost::MonikerBindToStorage* code that it did before.

What is important to understand here is that *the control never has to distinguish between author-time and publish/run-time differences as far as its data path storage is concerned*. The control always has some moniker naming its persistent data store(s), and always calls *IBindHost::MonikerBindToStorage* in all circumstances. This maintains a single programming model across all document modes, while still giving the container complete flexibility in choose storage locations and how the data in the document is distributed.

27 Exposing Path Properties from Nested Controls

> *This section is proposal only for authoring tools.*

© Microsoft Corporation,1995-1996, All Rights Reserved.

Some controls, regardless of whether they themselves use any path properties, may contain other nested controls that do use such paths. Nevertheless, an authoring tool will want to construct a list of *all* paths used by all controls, regardless of the level of nesting involved. This facilitates site management—the authoring tool can examine all external data references from the top-level document and correct the paths involved if the site is being relocated.

Any control that itself uses one or more path properties need only expose those properties as described in earlier sections. Those controls that themselves *contain* other controls must expose those nested *controls* such that an authoring tool can check the path properties in those controls as well.

For this reason, a containing control must implement *IOleContainer*[1] with the following behavior:

Member Function	Behavior
IUnknown members	Implemented as usual
ParseDisplayName	Returns E_NOTIMPL
LockContainer	Returns E_NOTIMPL.
EnumObjects	Returns an *IEnumUnknown* enumerator through which the caller can retrieve the *IUnknown* pointers for each nested control.

Given this behavior, the top-level container or authoring tool that wishes to examine all path properties in all controls, regardless of the nesting level, uses *IProvideClassInfo::GetClassInfo* to retrieve the control's type information, then uses the same procedures described earlier to build a list of the control's dispIDs that each correspond to some data path property. The caller can then retrieve this property value, examine it, and modify it as necessary. This procedure is summarized in the following pseudo-code:

```
//In top-level container
construct IEnumUnknown for all top-level controls in pEnum
FindAllPaths(pEnum)
//Done

function FindAllPaths(IEnumUnknown *pEnum)
    }
    IUnknown *pObj;

    while (SUCCEEDED(pEnum->Next(&pObj)))
        {
        call IProvideClassInfo::GetClassInfo or obtain coclass type information via type lib

        if (SUCCEEDED(ITypeInfo2::GetCustData) and value is greater than zero)
            {
            iterate over properties
                {
                for each path property
                    {
                    examine the path
                    update/modify the path as necessary
                    }
                }

            ITypeInfo2::Release();
            }

        if (SUCCEEDED(pObj->QueryInterface(IID_IOleContainer, &pCont)))
            {
            pCont->EnumObjects(&pEnumNested)

            if (NULL!=pEnumNested)
                FindAllPaths(pEnumNested)

            pCont->Release();
            }

        pObj->Release();
```

[1] The containing control might also choose to implement *IOleItemContainer* as well (which is derived from *IOleContainer*) in order to allow easier navigation to a particular control give the control's name.

```
        }
    }
```

Through this process the top-level container can examine all path properties in use in the document. It is unlikely that this will be a seriously expensive operation in all except the most complex documents. Typically there will only be a handful of controls on a page with any path properties at all, and even fewer containing controls with a significant number of nested controls. Second, third, fourth, and deeper levels of nesting will be exceptionally rare, so implementing this scheme with a recursive code structure will not be expensive in probably 90%+ of situations.

What is described here for a containing control does *not* apply to controls that internally use some kind of helper object as a supporting component. In such cases the helper object is not a “nested control”—it is merely a provider of a useful service. The control that is using the helper should not consider itself a “containing control” because it does not implement the necessary container-side services that would be necessary to nest an actual control.

28 Communicating Control “Readiness”

For all persistence interfaces other than *IPersistMoniker,* the authoring tool or container assumes that once *IPersist*::Load* returns, the control has loaded all of its *properties*. However, controls that use data paths may not actually have all of their data at this time. That is, within *IPersistStreamInit::Load,* for instance, a control might load a data path property for an AVI file and begin an asynchronous retrieval of that AVI data before returning from *Load* as shown earlier in Section 4.4.

When data is coming in asynchronously like this, the control may not necessarily be ready to handle all requests the container might make of it. Accordingly, the control may return E_PENDING from some interface member functions. The first section below describes the use of E_PENDING.

In addition, certain user-supplied code, in a container like Visual Basic for example, may want to take action on the current “readiness” state of a control such as enabling or disabling one control based on the ability of another control to accept certain calls. The second section below described a standard property, *ReadyState*, which reflects this aspect of a control, and a standard event, *OnReadyStateChange*, that informs the container that the state has changed. Because the *ReadyState* property is primarily of interest to user-supplied code, this standard event is used instead of *IPropertyNotifySink::OnChanged* to (a) make the event visible in programming tools, and (b) to pass the new *ReadyState* value along with the event.

29 Use of E_PENDING

In the lifetime of any particular control that uses data paths there are generally four (possibly five) distinct states of readiness that a control may have:

State	Description
Uninitialized	The control is waiting to be initialized through *IPersist*::Load.*
Loading	The control is initializing itself through one or more asynchronous properties. Some property values may not be available.
Loaded/Can-Render	The control has returned from *IPersist*::Load* such that all its properties are available and it has started any asynchronous data transfers. The control’s properties are available and it is ready to draw at least something through *IViewObject2::Draw.* Making properties available doesn’t necessarily require the control’s type information.
Interactive	The control is capable of interacting with the user in at least some limited sense and can supply its type information. Full interaction may not be available until all asynchronous data arrives.
Complete	The control is completely ready for all requests.

© Microsoft Corporation,1995-1996, All Rights Reserved.

The exact number of states may be reduced to only two or three. Final design is TBD.

These states are listed in order of priority which is to say that a control should strive to makes its properties and some basic rendering capabilities available as soon as it can (even if the rendering is to just draw a rectangle with text in it) before going on to become interactive and before being able to fulfill requests that depend on asynchronous data.

From the point in time when a control is first instantiated, certain interface members may return E_PENDING to indicate that the control have not retrieved enough data yet to carry out the request (note that all *interfaces* must be available through *QueryInterface* though none of their members may do anything yet). To understand the implications, it is useful to step through the moments in time during the loading of a document and the creation and initialization of the controls in that document.

36. (**NonExistent!):** The container has loaded the document but has not yet instantiated any control. At this point the container can display the document with nothing but rectangle placeholders for each visible control. That is, since no controls are yet loaded, the container cannot call *IViewObject2::Draw* for any of them, and must rely on cached information (extent, label, etc.) to render anything for a control.
37. **Unintiailized:** The container has loaded a control but the control is completely uninitialized—that is, the container has not yet called *IPersist*::Load*. The control must be loaded first before the container can expect to call any other interface member. If the container wishes to draw an image for the control it must do so itself using any cached information (like a caption string).
38. **Loading:** A control has returned from *IPersist*::Load* and is making properties available as they arrive. The control must be capable of rendering something minimal (like a rectangle and text) through *IViewObject2::Draw* and should give priority to retrieving its *Caption* and *Text* properties first, followed by rendering-related properties (*BackColor, ForeColor, DrawStyle,* etc.)
39. **Loaded/Can-Render:** The control has returned from a call to *IPersist*::Load* such that it has loaded its properties (or a control that was **Loading** has now retrieved all its immediate properties). At this point the control must be prepared to at least render something minimal (like a rectangle and text) through *IViewObject2::Draw*. In simple cases, like a button, the control will already have all its data necessary for rendering such as colors, text, font, and styles.
40. **Interactive:** The control has enough information to be mostly interactive: it can be in-place activated (and perform layout negotiation), handle all property operations, accept user input, and so on. The control also has its type information available. In addition, progressively higher detailed renderings (or other streamed data) may be coming into the control at this point such that the control sends *IAdviseSink::OnViewChange* notifications to tell the container that the control should be redrawn (if the control doesn't have a window, that is; if it has a window then it just redraws when it wants to).
41. **Complete:** The control has loaded all of its data including that obtained through paths such that it can do more sophisticated renderings using possibly medium and high-detail graphics.

During each state other than **Complete,** various interface member functions may not be operative, that is, they will return E_PENDING, and certain properties may not be available. The following table describes which interfaces and methods *must* be ready in each of these states, if the control supports those interfaces at all.

State	Required Interfaces/Members/Properties
Uninitialized	*IPersist*::Load* and *IPersist*::InitNew*
Loading	*IViewObject2::Draw, Idispatch::Invoke* for some properties.
Loaded/Can-Render	All of *IViewObject2, Idispatch::Invoke* for all properties that don't depend on extra data.
Interactive	All of *IDispatch* (methods and type information), all of *IRunnableObject, IOleObject*, *IOleInPlaceObject* (and *IOleInPlaceActiveObject*),

	IProvideClassInfo2, *IspecifyPropertyPages, IDataObject, IOleControl, IConnectionPointContainer,* and *IConnectionPoint*, with the exception of any operation that depends on asynchronous data that is not yet available, such as *IDataObject::GetData* for certain formats.
Complete	Everything is ready including other members of rest of *IPersist**.

Obviously this puts a little necessary burden on a control which will need to maintain one or more "readiness" flags that are checked on entry to various interface members such that the function returns E_PENDING if the flag says "not ready." The more distinct states the control chooses to support the more complex this will become internally; however, the container doesn't need to differentiate these states: it simply handles E_PENDING by providing some default action of its own where appropriate, then trying the operation again later.

In other words, these states are not something that the container has to internally maintain—they are simply a guide to how a control implementation should be structured to work best in the Internet environment.

Many controls will never have occasion to differentiate between **Loaded** and **Interactive** because nearly all simple controls have almost data dependencies for becoming interactive. The differentiation between the states is made here because some controls with data paths may not be ready for **Interactive** until at least some data is received asynchronously. Nevertheless, a control should attempt to become interactive as soon as possible.

30 The *OnReadyStateChange* Event and *ReadyState* Property

How does a container know when any given control is ready to render itself, can be interactive with the user, or has completely retrieved all its necessary data for full operation? That is, how does a container know the "readiness" state of a control?

A simple scenario demonstrates the need for the container to have such knowledge. Imagine that you have a form on which is a video control and a "Play" button that calls the *Play* method in the video control. The video control has a *VideoPath* property set to some URL. Now when this form is first opened, the video control will not have any data which it would play, so the container would want to disable the "Play" button immediately. When the video control is asked to load itself, it will loads it's embedded and begin an asynchronous transfer of its AVI file. When the transfer is complete the container will want to enable the Play button.

More precisely, user code would pick up some event from the video control and use that event to enable the button (the container would not generally be hardcoded to do this). In addition, user code outside of an a change event may need to know the current readiness state of a control.

For the purposes described here, readiness is defined as one of the following values that each correspond to one of the states described in the previous section:

```
enum
    {
    READYSTATE_UNINITIALIZED=0, //Never used except as default initialization state
    READYSTATE_LOADING=1,       //Control is currently loading its properties
    READYSTATE_LOADED=2,        //Control has been initialized via IPersist*::Load
    READYSTATE_INTERACTIVE=3,   //Control is interactive but not all data is available
    READYSTATE_COMPLETE=4       //Control has all its data
    }
```

Again, the "loading" state is only used for control that are initializing themselves asynchronously through *IPeristMoniker::Load*. Otherwise controls begin with READYSTATE_LOADED.

A control makes its current state available through the standard *ReadyState* property (DISPID_READYSTATE, of type *long*). Until a control is initialized with *IPersist*::Load,* the value of this property is undefined (and unavailable, since the

© Microsoft Corporation,1995-1996, All Rights Reserved.

control is not ready at all!) When a control has loaded its properties, the property must be at least READYSTATE_LOADED. In many cases the control will not differentiate between loaded and interactive, in which case the control is interactive immediately and this property reflects at least READYSTATE_INTERACTIVE. In addition, some controls do not differentiate between "loaded" and any other state (such as buttons which load all their properties and have no external data to retrieve) in which case the property should always be READYSTATE_COMPLETE. In this case a control need not support the *ReadyState* property at all, in which case the container always assumed READYSTATE_COMPLETE.

When a control differentiates at least two of these states, it must notify the container of the state transition with the standard control event called *OnReadyStateChange* (DISPID_ONREADYSTATECHANGE) which has the following prototype:

```
[id(DISPID_ONREADYCHANGE)]
        void OnReadyStateChange(long lReadyState);
```

The state flag passed with this event is one of the READYSTATE_* flags shown above. The general meaning of these states was described in the last section, but to make use of the event the programmer must understand—from the control's documentation—what methods and properties or other features of the control are available in any given state. In our example, the video control might not say its "interactive" until it as at least some of its data, and its definition of "complete" might mean that it's ready to play. On the other hand, a graphical hyperlink control might describe itself as "interactive" as soon as it's loaded, since it can receive user input, then only send the "complete" state when it has actually retrieved all of its graphics.

Because this event is designed for use in user code, the programmer is responsible for understanding what each particular state means for whatever control might be sending it. In the video example, the programmer has read that the video's *Play* method is not available until the ready state is "complete" in which case the VB code enable the "Play" button on READYSTATE_COMPLETE only:

```
Private Sub video1.OnReadyStateChange(ReadyState as Long)

    if ReadyState=READYSTATE_COMPLETE
        buttonPlay.Enabled=TRUE

End Sub
```

Controls that do not differentiate between all the states can simply choose to send only the appropriate ones, such as READYSTATE_COMPLETE if the control is completely ready as soon as it's loaded. Sending "complete" implies that the control has already passed through "loaded" and "interactive." Sending "interactive" implies that "loaded" has already passed but "complete" has not been reached. Again, because user code is what's generally interested in this event, the programmer is assumed to understand any particular control's behavior with this event. User code should not depend on all three states being sent explicitly.

Of course, any control that may not be ready to receive certain requests still has to protect itself in case a container makes some request that the control cannot yet fulfill. In that case the control returns E_PENDING from the interface member function.

31 Other Considerations

32 Component Categories for Describing Internet-Aware Objects

Current Sweeper components do not pay attention to categories.

In all the various features defined in this specification, there are various useful categories for describing what a object requires from its container as well as what features an object supports, which is useful in a container's browsing UI. The following tables describe the meaning for various categories as both "implemented" and "required" categories:

"Required" Categories	**Description**
CATID_PersistsToMoniker, CATID_PersistsToStreamInit, CATID_PersistsToStream, CATID_PersistsToStorage, CATID_PersistsToMemory, CATID_PersistsToFile, CATID_PersistsToPropertyBag	Each of these categories are mutually exclusive and are only used when an object supports only one persistence mechanism at all (hence the mutual exclusion). Containers that do not support the persistence mechanism described by one of these categories should prevent themselves from creating an objects of classes so marked.
CATID_RequiresDataPathHost	The object *requires* the ability to save data to one or more paths and requires container involvement, therefore requiring container support for *IbindHost.*

CATID_RequiresDataPathHost is of questionable value—controls should always degenerate gracefully in the absence of the bind host.

"Implemented" Categories	**Description**
CATID_PersistsToMoniker, CATID_PersistsToStreamInit, CATID_PersistsToStream, CATID_PersistsToStorage, CATID_PersistsToMemory, CATID_PersistsToFile, CATID_PersistsToPropertyBag	Object supports the corresponding *IPersist** mechanism for the category.
CATID_InternetAware	The object is "Internet-aware" meaning that (a) the object needs no special behavior to work well in the Internet environment, or (b) the object uses large data sets but is written to retrieve that data asynchronously. In addition, all "Internet-aware" objects are marked with OLEMISC_SETCLIENTSITEFIRST. This category is defined to aid authoring tools differentiate between objects that may or may not understand the special implications of the Internet.

The following table provides the exact CATIDs assigned to each category:

Category	**CATID**
CATID_RequiresDataPathHost	0de86a50-2baa-11cf-a229-00aa003d7352
CATID_PersistsToMoniker	0de86a51-2baa-11cf-a229-00aa003d7352
CATID_PersistsToStorage	0de86a52-2baa-11cf-a229-00aa003d7352
CATID_PersistsToStreamInit	0de86a53-2baa-11cf-a229-00aa003d7352
CATID_PersistsToStream	0de86a54-2baa-11cf-a229-00aa003d7352
CATID_PersistsToMemory	0de86a55-2baa-11cf-a229-00aa003d7352
CATID_PersistsToFile	0de86a56-2baa-11cf-a229-00aa003d7352
CATID_PersistsToPropertyBag	0de86a57-2baa-11cf-a229-00aa003d7352
CATID_InternetAware	0de86a58-2baa-11cf-a229-00aa003d7352

For more information on categories, see the **Component Categories** document.

© Microsoft Corporation,1995-1996, All Rights Reserved.

33 Summary

34 Added Standard Properties, Methods, Events, and Interfaces

The following table summarizes the added standard dispIDs described in this specification:

Control Property	**Value**
DISPID_READYSTATE	(-525)

Control Methods	**Value**
none	

Control Events	**Value**
DISPID_READYSTATECHANGE	(-609)

The following table summarizes the new interfaces and other GUID values described in this specification, as well as those that are documented here with the exception of category IDs described in section 6.1):

Symbol	**Value**
IID_IBindHost	FC4801A1-2BA9-11CF-A229-00AA003D7352
IID_IServiceProvider	6D5140C1-7436-11CE-8034-00AA006009FA
IID_IObjectWithSite	FC4801A3-2BA9-11CF-A229-00AA003D7352
IID_IPersistMemory	BD1AE5E0-A6AE-11CE-BD37-504200C10000
IID_IPersistPropertyBag	37D84F60-42CB-11CE-8135-00AA004BB851
IID_IPropertyBag	55272A00-42CB-11CE-8135-00AA004BB851
IID_IErrorLog	3127CA40-446E-11CE-8135-00AA004BB851
GUID_PathProperty	0002DE80-0000-0000-C000-000000000046
GUID_HasPathProperties	0002DE81-0000-0000-C000-000000000046
SID_SBindHost	FC4801A1-2BA9-11CF-A229-00AA003D7352 (same as IID_IbindHost)

IPersistMoniker and its IID is documented in **Asynchronous Monikers**.

35 Requirements for Internet-Aware Controls and Objects

If a control does not use any data paths for BLOB data, then working well within Internet-aware containers mostly means that a control supports as many *IPersist** interfaces as is reasonable for it, giving the container and asynchronous monikers a great deal of flexibility in how they initialize the control in both persistent embedding and persistent linking cases.

Controls that only support one persistence interface **must** mark themselves as Mandatory users of the appropriate category.

> *Current Sweeper components ignore the above category so it isn't technically necessary at this time.*

Controls with one or more data paths have these requirements:

a. **Must** support either *IOleObject* or *IObjectWithSite* as a siting mechanism.

b. **Must** mark data path properties in type information with the **[bindable]** attribute as well as the appropriate custom attribute using *GUID_PathProperty*.

c. **Must** mark the **coclass** entry in the control's type information with the custom attribute using *GUID_HasPathProperties*.

d. **Must** follow the rules of moniker creation and persistence, using the container's *IBindHost* as necessary and as *IBindHost* is available.

e. **Must** understand how to bind with an asynchronous moniker using a container-provided bind context from *IBindHost* if available and how to receive data through *IBindStatusCallback.*

f. **Must** prioritize initialization sequences and data retrieval to render and interact as soon as possible, returning E_PENDING from member functions as appropriate for the control's initialization state.

g. **Should** supply a *ReadyState* property, if applicable, and fire *OnReadyStateChange* events as needed.

h. **Should** support *IPersistPropertyBag* for supporting HTML PARAM attributes.

Current Sweeper components do not involve IObjectWithSite for (a)

This specification also recommends that controls be OLE Accessibility-aware for the benefit of users requiring accessibility services.

36 Requirements for Internet-Aware Containers/Clients/Controllers

Whereas much of a control's functionality is optional (with the exceptions of necessary support for data paths), a container, on the other hand, generally wants to support as many features as possible. As guidelines, a container that wishes to host Internet-aware controls at run-time:

i. **Must** expect and handle E_PENDING, especially from *IViewObject2::Draw*

j. **Must** provide a site to a control through either *IOleObject* or *IObjectWithSite* mechanisms

k. **Must** supply initialized bind contexts and name parsing through *IBindHost*

l. **Must** support at least *IPersistStreamInit* as a control persistence mechanism.

m. **Must** support the HTML Extensions for COM Objects if the client is HTML-oriented.

n. Should manage relative creation/initialization priorities for each control in the document.

o. Should supply browsers for common data types required by data paths (authoring tools)

p. Should support as many *IPersist** interfaces as possible to allow maximum control flexibility.

Current sweeper components do not support IObjectWithSite in (b) so IOleObject is the only siting mechanism in current use. Current components also do not implement (f), and support IPersistStorage, IPersistStreamInit, and IPersistPropertyBag as persistence mechanisms relating to (h).

This specification also recommends that controls be OLE Accessibility-aware for the benefit of users requiring accessibility services.

37 Standard Internet-Aware Objects

As a benefit to control developers, Microsoft will supply Internet-Aware picture, sound, and video objects which will make dealing with these data formats in the Internet environment a non-issue for many controls. The existing picture object, for example, makes a control's manipulation of a bitmap, metafile, or icon nearly effortless. The additional objects described here aim to do the same for their respective data types, where all standard objects will support the ability to retrieve large data BLOBs in an asynchronous manner.

© Microsoft Corporation,1995-1996, All Rights Reserved.

A near-future API function called *OleLoadPicturePath* will be available with the final version of Internet Explorer 3.0. *OLELoadPicturePath* will take a moniker and a bind context (which the control retrieves from *IBindHost*) and will take care of the details of downloading the picture. In return, *OLELoadPicturePath* provides an *IPicture* interface pointer that is identical to the standard picture object that already exists. The control can simply call *IPicture::Render* to draw the image.

The exact specifications for other picture object enhancements and other standard objects are not available at this time. The design may be done along the lines of a "data type" architecture such that new objects supporting other data types can be added in the same manner. In addition, the architecture may support plug-in "player" objects that extend the usable formats for each data type object. For example, a standard picture object would probably support formats like BMP, DIB, Metafile, Icon, GIF, and JPEG natively. Individual "players" could be then plugged into this picture object for supporting other formats like TIFF, PCX, or other proprietary formats. As each player would be a COM object in itself, controls could take advantage of any "code download" technology for these players.

> *Specifications are not available at this time for these. Proposal only.*

38 Interface Reference

This section provides the details for the *IBindHost, IServiceProvider, IObjectWithSite, IPersistMemory, IPersistPropertyBag, IPropertyBag,* and *IErrorLog* interfaces. For *IBinding, IBindStatusCallback*, and *IPersistMoniker,* see **Asynchronous Monikers**. Specifications for *IOleBuilderManager* are not available at this time.

39 The *IBindHost* Interface

This interface is implemented as a service available from a site object through which a container supplies services and information that objects require when performing an asynchronous data transfer, specifically name-to-moniker parsing and a container-initialized bind context.

IDL:

```
[
uuid(fc4801a1-2ba9-11cf-a229-00aa003d7352)
    , object,pointer_default(unique)
]

interface IBindHost : IUnknown
    {
    HRESULT CreateMoniker([in] LPCOLESTR pszName, [in] IBindCtx *pBC
        , [out] IMoniker** ppmk, [in] DWORD dwReserved);

    HRESULT MonikerBindToStorage([in] IMoniker *pMk, [in] IBindCtx *pBC
        , [in] IBindStatusCallback *pBSC, [in] REFIID riid
        , [out, iid_is(riid)] void **ppvObj);

    HRESULT MonikerBindToObject([in] IMoniker *pMk, [in] IBindCtx *pBC
        , [in] IBindStatusCallback *pBSC, [in] REFIID riid
        , [out, iid_is(riid)] void **ppvObj);
    }
```

40 *IBindHost::CreateMoniker*

HRESULT CreateMoniker([in] LPCOLESTR pszName, [in] IBindCtx *pBC,
[out] IMoniker **ppmk, [in] DWORD dwReserved);

CreateMoniker provides the caller with a means to turn some sort of text name into a moniker such that the caller does not have to interpret the name in any way itself. In many cases, the implementation of *CreateMoniker* will simply call *MkParseDisplayNameEx,* but this method gives the implementor of *IBindHost* a chance to catch host-specific strings that *MkParseDisplayNameEx* would not otherwise recognize.

Argument	Type	Description
pszName	*LPCOLESTR*	[in] Pointer to the string containing the name to parse.
pBC	*IBindCtx **	**Optional** bind context to be used when creating the moniker. This parameter is currently ignored, but may be used in the future for passing additional information.
ppmk	*IMoniker ***	[out] The address of the caller's *IMoniker* * variable in which to store the moniker created from *pszName.* The caller is responsible for calling *IMoniker::Release* when the moniker is no longer needed.
dwReserved	*DWORD*	Reserved for future use, must be zero.

© Microsoft Corporation,1995-1996, All Rights Reserved.

Return Value	Meaning
S_OK	The moniker was successfully obtained and the caller is responsible for the interface pointer.
E_OUTOFMEMORY	There is insufficient memory to create the moniker.
E_UNEXPECTED	An unknown error occurred
MK_E_SYNTAX	The bind host was unable to parse the string into a moniker because the information in the name was unrecognizable.

Comments:
E_NOTIMPL is disallowed—a bind host is responsible for providing moniker parsing services.

41 *IBindHost::MonikerBindToStorage*

HRESULT MonikerBindToStorage([in] IMoniker *pMk, [in] IBindCtx *pBC, [in] IBindStatusCallback *pBSC, [in] REFIID riid, [out] void **ppvObj);

MonikerBindToStorage should be the single mechanism controls use when trying to bind to the data addressed by a moniker. This function behaves exactly the same as *IMoniker::BindToStorage*, except that it provides the control container (implementor of *IBindHost*) enough authority over the bind operation so that the control container can take charge of setting bind options and priority, while delegating all results and callbacks for the bind operation back to the control.

Argument	Type	Description
pMk	*IMoniker **	[in] Moniker to bind to.
pBC	*IBindCtx **	**Optional** bind context to be used when binding the moniker. This parameter can be used by the control for passing in additional bind options, such as a format enumerator (*IEnumFormatETC*)..
pBSC	*IBindStatusCallback **	[in] The callback interface whereby the control receives asynchronous callbacks pertaining to the bind operation. **The control must provide its callback interface separately, not registered on the *pBC*** parameter.
riid	*REFIID*	[in] IID of the desired storage interface. Serves the same purpose as the analogous parameter in *IMoniker::BindToStorage*
ppvObj	*void ***	[out] The result of the bind operation (if it is synchronous). This parameter serves the same purpose as the analogous parameter in *IMoniker::BindToStorage.*

Return Value	Meaning
S_OK	The bind operation completed synchronously and successfully. The result of the bind operation is available in *ppbObj*.
MK_S_ASYNCHRONOUS	The bind operation will complete asynchronously. Behavior matches that of *IMoniker::BindToStorage.*
E_OUTOFMEMORY	There is insufficient memory to create the moniker.
E_UNEXPECTED	An unknown error occurred

42 *IBindHost::MonikerBindToObject*

HRESULT MonikerBindToObject([in] IMoniker *pMk, [in] IBindCtx *pBC, [in] IBindStatusCallback *pBSC, [in] REFIID riid, [out] void **ppvObj);

MonikerBindToObject should be the single mechanism controls use when trying to bind to the object addressed by a moniker. This function behaves exactly the same as *IMoniker::BindToObject*, except that it provides the control container (implementor of *IBindHost*) enough authority over the bind operation so that the control container can take charge of setting bind options and priority, while delegating all results and callbacks for the bind operation back to the control.

Argument	Type	Description
pMk	*IMoniker **	[in] Moniker to bind to.
pBC	*IBindCtx **	**Optional** bind context to be used when binding the moniker. This parameter can be used by the control for passing in additional bind options, such as a format enumerator (*IEnumFormatETC*)..
pBSC	*IBindStatusCallback **	[in] The callback interface whereby the control receives asynchronous callbacks pertaining to the bind operation. **The control must provide its callback interface separately, not registered on the *pBC*** parameter.
riid	*REFIID*	[in] IID of the desired storage interface. Serves the same purpose as the analogous parameter in *IMoniker::BindToObject*
ppvObj	*void ***	[out] The result of the bind operation (if it is synchronous). This parameter serves the same purpose as the analogous parameter in *IMoniker::BindToObject.*

Return Value	Meaning
S_OK	The bind operation completed synchronously and successfully. The result of the bind operation is available in *ppbObj*.
MK_S_ASYNCHRONOUS	The bind operation will complete asynchronously. Behavior matches that of *IMoniker::BindToObject.*
E_OUTOFMEMORY	There is insufficient memory to create the moniker.
E_UNEXPECTED	An unknown error occurred

43 The *IServiceProvider* Interface

IServiceProvider is a generic access mechanism to location a GUID-identified "service" that is provided through a control or any other objects that it can communicate with. For example, an embedded object (such as an OLE Control) normally only communicates with its associated "client site" object in the container via the *IOleClientSite* interface supplied through *IOleObject::SetClientSite*. Such an embedded object is required to ask the client site for some other service that the *container* supports were that service may not necessarily implemented in the site itself.

In this regard, the site musts provide a means through which the control managed by that site can access the service when necessary. A specific example of this necessity can be found in the function *IOleInPlaceSite::GetWindowContext,* through which an in-place object or control can access interface pointers for the document object that contains the site and the frame object that contains the document. Because these interfaces pointers exists on separate objects, the control cannot call the site's *QueryInterface* to obtain those pointers.

The generic architecture for achieving the same ends, without requiring ad hoc solutions where the need arises, is the interface *IServiceProvider* whose existence says "somewhere in the code of which I'm part exists some set services that can be accessed through this interface."

The interface itself has only one member, *QueryService*, through which a caller specifies the service ID (SID, a GUID), the IID of the interface desired in return, and the address of the caller's interface pointer variable.

IDL:

```
[
uuid(6d5140c1-7436-11ce-8034-00aa006009fa)
    , object,pointer_default(unique)
]
interface IServiceProvider : IUnknown
    {
    HRESULT QueryService([in] REFGUID guidService
        , [in] REFIID riid, [out, iid_is(riid)] void **ppv);
```

© Microsoft Corporation,1995-1996, All Rights Reserved.

```
    }
```

44 *IServiceProvider::QueryService*

HRESULT QueryService([in] REFGUID guidService, [in] REFIID riid, [out, iid_is(riid)] void **ppv);

QueryService is the factory method for any services exposed through an implementation of *IServiceProvider*. It creates or accesses the implementation the service identified with *guidService,* returning in **ppv* the pointer to the interface specified by *riid*.

Argument	Type	Description
guidService	*REFGUID*	[in] The unique identifier of the service (a SID)
riid	*REFIID*	[in] The unique identifier of the interface the caller wishes to receive for the service.
ppv	*void ***	[out] The address of the caller-allocated variable to receive the interface pointer of the service on successful return from this function. The caller becomes responsible for calling *Release* through this interface pointer when the service is no longer needed.

Return Value	Meaning
S_OK	The service was successfully created or retrieved. The caller is responsible for calling *((IUnknown *)*ppv)->Release();*
E_OUTOFMEMORY	There is insufficient memory to create the service.
E_UNEXPECTED	An unknown error occurred
E_NOINTERFACE	The service exists but the interface requested does not exist on that service.
SVC_E_UNKNOWNSERVICE	The service identified with *guidService* is not recognized.

Comments:
Because there is only one member function in this interface, E_NOTIMPL is not a valid return code—if the function is not implemented the interface has no reason to exist.

45 The *IObjectWithSite* Interface

Often an object will need to communicate directly with a “container site” that is managing the object itself. Outside of *IOleObject::SetClientSite,* there is no generic means through which an object becomes aware of its site. *IObjectWithSite* provides simple objects with a lightweight means (lighter than *IOleObject*) with a siting mechanism. This interface should only be used when *IOleObject* is not already in use.

Through *IObjectWithSite,* a container can pass the *IUnknown* pointer of its site to the object through *SetSite*. Callers can also retrieve the latest site passed to *SetSite* through *GetSite*. This latter function is included as a hooking mechanism, allowing a third party to intercept calls from the object to the site.

IDL:

```
[
uuid(fc4801a3-2ba9-11cf-a229-00aa003d7352)
    , object,pointer_default(unique)
]
interface IObjectWithSite : IUnknown
    {
    HRESULT SetSite([in] IUnknown *pUnkSite);
    HRESULT GetSite([in] REFIID riid, [out, iid_is(riid)] void **ppvSite);
    }
```

46 *IObjectWithSite::SetSite*

HRESULT SetSite([in] IUnknown *pUnkSite);

Provides the site's *IUnknown* pointer to the object. The object should hold onto this pointer, calling *AddRef* in doing so. If the object already has a site, it should first call *pUnkSite->AddRef* to secure the new site, call *IUnknown::Release* on the existing site, the save *pUnkSite*.

Argument	Type	Description
pUnkSite	*IUnknown **	[in] The interface pointer of the site managing this object. If NULL, the object should call *IUnknown::Release* on any existing site at which point the object no longer knows its site.

Return Value	Meaning
S_OK	Returned in all circumstances

Comments:
E_NOTIMPL is disallowed—without implementation of *SetSite,* the *IObjectWithSite* interface is unnecessary.

47 *IObjectWithSite::GetSite*

HRESULT GetSite([in] REFIID riid, [out, iid_is(riid)] void **ppvSite);

Retrieves the last site set with *IObjectWithSite::SetSite.* If there's no known site, the object return a failure code.

Argument	Type	Description
riid	*REFIID*	[in] The IID of the interface pointer that should be returned in *ppvSite.*
ppvSite	*void ** *	[out] The address of the caller's *void ** variable in which the object stores the interface pointer of the site last seen in *IObjectWithSite::SetSite*. The specific interface returned depends in the *riid* argument—in essence, the two arguments act identically to those in *QueryInterface*. If the appropriate interface pointer is available, the object must call *AddRef* on that pointer before returning successfully. If no site is available, or the requested interface is not supported, the object sets this argument to NULL and returns a failure code.

Return Value	Meaning
S_OK	The site was returned successfully and the caller is responsible for calling ((*IUnknown* *)*(*ppvSite))->Release()* when the site is no longer needed
E_FAIL	There is no site in which case **ppvSite* contains NULL on return.
E_NOINTERFACE	There is a site but the it does not support the interface requested by *riid*.

Comments:
E_NOTIMPL is disallowed—any object implementing this interface must be able to return the last site seen in *SetSite.*

© Microsoft Corporation,1995-1996, All Rights Reserved.

48 The *IPersistMemory* Interface

IPersistMemory operates exactly as *IPersistStreamInit*, except that it allows the caller to provide a fixed-size memory block (identified with a *void* *) as opposed to *IPersistStreamInit* which involves an arbitrarily expandable *IStream*.

IDL:

```
[
uuid(BD1AE5E0-A6AE-11CE-BD37-504200C10000)
    , object, pointer_default(unique)
]
interface IPersistMemory : IPersist
    {
    HRESULT IsDirty(void);
    HRESULT Load([in, size_is(cbSize)] void *pvMem, [in] ULONG cbSize);
    HRESULT Save([in,out, size_is(cbSize)] void *pvMem, [in] BOOL fClearDirty, [in] ULONG
        cbSize);
    HRESULT GetSizeMax([out] ULONG* pcbSize);
    HRESULT InitNew(void);
    };
```

The *cbSize* argument to *Load* and *Save* indicate the amount of memory accessible through *pvMem*.

The *IsDirty, GetSizeMax,* and *InitNew* members are semantically and syntactically identical to those in *IPersistStreamInit*. Only *Load* and *Save* differ as described below.

49 *IPersistMemory::Load*

HRESULT IPersistMemory::Load([in] void *pvMem, [in] ULONG cbSize);

Instructs the object to load its persistent data from the memory pointed to by *pvMem* where *cbSize* indicates the amount of memory at *pvMem*. The object must not read past the address *(BYTE*)((BYTE *)pvMem+cbSize)*.

Argument	Type	Description
pvMem	*void **	[in] The address of the memory from which the object can read up to *cbSize* bytes of its data.
cbSize	*ULONG*	[in] The amount of memory available at *pvMem* form which the object can read its data.

Return Value	Meaning
S_OK	The object successfully loaded its data.
E_UNEXPECTED	This member was called after the object was already initialized with *IPersistMemory::Load*. Only one initialization is allowed per instance.
E_POINTER	The pointer in *pvMem* is NULL.

Comments:
Any object that implements *IPersistMemory* has some information to load persistently, therefore E_NOTIMPL is not a valid return code.

50 *IPersistMemory::Save*

HRESULT IPersistMemory::Save([in] void *pvMem, [in] BOOL fClearDirty, [in] ULONG cbSize);

Instructs the object to save its persistent data to the memory pointed to by *pvMem* where *cbSize* indicates the amount of memory available at *pvMem*. The object must not write past the address *(BYTE*)((BYTE *)pvMem+cbSize)*. The *fClearDirty* flag determines whether the object is to clear its dirty state after the save is complete.

Argument	Type	Description
pvMem	*void **	[in] The address of the memory in which the object should save up to *cbSize* bytes of its data.
fClearDirty	*BOOL*	[in] A flag indicating whether the object should clear its dirty state on return from *Save* or leave that state as-is.
cbSize	*ULONG*	[in] The amount of memory available at *pvMem* to which the object can write its data.

Return Value	Meaning
S_OK	The object successfully initialized itself.
E_UNEXPECTED	This member was called before the object was initialized with *IPersistMemory::InitNew* or *IPersistMemory::Load*.
E_INVALIDARG	The number of bytes indicated by *cbSize* is too small to allow the object to save itself completely.
E_POINTER	The pointer in *pvMem* is NULL.

Comments:
Any object that implements *IPersistMemory* has some information to save persistently, therefore E_NOTIMPL is not a valid return code.

The caller should ideally allocate as many bytes as the object returns from *IPersistMemory::GetSizeMax*.

51 The *IPersistPropertyBag* Interface

The *IPersistPropertyBag* interface works in conjunction with *IPropertyBag* (see 9.6 below) and *IErrorLog* (see 9.7 below) to define an individual property-based persistence mechanism. Whereas a mechanism like *IPersistStream* gives an object an *IStream* in which to store its binary data, *IPersistPropertyBag* provides an object with an *IPropertyBag* interface through which it can save and load individual properties. The implementor of *IPropertyBag* can then save those properties in whatever way it chooses, such as name/value pairs in a text file. Errors encountered in the process (on either side) are recorded in an "error log" through *IErrorLog*. This error reporting mechanism work on a per-property basis instead of an "all properties as a whole" basis through just the return value of *IPersist*::Load* or *IPersist*::Save*.

Current Sweeper components do not use IErrorLog in any way.

IDL:

```
[
uuid(37D84F60-42CB-11CE-8135-00AA004BB851)
    , object, pointer_default(unique)
]
interface IPersistPropertyBag : IPersist
    {
    HRESULT InitNew(void);
    HRESULT Load([in] IPropertyBag *pPropBag, [in] IErrorLog *pErrorLog);
    HRESULT Save([in] IPropertyBag *pPropBag, [in] BOOL fClearDirty
        , [in] BOOL fSaveAllProperties);
    };
```

The basic mechanism is that a container tells the object to save or load its properties through *IPersistPropertyBag*. For each property, the object calls the container's *IPropertyBag* interface passed to the *IPersistPropertyBag* members.

© Microsoft Corporation,1995-1996, All Rights Reserved.

IPropertyBag::Write obviously saves a property in whatever place the container wants to put it, and *IPropertyBag::Read* retrieves a property.

This protocol is essentially a means of sequentially communicating individual property values from the object to the container, which is useful for doing save-as-text operations and the like. The object gives the container the choice of the format in which each property is saved, while retaining itself the decision as to *which* properties are saved or loaded.

52 *IPersistPropertyBag::InitNew*

```
HRESULT InitNew(void);
```

Informs the object that it is being initialized as a newly created object.

Argument	Type	Description
none		

Return Value	Meaning
S_OK	The object successfully initialized itself. This should be returned even if the object doesn't do anything in the function.
E_UNEXPECTED	This member was called after *IPersistPropertyBag::Load* or *IPersistPropertyBag::Save*.

Comments:
E_NOTIMPL should not be returned—use S_OK when the object has nothing to do in the function.

53 *IPersistPropertyBag::Load*

```
HRESULT Load([in] IPropertyBag *pPropBag, [in] IErrorLog *pErrorLog);
```

Instructs the object to initialize itself using the properties available in the property bag, notifying the provided error log object when errors occur. All property storage must take place within this function call as the object cannot hold the *IPropertyBag* pointer.

Argument	Type	Description
pPropBag	*IPropertyBag* *	[in] A pointer to the caller's property bag through which the object can read properties. Cannot be NULL.
pErrorLog	*IErrorLog* *	[in] A pointer to the caller's error log in which the object stores any errors that occur during initialization. Can be NULL in which case the caller is not interested in errors.

Return Value	Meaning
S_OK	The object successfully initialized itself.
E_UNEXPECTED	This member was called after *IPersistPropertyBag::InitNew* has already been called. They two initialization methods are mutually exclusive.
E_POINTER	The address in *pPropBag* is not valid (such as NULL) and therefore the object cannot initialize itself.
E_FAIL	The object was unable to retrieve a critical property that is necessary for the object's successful operation. The object was therefore unable to initialize itself completely.

Comments:
E_NOTIMPL is not a valid return code as any object implementing this interface must support the entire functionality of the interface.

54 *IPersistPropertyBag::Save*

HRESULT Save([in] IPropertyBag *pPropBag, [in] BOOL fClearDirty, [in] BOOL fSaveAllProperties);

Instructs the object to save its properties to the given property bag, optionally clearing the object's "dirty" flag. The caller can request that the object save all properties or that the object save only those that are known to have changed.

Argument	Type	Description
pPropBag	*IPropertyBag* *	[in] A pointer to the caller's property bag through which the object can write properties. Cannot be NULL.
fClearDirty	*BOOL*	[in] A flag indicating whether the object should clear its dirty flag when saving is complete. TRUE means clear the flag, FALSE means leave the flag unaffected. FALSE is used when the caller wishes to do a "Save Copy As" type of operation.
fSaveAllProperties	*BOOL*	[in] A flag indicating whether the object should save all its properties (TRUE) or only those that have changed since the last save or initialization (FALSE).

Return Value	Meaning
S_OK	The object successfully saved the requested properties itself.
E_FAIL	There was a problem saving one of the properties. The object can choose to fail only if a necessary property could not be saved, meaning that the object can assume default property values if a given property is not seen through *Load* at some later time.
E_POINTER	The address in *pPropBag* is not valid (such as NULL) and therefore the object cannot initialize itself.

Comments:
E_NOTIMPL is not a valid return code as any object implementing this interface must support the entire functionality of the interface.

55 The *IPropertyBag* Interface

When a client wishes to have exact control over how individually named properties of an object are saved, it would attempt to use an object's *IPersistPropertyBag* interface as a persistence mechanism. In that case the client supplies a "property bag" to the object in the form of an *IPropertyBag* interface.

IDL:

```
[
uuid(55272A00-42CB-11CE-8135-00AA004BB851)
    , object, pointer_default(unique)
]
interface IPropertyBag : IUnknown
    {
    HRESULT Read([in] LPCOLESTR pszPropName, [in,out] VARIANT *pVar
        , [in,out] IErrorLog *pErrorLog);
    HRESULT Write([in] LPCOLESTR pszPropName, [in] VARIANT *pVar);
```

© Microsoft Corporation,1995-1996, All Rights Reserved.

```
    };
```

When the object wishes to read a property in *IPersistPropertyBag::Load* it will call *IPropertyBag::Read*. When the object is saving properties in *IPersistPropertyBag::Save* it will call *IPropertyBag::Write.* Each property is described with a name in *pszPropName* whose value is exchanged in a *VARIANT*. This information allows a client to save the property values as text, for instance, which is the primary reason why a client might choose to support *IPersistPropertyBag*.

The client records errors that occur during *Read* into the supplied "error log."

56 *IPropertyBag::Read*

HRESULT Read([in] LPCOLESTR pszPropName, [in,out] VARIANT *pVar, [in,out] IErrorLog *pErrorLog);

Asks the property bag to read the property named with *pszPropName* into the caller-initialized *VARIANT* in *pVar*. Errors that occur are logged in the error log pointed to by *pErrorLog.* When *pVar->vt* specifies another object pointer (VT_UNKNOWN) then the property bag is responsible for creating and initializing the object described by *pszPropName*. See *IPropertyBag::Write* for more details.

Argument	Type	Description
pszPropName	*LPCOLESTR*	[in] The name of the property to read. Cannot be NULL.
pVar	*VARIANT **	[in, out] The address of the caller-initialized *VARIANT* that is to receive the property value on output. The function must set both type and value fields in the *VARIANT* before returning. If the caller initialized the *pVar->vt* field on entry, the property bag should attempt to coerce the value it knows into this type. If the caller sets *pVar->vt* to VT_EMPTY, the property bag can use whatever type is convenient.
pErrorLog	*IErrorLog **	[in] A pointer to the caller's error log in which the property bag stores any errors that occur during reads. Can be NULL in which case the caller is not interested in errors.

Return Value	Meaning
S_OK	The property was read successfully. The caller becomes responsible for any allocations that are contained in the *VARIANT* in *pVar*.
E_POINTER	The address in *pszPropName* is not valid (such as NULL).
E_INVALIDARG	The property named with *pszPropName* does not exist in the property bag.
E_FAIL	The property bag was unable to read the specified property, such as if the caller specified a data type to which the property bag could not coerce the known value. If the caller supplied an error log, a more descriptive error was sent there.

Comments:
E_NOTIMPL is not a valid return code as any object implementing this interface must support the entire functionality of the interface.

57 *IPropertyBag::Write*

HRESULT Write([in] LPCOLESTR pszPropName, [in] VARIANT *pVar);

Asks the property bag to save the property named with *pszPropName* using the type and value in the caller-initialized *VARIANT* in *pVar*. In some cases the caller may be asking the property bag to save another object, that is, when *pVar->vt* is VT_UNKNOWN. In such cases, the property bag queries this object pointer for some persistence interface, like *IPersistStream* or even *IPersistPropertyBag* again, and has that object save its data as well. Usually this results in the property bag having some byte array for this object which can be saved as encoded text (hex string, MIME, etc.). When the property bag is later used to reinitialize a control, the client that owns the property bag must recreate the object when the caller asks for it, initializing that object with the previously saved bits.

This allows very efficient persistence operations for large BLOB properties like a picture, where the owner of the property bag itself directly asks the picture object (managed as a property in the control being saved) to save into a specific location. This avoids potential extra copy operations that would be involved with other property-based persistence mechanisms.

Argument	Type	Description
pszPropName	*LPCOLESTR*	[in] The name of the property to write. Cannot be NULL.
pVar	*VARIANT **	[in] The address of the caller-initialized *VARIANT* that holds the property value to save. The caller owns this *VARIANT* and is responsible for all allocations therein. That is, the property bag itself does not attempt to free data in the *VARIANT*.

Return Value	Meaning
S_OK	The property bag successfully saved the requested property.
E_FAIL	There was a problem writing the property. It is possible that the property bag does not understand how to save a particular *VARIANT* type.
E_POINTER	The address in *pszPropName* or *pVar* is not valid (such as NULL). The caller must supply both.

Comments:
E_NOTIMPL is not a valid return code as any object implementing this interface must support the entire functionality of the interface.

58 The *IErrorLog* Interface

> *Current Sweeper components do not use IErrorLog in any way.*

The *IErrorLog* interface is an abstraction for an "error log" that is used to communicate detailed error information between a client and an object. The caller of the single interface member, *AddError*, simply "logs" an error where the error is an *EXCEPINFO* structure attached to a specific property. The implementor of the interface is responsible for handling the error in whatever way it desires.

IErrorLog is used in the protocol between a client that implements *IPropertyBag* and an object that implements *IPersistPropertyBag*.

IDL:

```
[
uuid(3127CA40-446E-11CE-8135-00AA004BB851)
    , object, pointer_default(unique)
]
interface IErrorLog : IUnknown
    {
    HRESULT AddError([in] LPCOLESTR pszPropName, [in] LPEXCEPINFO pExcepInfo);
    };
```

© Microsoft Corporation,1995-1996, All Rights Reserved.

59 *IErrorLog::AddError*

HRESULT AddError([in] LPCOLESTR pszPropName, [in] LPEXCEPINFO pExcepInfo);

Logs an error (an *EXCEPINFO* structure) in the error log for a named property.

Argument	Type	Description
pszPropName	*LPCOLESTR*	[in] The name of the property involved with the error. Cannot be NULL.
pExcepInfo	*EXCEPINFO **	[in] The address of the caller-initialized *EXCEPINFO* structure that describes the error to log. Cannot be NULL.

Return Value	Meaning
S_OK	The error was logged successfully.
E_FAIL	There was a problem logging the error.
E_OUTOFMEMORY	There was not enough memory to log the error.
E_POINTER	The address in *pszPropName* or *pExcepInfo* is not valid (such as NULL). The caller must supply both.

Comments:
E_NOTIMPL is not a valid return code as the member function is the only one in the entire interface.