Chapter Six
Uniform Data Transfer through Data Objects

Have you ever flown up in an airplane and thought about all the little houses down there? If you look at them really close, they're just little piles of people's stuff.

Comedian George Carlin

Back in the early stages of Chapter 3 I defined an generic object as an instantiation of some class where that class is the definition a data structure and functions that manipulate that structure. A Windows Object, to be precise, could only be seen from the user of that object as the functions, collected into interfaces. But like all other objects, Windows Objects do have data associated with them, the object's stuff. The questions is, will that object let you look at its stuff?

A "data object", or the *source* of data, is a general term for any other Windows Object that you treat though the single standard data transfer interface: IDataObject. If some Windows Object supports this interface, then you can call that object a Data Object and treat it as anything else you can call a data object. So what we mean by "data object" is different than the storage and stream objects we saw in the last chapter. To further reinforce this fact, there really isn't one single way for the object user, or *consumer* of the data, to obtain an IDataObject pointer to any given object; to be honest, you can obtain such a pointer by a variety of mechanisms of which two, clipboard and drag-drop, are the subjects of the next two chapters.

That leaves us here in this chapter to define what data objects are and how they behave through the semantics of the IDataObject interface. As we'll see, all of the functionality currently distributed between the Windows APIs for the clipboard, DDE, and OLE 1.0 is collected together into one uniform IDataObject interface. So no matter what mechanism you use to obtain the pointer to a data object, you can then treat that source of data in a very standard way which is exactly why I coined the term "Uniform Data Transfer." OLE 2.0 standardizes the use of a data object separating it from what I call a transfer protocol.

A transfer protocol is some mechanism for transferring an IDataObject pointer from the source of the data to the consumer of the data. More generically, the protocol is some mechanism to transfer information about the data, that is, to set up some standardized dialog between the source and the consumer such that they agree on the data under scrutiny. Up to this time the actual transfer of real tangible data has been tightly bound with the protocol used to discuss the data. In OLE 2.0, the functions of requesting (or setting) data are divorced from all protocol which overall simplifies and homogenizes your application's data dealings.

Another problem with existing Windows APIs for data transfer is both source and consumer must limit their conversations about the data to a single UINT of a clipboard format that only describe the data structure. In addition, the only standard medium in which that data structure can reside is global memory–no standards exists for storing and communicating data in files or other types of storage. OLE 2.0 introduces two new data structures that essentially provide a better clipboard format and a better global memory, enabling far richer description of data than a UINT and far more mediums on which to transfer that data than a single HGLOBAL. Since these structures are essential for both data object and object user, they will be an early topic, then followed by an implementation of a data object as a component object such as those in Chapter 4.

The user of a data object may also be interested in notifications when the data changes in the source, that is, the consumer may want to establish a link with the source. Watch it! This is a different idea than a *linked compound document object*. In this context a link is simply a notification mechanism established between the source and user, not a reference to data that exists in another file which is the topic of compound document links in Chapters 12 and 13. OLE 2.0 provides an interface called IAdviseSink and a few member functions in IDataObject to enable both 'hot' (data sent with the notification) and 'warm' (only a notification of change) data links such as those you can create today with DDE. In this chapter we will only look at two member functions in IAdviseSink as the others must wait until we discuss compound documents. Since data objects and DDE have similarities along these veins, the last section of this chapter will discuss the important similarities and differences between these two expressions of data transfer.

Any given data object, which knows how to render its data in specific formats, may also know how to draw a graphical representation on some device context. This chapter will also take a look at an interface called IViewObject which contains this functionality and will explain a few things about a data cache that lives inside OLE 2.0. Much of this discussion will become more significant when we deal with compound documents, but what we can exploit now is that the default IViewObject implementation inside OLE2.DLL knows how to draw bitmaps and metafiles. In addition, OLE2.DLL also knows how to serialize the same presentations to a storage object. This chapter will demonstrate how you can 'freeload' off OLE 2.0 to handle these normally complex functions, exchanging the tedium of painting an d file I/O with bitmaps and metafiles

with a few calls to OLE 2.0 APIs and interfaces.

Again this chapter will answer the question about how you obtain a pointer to some data object, but only for data objects that are also component objects. How to obtain a data object representing the clipboard, that is the OLE 2.0 clipboard protocol, is the subject of Chapter 7. How to obtain a data object involved through the OLE 2.0 drag-drop protocol is the subject of Chapter 8. Data objects that represent data in compound document scenarios are the topics of Chapters 9-11. But regardless of what protocol you use, you can still treat all these data objects the same. No matter how high you're flying, Uniform Data Transfer and the IDataObject interface lets you see every object down below as a little pile of stuff.

What is a Data Object?

Somewhere something, some Windows Object, has some data, and you want to retrieve renderings of that data. But how, again, do you obtain an IDataObject pointer on some arbitrary source? In Chapter 4 we saw Windows Objects that were identified with a CLSID and instantiated through functions like CoCreateInstance and IClassFactory::CreateInstance. That is only one of the four methods by which you can obtain a pointer as listed in "The Ultimate Question to Life, the Universe, and Objects" in Chapter 3. Chapter 5 looked at storage and stream objects in Compound Files that were obtained either by explicit APIs in STORAGE.DLL or by calling a member function in the IStorage interface.

In all reality, you might obtain an IDataObject pointer through any of the four methods. Which one you use depends greatly on the context in which you identify the source of the data you want. In some cases, as will be demonstrated in this chapter, the data object may be it's own entity identified with a CLSID. In this case it's not that the CLSID identifies some magical "data object" but instead identifies some object for which IDataObject is the primary interface.

For example, a stand-alone data object in a DLL may be the best way to expose data collection functions of a very specialized piece of hardware you might install in your computer. Normally this sort of board would be shipped with some DLL that exports a propriety API through which you could access the data. With OLE 2.0, such a board could ship with a Windows Object in a DLL with its own CLSID such that anyone interested in accessing the data need only call CoCreateInstance with the CLSID to set up the access (instead of using some other API) and would use the member functions of IDataObject to actually retrieve the data. The result? Fewer new APIs for everyone to design, implement, or learn. Furthermore, a Component Data Object like this, when no outside application has any dependency on specific APIs, *is replaceable at will*, that is, component objects that deal exclusively through well-published interfaces are truly plug-and-play components!

But I digress–I really wanted to point out that a data object can be a very specialized object, or it can simply be the expression of some other type of object, such as a compound document object, as a source of data. In this case you might obtain a pointer to IDataObject through QueryInterface. A data object might also be the object representation of some store of data, like the clipboard, and for such cases you might call a specific OLE 2.0 API to obtain the IDataObject pointer. When that data is not in storage, per se, but is being dumped on you as in a drag-drop operation, you might receive the IDataObject pointer as a parameter to a member function you implement as a drag-drop target as shown in Chapter 8.

So as you can see, data objects are just expressions of data sources through IDataObject. But before we can get into the specifics of the interface itself, let's look at how OLE 2.0 allows you to describe data and the mediums on which you can transfer data through two new structures, FORMATETC and STGMEDIUM that are defined in DVOBJ.H in the OLE 2.0 Toolkit. Note that DVOBJ.H generally contains all the definitions relevant to this chapter.

New and Improved Ultra-Structures!

The following is a paid commercial announcement.

> Hello friends, Ole' Bob Data here asking you to perk up your ears and listen to what I have to say to you today. Friends, are you troubled by only being allowed to describe data using a lousy little clipboard format? Are you troubled by only being allowed to exchange data in a crummy global memory handle? Are you sick and tired of waiting around while you try to copy a thirty-megabyte-twenty-four-bit-device-independent-bitmap making your disk chug like grandma's old Hoover?! Well then listen to me friends, for what I have for you today will put and end to your misery and put an end to hard disk swapping that sounds like a Studabaker lug nut in a meat grinder! It's New and Improved Ultra-Structures, free with every purchase of OLE 2.0 and free with every copy of a data object! No longer do you just say "bitmap"! No longer do you just say "metafile"! Be free! Be fresh! Send me your paycheck! Tell your data object that you want a bitmap but you want just a

thumbnail sketch! Tell your data object that you want a metafile but you want it created for a PostScript device! Tell your data object that you want every known translation of the Bible, the Koran, and the Collected Sayings of Mao-Tse Tung but not just a lousy temperamental piece of global memory but in a compound file!!! The choice is yours! But how much are you willing to pay? Five APIs? Twenty APIs? A hundred new API? NO! These are your free with your qualified use of a data object! Available at a data object near you.

Taxlicensinganddestinationchangesapplicablebutvoidwhereprohibitedanddoesnotincludedealerprepmarkuportheoverheadinpaying forridiculousadvertisementslikethisorthetendollarsaminuteweautomaticallychargetoeverycreditcardinyournameaswellasallyourhouseholdutilitiesincludingphonepowerandcableTVjustforlisteningtous.

Now back to our scheduled programming.

All versions of Windows since version 1.0 back in 1985 have only described standardized data transfers (that is, through system protocols like clipboard and DDE) using a simple clipboard format and a global memory handle. When you copy data to the clipboard you call SetClipboardData passing a clipboard format and a global memory handle. To paste, you call GetClipboardData with a clipboard format and again, get back a global memory handle. Dynamic Data Exchange, DDE, is restricted in the same fashion: WM_DDE_DATA messages only carry with them a global memory handle containing the data and an item that describes what might be in that memory.

OLE 1.0 suffered greatly by restricting itself to global memory data transfers. As happens with the clipboard and DDE, many copies of the same data generally exist in memory at any given moment. Small data sets are never a problem, but when an OLE 1.0 server supplied an object containing a large 24-bit DIB, that data had to be contained both in a metafile and in the object's 'native' data, then shuffled to the container application over DDE during which another copied might be made. The container itself had to store that object data somewhere in memory, and eventually to its own disk file. This led to highly inefficient use of memory and poor performance. Something had to change.

Not only did the OLE 2.0 architects have to contend with the efficiency problem, but they were also faced with adding the drag-drop data protocol for transfer of any arbitrary data wherever the clipboard could be used. They must have asked themselves "Should we just do this through clipboard formats and global memory? Should OLE 2.0's compound document protocol continue to be hampered by multiple copies of the same data in different places? Could we let OLE 2.0 remain incapable of even describing the device for which data was created?"

This mess just couldn't continue to frustrate endless developers, and thus it became necessary to expand on the idea of a data in global memory described by a clipboard format, an expansion that can still make use of most of the code you already have for dealing with the clipboard or other transfer protocols. Enter two new structures, FORMATETC and STGMEDIUM.

FORMATETC (pronounced "format etcetera") is a generalization, and improvement, of the clipboard format and contains a rich description of data. The name comes from the idea that it contains a clipboard format, and, uh, well, some more stuff, the etcetera:

- *cfFormat* (UINT): The clipboard format identifying the internal structure of the data. This format can be standard, like CF_TEXT, or a registered format that both source and consumer register.
- *ptd* (LPTARGETDEVICE): Information about the device for which the data was rendered, such as a screen or printer, contained in a TARGETDEVICE structure that look and acts similar to a DEVNAMES structure:

```
typedef struct FARSTRUCT tagDVTARGETDEVICE
  {
  DWORD   tdSize;
  WORD    tdDriverNameOffset;
  WORD    tdDeviceNameOffset;
  WORD    tdPortNameOffset;
  WORD    tdExtDevmodeOffset;
  BYTE    tdData[1];   //Contains the names and DEVMODE
  } DVTARGETDEVICE;
```

- *dwAspect* (DWORD): How much detail is contained in the rendering, either the full content (DVASPECT_CONTENT) as would normally be shown in some kind of document, a thumbnail sketch (DVASPECT_THUMBNAIL) that would be used in a print preview or document preview window, and icon (DVASPECT_ICON) appropriate for small presentations such as in email messages, or a full "printer document" (DVASPECT_DOCPRINT) that includes all page numbers, headers, footers, just as if the data was printed as a document from its native application..
- *lindex* (LONG): Identifier for the 'piece' of the data when the data must be split across page

boundaries. An *lindex* of -1 identifies the entire data and is the most common value. Otherwise *lindex* only has meaning in DVASPECT_CONTENT, where it identifies a piece for extended layout negotiation, and in DVASPECT_DOCPRINT where it identifies the page number.
NOTE: Page-layout capabilities are not supported in OLE version 2.0 but will be implemented in future versions of the libraries. Therefore the lindex in any FORMATETC should always be -1. The debug OLE 2.0 libraries will display assertion failures if you forget.

- *tymed* (DWORD): The medium in which the data lives. See STGMEDIUM below.

Obviously filling out an array like this every time you want to describe a data format will get tedious, having to create five lines of code just to fill the structure. For this reason I have defined two macros in INC\BOOKGUID.H (which is included by all samples after Chapter 2) that facilitate filling a FORMATETC. SETFormatEtc that allows you to set every field in a FORMATETC structure explicitly, and SETDefFormatEtc that allows you to set *cfFormat* and *tymed* while filling the other fields with defaults:

```
#define SETFormatEtc(fe, cf, asp, td, med, li)  \
   {\
   (fe).cfFormat=cf;\
   (fe).dwAspect=asp;\
   (fe).ptd=td;\
   (fe).tymed=med;\
   (fe).lindex=li;\
   };

#define SETDefFormatEtc(fe, cf, med)  \
   {\
   (fe).cfFormat=cf;\
   (fe).dwAspect=DVASPECT_CONTENT;\
   (fe).ptd=NULL;\
   (fe).tymed=med;\
   (fe).lindex=-1;\
   };
```

I encourage you to use these macros to make your data object programming life easier. You frequently need to fill FORMATETC structures and these macros conveniently reduce the filling to one line. The OLE 2.0 Toolkit also has a file OLESTD.H that defines similar macros (SETFORMATETC and SETDEFFORMATETC). I use my own definitions because many of the samples in this book do not have occasion to use anything else in OLESTD.H but will generally use BOOKGUID.H.

FORMATETC is only half of the picture, of course, because it only describes what is contained in some actual rendering of the data. We still need some reference to that data so STGMEDIUM ("storage medium") is a generalization of the global memory handle, holds a mixture of different data references:

- *tymed* (DWORD): An identifier for the type of medium used: global memory (TYMED_HGLOBAL), disk file (TYMED_FILE), storage object (TYMED_ISTORAGE), stream object (TYMED_ISTREAM), GDI object (TYMED_GDI), METAFILEPICT (TYMED_MFPICT), or undefined (TYMED_NULL).
- *hGlobal-lpszFileName-pStg-pStm* (union of HGLOBAL, LPSTR, LPSTORAGE, and LPSTREAM): A reference to the actual data the meaning of which is defined by *tymed*. Medium types TYMED_HGLOBAL, TYMED_GDI, and TYMED_MFPICT store their memory or GDI handles in *hGlobal*. TYMED_FILE stores a pointer to the filename in *lpszFileName*, and TYMED_ISTORAGE and TYMED_ISTREAM store pointers to their objects in *pStg* and *pStm*, respectively.
- *punkForRelease* (LPUNKNOWN): If NULL then the owner of the STGMEDIUM (typically the consumer) must free the memory and other allocations contained in the structure by calling ReleaseStgMedium. If non-NULL, the owner must call *punkForRelease->Release()* to allow some other unidentifiable agent to perform the cleanup.

The FORMATETC and STGMEDIUM structures open up a wide range of possibilities in data transfer and solve a number of the key problems with previous protocols and data transfer techniques. The most fundamental benefit of these richer descriptions is that data no longer *has* to live in global memory. If the data is best suited to live in a disk file, then you can describe that fact using TYMED_FILE. If the data is best suited to living in a storage or stream object you can express that through TYMED_ISTORAGE and TYMED_ISTREAM. For example, very large bitmaps that do not fit in memory can be kept by the source application in a storage object (that is, a compound file) on disk. When another application wants a copy of a

bitmap, the source can copy that data to a new temporary compound file (using StgCreateDocfile(NULL, ...) and IStorage::CopyTo) and pass the IStorage pointer in a STGMEDIUM structure. The consumer receives the marshaled IStorage pointer and can incrementally access that bitmap as necessary, for perhaps the entire bitmap is not needed all at once. So in all, very little memory is used opting instead for generally more available disk space. Furthermore such large data will typically end up on disk anyway, so it makes complete sense to put it there in the first place.

Consumers of data, that is, of a STGMEDIUM, usually become responsible to free the data when they are done with it. One potential difficulty with the richness of STGMEDIUM is figuring out just how to free whatever might be in it. Already you should have the image of a big *switch(stm.tymed)* statement floating in your head that would call the right API depending on the actual data reference in the structure. Don't bother– OLE 2.0 provides a single API to perform cleanup on any STGMEDIUM: ReleaseStgMedium. The actual calls made to free the STGMEDIUM varies with the medium type:

tymed	Freeing Mechanism
any	If *punkForRelease* is non-NULL, *punkForRelease->Release()* is always called.
TYMED_HGLOBAL	GlobalFree(*hGlobal*)
TYMED_FILE	OpenFile(*lpszFileName*, &of, OF_DELETE)
TYMED_ISTORAGE	*pStg*->Release()
TYMED_ISTREAM	*pStm*->Release()
TYMED_GDI	DeleteObject((HGDIOBJ)*hGlobal*)
TYMED_MFPICT	LPMETAFILEPICT pMFP: pMFP=GlobalLock(*hGlobal*); DeleteMetaFile(pMFP->hMF); GlobalUnlock(*hGlobal*) GlobalFree(*hGlobal*)

Providing the ReleaseStgMedium API is exactly why the *tymed* fields of both STGMEDIUM and FORMATETC differentiate between handles for global memory, GDI objects, and metafile pictures: only with such precise identification can ReleaseStgMedium know how to perform cleanup correctly.

Data Objects and the IDataObject Interface

Any agent that can be considered a source of data can describe their data using a data object on which is implemented the IDataObject interface. A data object is one view of a compound document object as we'll see in Chapters 9-11, but any and all code that in one way or another has data to share can describe that data as a data object through the IDataObject. The great benefit of doing such is that once you have a *single* data object around, you can use that data object in any transfer protocol, be it clipboard, drag-drop, or compound documents. As a source, you will centralize the code that renders data into this data object. As a consumer you will centralize the code necessary to check whether the data is actually usable and the code used to paste that data. Such centralization reduces the overall amount of code you must implement as well as reducing the number of differing APIs for dealing with each protocol.

Centralization is possible because the IDataObject interface combines the functionality of the existing data transfer protocols, thereby providing more functionality in a data transfer than is available for any single existing protocol. The definition of the IDataObject interface is as follows (the ubiquitous IUnknown members are omitted):

```
DECLARE_INTERFACE_(IDataObject, IUnknown)
  {
  [IUnknown methods included]

  //IDataObject methods
  STDMETHOD(GetData) (THIS_ LPFORMATETC pformatetcIn,
    LPSTGMEDIUM pmedium ) PURE;
  STDMETHOD(GetDataHere) (THIS_ LPFORMATETC pformatetc,
    LPSTGMEDIUM pmedium ) PURE;
  STDMETHOD(QueryGetData) (THIS_ LPFORMATETC pformatetc ) PURE;
  STDMETHOD(GetCanonicalFormatEtc) (THIS_ LPFORMATETC pformatetc,
    LPFORMATETC pformatetcOut) PURE;
  STDMETHOD(SetData) (THIS_ LPFORMATETC pformatetc,
    STGMEDIUM FAR * pmedium, BOOL fRelease) PURE;
  STDMETHOD(EnumFormatEtc) (THIS_ DWORD dwDirection,
```

```
    LPENUMFORMATETC FAR* ppenumFormatEtc) PURE;

STDMETHOD(DAdvise) (THIS_ FORMATETC FAR* pFormatetc, DWORD advf,
    LPADVISESINK pAdvSink, DWORD FAR* pdwConnection) PURE;
STDMETHOD(DUnadvise) (THIS_ DWORD dwConnection) PURE;
STDMETHOD(EnumDAdvise) (THIS_ LPENUMSTATDATA FAR* ppenumAdvise) PURE;
};

typedef     IDataObject FAR* LPDATAOBJECT;
```

Many of the member functions have equivalents in specific Windows APIs; keep in mind, however, that data objects are used to describe data transferred via *any* protocol, and thus provides the ability to treat data in a uniform fashion regardless of how you obtained it. The list below described each IDataObject member in a little more detail and lists the similar (but not always exact) functionality that currently exists in the clipboard, DDE, and OLE 1.0 transfer protocols. Drag-drop is not present in these lists since it's a new feature provides in OLE 2.0.

- **::GetData** renders the data described by a FORMATETC returning it in the STGMEDIUM which is then the caller's responsibility.

Protocol	Similar API, Message
Clipboard	GetClipboardData
DDE	WM_DDE_REQUEST, WM_DDE_DATA
OLE 1.0	OleGetData

- **::SetData** provides data to the source described by a FORMATETC and referenced by a STGMEDIUM. The data object s responsible for the releasing the data if the fRelease flag is TRUE.

Protocol	Similar API, Message
Clipboard	SetClipboardData
DDE	WM_DDE_POKE
OLE 1.0	OleSetData

- **::GetDataHere** allows the caller to provide an already allocated medium into which the data object should render the data. For example, if the caller provides a global memory handle and asks for CF_TEXT the object should copy its textual data into that memory instead of allocating new memory as is done in ::GetData (failing if the caller's memory is too small, of course). This is a capability not found in any existing data transfer mechanisms.
- **::QueryGetData** answers whether the data object is capable of rendering data described by a specific FORMATETC structure. The caller may be as specific as desired. QueryGetData returns NOERROR for 'yes,' S_FALSE for 'no,' so be careful not to use the SUCCEEDED or FAILED macros to test return values.

Protocol	Similar API, Message
Clipboard	IsClipboardFormatAvailable
DDE	None (perhaps handled through WM_DDE_CONNECT, ADVISE)
OLE 1.0	None

- **::GetCanonicalFormatEtc** provides a different but logically equivalent FORMATETC structure allowing the caller to determine if a rendering it already has obtained is identical as what would be obtained by calling ::GetData with a different FORMATETC. Use of this function can eliminate unnecessary calls to ::GetData. Note that the *tymed* fields in the FORMATETC structure is irrelevant here and so should be ignored. There is no equivalent to this function in any existing protocol.
- **::EnumFormatEtc** instantiates and returns a FORMATETC enumerator through which the caller may determine all available FORMATETCs that the object can *currently* provide. There must be a unique element in the enumeration for each *cfFormat*, *dwAspect*, and *ptd* variation although you can combine TYMED_* values in *tymed*. The enumerator object implements the single IEnumFORMATETC interface

and the caller is responsible for calling IEnumFORMATETC::Release when done to allow the object to free itself. The caller can ask for an enumerator for either direction ::GetData or ::SetData. See "FORMATETC Enumerators and Format Ordering" below for more details on this enumerator.

Protocol	Similar API, Message
Clipboard	EnumClipboardFormats (Get direction only)
DDE	None
OLE 1.0	None

- **::DAdvise**[1] sets up an advisory connection between the data object and a caller-provided advise sink where the caller indicates the data of interest in a FORMATETC. The data object calls IAdviseSink::OnDataChange when a change occurs, possibly sending the data along with the notification.

Protocol	Similar API, Message
Clipboard	None
DDE	WM_DDE_ADVISE
OLE 1.0	None

- **::DUnadvise** terminates an advisory connection previously established with ::Advise.

Protocol	Similar API, Message
Clipboard	None
DDE	WM_DDE_UNADVISE
OLE 1.0	None

- **::EnumDAdvise** returns an enumerator with the IEnumSTATDATA interface. There is no equivalent to this function in any existing protocol. For more information, refer to the OLE 2.0 Programmer's Reference.

You can see from the list above that any single existing protocol does not support the full range of functionality that OLE 2.0 data objects provide for all the protocols (with the exception of DDE). This is not to say that any arbitrary data object you obtain from a clipboard transfer will actually implement every member function–some, like a static bitmap on the clipboard, will refuse any advisory connections. Others may not support any ::SetData calls. But you are always allowed to at least try and learn the data object's capabilities through error return values. With existing protocols, you are not even allowed to play a little.

The next few sections deal with how to both implement and use a general data object implemented as a component object. The samples shown in these sections contain more code than is relevant for this immediate discussion as they also serve the discussion of notification in "Advising and Notification with Data Objects" below.

Stand-Alone Data Objects

A stand-alone data object is one that you obtain by calling a function like CoCreateInstance (or anything that does IClassFactory::CreateInstance). Since we will not address OLE 2.0's treatment of the clipboard or drag-drop until the next chapters, we will use CoCreateInstance in the examples here to instantiate a data object. This method is most useful in situations where the data object does not represent some piece of clipboard-like data but rather represents a source for data collected from a specialized piece of hardware in the machine or possibly from a database. In other words, data objects can represent any data and don't imply that there has to be any user interaction. When loaded, such a data object might automatically connect to a database or initialize the piece of hardware and start data collection. The user of that object, when interested, can ask the data object for the actual information at any time. The DLL data object implemented in "Component Data Objects" below is a good framework for just such a data object.

FORMATETC Enumerators and Format Ordering

1The 'D' in the function names identify these functions as belonging to IDataObject. Prior to the release of OLE 2.0 both IDataObject and IOleObject had member functions Advise, Unadvise, and EnumAdvise which played havoc on objects that used multiple inheritance. To insure that the names would not conflict, IDataObject's members are DAdvise, DUnadvise, and EnumDAdvise whereas those in IOleObject remained Advise, Unadvise, and EnumAdvise.

The IDataObject::EnumFormatEtc is responsible for creating and returning an IEnumFORMATETC object to the caller where that enumerator knows either the data obtainable from IDataObject::GetData or the data that can be handled in IDataObject::SetData. These enumerators share the same member functions as all IEnum* interfaces: ::Next, ::Skip, ::Reset, and ::Clone. They just deal with FORMATETC structures instead of some other type.

As we saw in Chapter 3, an enumerator is simple to implement given an array of the structures it should enumerate. IEnumFORMATETC is, in fact, about the only enumerator that most applications will have occasion to implement since the exact sequence and contents of the FORMATETCs enumerated is application-specific. Even so, such an enumerator, given a pointer to an array of structures and a count of structures, can be reduced to a standard implementation as shown in Listing 6-1. The code shown here is contained in the INTERFAC directory as IENUMFE.H and IENUMFE.CPP which provide a full enumerator implementation that you can just drop into your own code and use without any modifications (except the #include at the top of IENUMFE.CPP).

IENUMFE.H

```
/*
 * IENUMFE.H
 *
 * Definitions of a template IDataObject interface implementation.
 *
 * Copyright (c)1993 Microsoft Corporation, All Right Reserved
 */
```

Listing 6-1: IENUMFE.H and IENUMFE.CPP default implementations of IEnumFORMATETC.

```
#ifndef _IENUMFE_H_
#define _IENUMFE_H_


/*
 * IEnumFORMATETC object that is created from IDataObject::EnumFormatEtc.
 * This object lives on its own, that is, QueryInterface only knowns
 * IUnknown and IEnumFormatEtc, nothing more.  We still use an outer
 * unknown for reference counting, because as long as this enumerator
 * lives, the data object should live, thereby keeping the application up.
 */

class __far CEnumFormatEtc : public IEnumFORMATETC
    {
    private:
        ULONG           m_cRef;         //Object reference count.
        LPUNKNOWN       m_punkRef;      //IUnknown for reference counting.
        ULONG           m_iCur;         //Current element.
        ULONG           m_cfe;          //Number of FORMATETCs in us
        LPFORMATETC     m_prgfe;        //Source of FORMATETCs
```

```
   public:
      CEnumFormatEtc(LPUNKNOWN, ULONG, LPFORMATETC);
      ~CEnumFormatEtc(void);

      //IUnknown members that delegate to m_punkOuter.
      STDMETHODIMP        QueryInterface(REFIID, LPVOID FAR *);
      STDMETHODIMP_(ULONG) AddRef(void);
      STDMETHODIMP_(ULONG) Release(void);

      //IEnumFORMATETC members
      STDMETHODIMP Next(ULONG, LPFORMATETC, ULONG FAR *);
      STDMETHODIMP Skip(ULONG);
      STDMETHODIMP Reset(void);
      STDMETHODIMP Clone(IEnumFORMATETC FAR * FAR *);
   };


typedef CEnumFormatEtc FAR * LPCEnumFormatEtc;


#endif //_IENUMFE_H_
```

IENUMFE.CPP

```
/*
 * IENUMFE.CPP
 *
 * Implementation of the IEnumFORMATETC interface for a data object.
 * This class can be generically used for any FORMATETC enumerations
 * as you pass the desired array to the contstructor.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */


#include "ienumfe.h"


CEnumFormatEtc::CEnumFormatEtc(LPUNKNOWN punkRef, ULONG cFE
   , LPFORMATETC prgFE)
   {
   UINT       i;

   m_cRef=0;
   m_punkRef=punkRef;
```

```
   m_iCur=0;
   m_cfe=cFE;
   m_prgfe=new FORMATETC[(UINT)cFE];

   if (NULL!=m_prgfe)
      {
      for (i=0; i < cFE; i++)
         m_prgfe[i]=prgFE[i];
      }

   return;
   }


CEnumFormatEtc::~CEnumFormatEtc(void)
   {
   if (NULL!=m_prgfe)
      delete [] m_prgfe;

   return;
   }



/*
 * CEnumFormatEtc::QueryInterface
 * CEnumFormatEtc::AddRef
 * CEnumFormatEtc::Release
 *
 * Purpose:
 *  IUnknown members for CEnumFormatEtc object.  For QueryInterface
 *  we only return out own interfaces and not those of the data object.
 *  However, since enumerating formats only makes sense when the data
 *  object is around, we insure that it stays as long as we stay by
 *  calling an outer IUnknown for ::AddRef and ::Release.  But since we
 *  are not controlled by the lifetime of the outer object, we still keep
 *  our own reference count in order to free ourselves.
 */

STDMETHODIMP CEnumFormatEtc::QueryInterface(REFIID riid, LPVOID FAR *ppv)
   {
   *ppv=NULL;

   /*
    * Enumerators do not live on the same level as the data object, so
```

```
    * we only need to support out IUnknown and IEnumFORMATETC interfaces
    * here with no concern for aggregation.
    */
   if (IsEqualIID(riid, IID_IUnknown) || IsEqualIID(riid, IID_IEnumFORMATETC))
      *ppv=(LPVOID)this;

   //AddRef any interface we'll return.
   if (NULL!=*ppv)
      {
      ((LPUNKNOWN)*ppv)->AddRef();
      return NOERROR;
      }

   return ResultFromScode(E_NOINTERFACE);
   }


STDMETHODIMP_(ULONG) CEnumFormatEtc::AddRef(void)
   {
   ++m_cRef;
   m_punkRef->AddRef();
   return m_cRef;
   }

STDMETHODIMP_(ULONG) CEnumFormatEtc::Release(void)
   {
   ULONG       cRefT;

   cRefT=--m_cRef;

   m_punkRef->Release();

   if (0==m_cRef)
      delete this;

   return cRefT;
   }



STDMETHODIMP CEnumFormatEtc::Next(ULONG cFE, LPFORMATETC pFE
   , ULONG FAR * pulFE)
   {
   ULONG             cReturn=0L;

   if (NULL==m_prgfe)
      return ResultFromScode(S_FALSE);
```

```
   if (NULL!=pulFE)
      *pulFE=0L;

   if (NULL==pFE || m_iCur >= m_cfe)
      return ResultFromScode(S_FALSE);

   while (m_iCur < m_cfe && cFE > 0)
      {
      *pFE++=m_prgfe[m_iCur++];
      cReturn++;
      cFE--;
      }

   if (NULL!=pulFE)
      *pulFE=(cReturn-cFE);

   return NOERROR;
   }



STDMETHODIMP CEnumFormatEtc::Skip(ULONG cSkip)
   {

   if (((m_iCur+cSkip) >= m_cfe) || NULL==m_prgfe)
      return ResultFromScode(S_FALSE);

   m_iCur+=cSkip;
   return NOERROR;
   }



STDMETHODIMP CEnumFormatEtc::Reset(void)
   {
   m_iCur=0;
   return NOERROR;
   }



STDMETHODIMP CEnumFormatEtc::Clone(LPENUMFORMATETC FAR *ppEnum)
   {
   LPCEnumFormatEtc    pNew;
```

```
*ppEnum=NULL;

//Create the clone
pNew=new CEnumFormatEtc(m_punkRef, m_cfe, m_prgfe);

if (NULL==pNew)
   return ResultFromScode(E_OUTOFMEMORY);

pNew->AddRef();
pNew->m_iCur=m_iCur;

*ppEnum=pNew;
return NOERROR;
}
```

Notice first that the class used to implement this enumerator is named CEnumFormatEtc–we're implementing a fairly independent object with one interface, so this is not something we classify as an interface implementation. In any case, these enumerator objects maintain their own reference count, an index of the current element, a count of elements, and an array of FORMATETCs that define the elements. The count and the array are provided by the data object through the CEnumFormatEtc constructor; the enumerator makes a snapshot copy of the FORMATETC array. Note that the only difference between a ::GetData enumerator and a ::SetData enumerator would be the actual FORMATETCs enumerated; the data object, when creating either enumerator, can pass the appropriate array and count to the CEnumFormatEtc constructor.

The enumerator's QueryInterface only knows IUnknown and IEnumFORMATETC: it should not know IDataObject or any other interface. This does not mean, however, that the enumerator is *entirely* independent, because it makes little sense to have an enumerator around when the data object that created it has already been destroyed (that is, the enumerator has lost its context). Therefore the enumerator takes a third parameter to its constructor, an LPUNKNOWN to use for reference counting. This IUnknown should be the controlling unknown that is used by the data object's IDataObject interface implementation itself, as we'll see in the next section. When the enumerator gets an ::AddRef or ::Release, it also calls ::AddRef or ::Release on this reference-counting IUnknown. That guarantees that the object that defines the context of the enumerator will remain at least as long as the enumerator itself remains.

In a sense, the enumerator attaches itself to the object implementing IDataObject but is not tightly coupled via QueryInterface. Since it's an independent object, the enumerator still maintains its own reference count for ::AddRef and ::Release and will free itself when its reference count is released to zero.

As mentioned in the previous section, IDataObject::EnumFormatEtc is the logical equivalent of the EnumClipboardFormats API in Windows. Therefore the order or formats enumerated through IEnumFORMATETC should be exactly the same as the order in which you would place data on the clipboard using SetClipboardData, typically from high-fidelity formats to low-fidelity formats. This usually results in a sequence starting with your private data formats, then other standard interchange formats, followed by 'picture' or 'graphic' formats like CF_METAFILEPICT and CF_BITMAP. As we'll see in later chapters, OLE 2.0 introduces a number of new clipboard formats for compound document use that you wedge between your existing formats as some OLE 2.0 formats are better descriptions of data than something like CF_BITMAP.

Component Data Objects

Implementing a data object is generally implementing some object with an IDataObject interface implementation. The object may, in general, have more interfaces as well, but the one of important here is IDataObject. The code shown in Listing 6-2 is an implementation of a data object in both DLL (CHAP06\DDATAOBJECT) and EXE (CHAP06\EDATAOBJ) using the same module housing as we implemented in Chapter 4's Koala objects. Note that the code shown below does not contain either the module housing or the IEnumFORMATETC implementation, especially as the latter is identical to the code shown in Listing 6-1.

DATAOBJ.CPP

```
/*
 * DATAOBJ.CPP
 * Data Object for Chapter 6
 *
 * Implementation of the CDataObject and CImpIDataObject that work
 * in either an EXE or DLL.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */


#include "dataobj.h"
```

Listing 6-2: A Data Object implementation. DATAOBJ.CPP contains the object and IDATAOBJ.CPP contains the interface implementation. RENDER.CPP isolates the code to create the data formats into private functions in the object.

```
extern HINSTANCE   g_hInst;

DWORD              g_dwID=0;

//Names of data sizes
static char  * rgszSize[3]={"Small", "Medium", "Large"};



CDataObject::CDataObject(LPUNKNOWN punkOuter, LPFNDESTROYED pfnDestroy
   , UINT iSize)
   {
   UINT       i;

   m_cRef=0;
   m_punkOuter=punkOuter;
   m_pfnDestroy=pfnDestroy;
   m_iSize=iSize;

   m_hWndAdvise=NULL;
   m_dwAdvFlags=ADVF_NODATA;

   //NULL any contained interfaces initially.
   m_pIDataObject=NULL;
   m_pIDataAdviseHolder=NULL;

   //Initilize the FORMATETCs arrays we use for ::EnumFormatEtc
```

```
   m_cfeGet=CFORMATETCGET;

   //These macros are in bookguid.h
   SETDefFormatEtc(m_rgfeGet[0], CF_METAFILEPICT, TYMED_MFPICT);
   SETDefFormatEtc(m_rgfeGet[1], CF_BITMAP, TYMED_GDI);
   SETDefFormatEtc(m_rgfeGet[2], CF_TEXT, TYMED_HGLOBAL);

   for (i=0; i < DOSIZE_CSIZES; i++)
      m_rghBmp[i]=NULL;

   return;
   }


CDataObject::~CDataObject(void)
   {
   UINT        i;

   for (i=0; i < DOSIZE_CSIZES; i++)
      {
      if (NULL!=m_rghBmp[i])
         DeleteObject(m_rghBmp[i]);
      }

   if (NULL!=m_pIDataAdviseHolder)
      m_pIDataAdviseHolder->Release();

   if (NULL!=m_pIDataObject)
      delete m_pIDataObject;

   if (NULL!=m_hWndAdvise)
      DestroyWindow(m_hWndAdvise);

   return;
   }


BOOL CDataObject::FInit(void)
   {
   LPUNKNOWN       pIUnknown=(LPUNKNOWN)this;
   UINT            i;
   char            szTemp[80];
   UINT            cy;

   if (NULL!=m_punkOuter)
      pIUnknown=m_punkOuter;
```

```
//Allocate contained interfaces.
m_pIDataObject=new CImpIDataObject(this, pIUnknown);

if (NULL==m_pIDataObject)
    return FALSE;

for (i=0; i < DOSIZE_CSIZES; i++)
    {
    m_rghBmp[i]=LoadBitmap(g_hInst, MAKEINTRESOURCE(i+IDB_MIN));

    if (NULL==m_rghBmp[i])
        return FALSE;
    }


/*
 * Register the Advise window class first time through (g_dwID==0)
 */

if (0==g_dwID)
    {
    WNDCLASS    wc;

    wc.style         = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc   = AdvisorWndProc;
    wc.cbClsExtra    = 0;
    wc.cbWndExtra    = sizeof(LPCDataObject);
    wc.hInstance     = g_hInst;
    wc.hIcon         = LoadIcon(g_hInst, MAKEINTRESOURCE(IDR_ADVISORICON));
    wc.hCursor       = NULL;
    wc.hbrBackground  = (HBRUSH)(COLOR_WINDOW + 1);
    wc.lpszMenuName   = MAKEINTRESOURCE(IDR_MENU);
    wc.lpszClassName  = "Advisor";

    if (!RegisterClass(&wc))
        return FALSE;
    }

/*
 * Create an advise window with a unique caption: "[EXE | DLL] Advisor #xx"
 * where xx is counted globally every time a CDataObject is created.
 */

g_dwID++;
#ifdef EXEDATAOBJECT
wsprintf(szTemp, "%s EXE Advisor #%lu", (LPSTR)rgszSize[m_iSize], g_dwID);
```

```
  #else
   wsprintf(szTemp, "%s DLL Advisor #%lu", (LPSTR)rgszSize[m_iSize], g_dwID);
  #endif

   cy=(GetSystemMetrics(SM_CYBORDER)*2)+GetSystemMetrics(SM_CYMENU)
      + GetSystemMetrics(SM_CYCAPTION);

   m_hWndAdvise=CreateWindow("Advisor", szTemp
      , WS_OVERLAPPED | WS_CAPTION | WS_MINIMIZEBOX | WS_BORDER |
WS_VISIBLE
      , CW_USEDEFAULT, CW_USEDEFAULT, 200, cy, HWND_DESKTOP
      , NULL, g_hInst, this);

   if (NULL==m_hWndAdvise)
      return FALSE;

   return TRUE;
   }



STDMETHODIMP CDataObject::QueryInterface(REFIID riid, LPVOID FAR *ppv)
   {
   *ppv=NULL;

   if (IsEqualIID(riid, IID_IUnknown))
      *ppv=(LPVOID)this;

   if (IsEqualIID(riid, IID_IDataObject))
      *ppv=(LPVOID)m_pIDataObject;

   //AddRef any interface we'll return.
   if (NULL!=*ppv)
      {
      ((LPUNKNOWN)*ppv)->AddRef();
      return NOERROR;
      }

   return ResultFromScode(E_NOINTERFACE);
   }


STDMETHODIMP_(ULONG) CDataObject::AddRef(void)
   {
   return ++m_cRef;
   }
```

```
STDMETHODIMP_(ULONG) CDataObject::Release(void)
   {
   ULONG       cRefT;

   cRefT=--m_cRef;

   if (0==m_cRef)
      {
      /*
       * Tell the housing that an object is going away so it can
       * shut down if appropriate.
       */
      if (NULL!=m_pfnDestroy)
         (*m_pfnDestroy)();

      delete this;
      }

   return cRefT;
   }



LRESULT FAR PASCAL __export AdvisorWndProc(HWND hWnd, UINT iMsg
   , WPARAM wParam, LPARAM lParam)
   {
   LPCDataObject   pDO;
   DWORD           i;
   DWORD           iAdvise;
   DWORD           dwTime;
   DWORD           dwAvg;
   char            szTime[128];
   char            szTitle[80];
   HCURSOR         hCur, hCurT;

   pDO=(LPCDataObject)GetWindowLong(hWnd, 0);

   switch (iMsg)
      {
      case WM_NCCREATE:
         pDO=(LPCDataObject)((LONG)((LPCREATESTRUCT)lParam)->lpCreateParams);
         SetWindowLong(hWnd, 0, (LONG)pDO);
         return (DefWindowProc(hWnd, iMsg, wParam, lParam));

      case WM_CLOSE:
```

```
        //Eat to forbid task manager from closing us.
        return 0L;

    case WM_COMMAND:
        if (NULL==pDO->m_pIDataAdviseHolder)
            break;

        //Send IAdviseSink::OnDataChange many times.
        i=(DWORD)(LOWORD(wParam)-IDM_ADVISEITERATIONSMIN+1);
        iAdvise=(i*i)*16;

        hCur=LoadCursor(NULL, MAKEINTRESOURCE(IDC_WAIT));
        hCurT=SetCursor(hCur);
        ShowCursor(TRUE);

        dwTime=GetTickCount();

        i=0;
        while (TRUE)
            {
            #ifdef EXEDATAOBJECT
            #ifndef WIN32
            MSG     msg;

            if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
                {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
                }
            else
            #endif
            #endif
                {
                pDO->m_pIDataAdviseHolder->SendOnDataChange(
                    pDO->m_pIDataObject, 0, ADVF_NODATA);

                if (++i >= iAdvise)
                    break;
                }
            }

        dwTime=GetTickCount()-dwTime;
        dwAvg=dwTime/iAdvise;

        SetCursor(hCurT);
        ShowCursor(FALSE);
```

```
            wsprintf(szTime, "Total\t=%lu milliseconds\n\rAverage\t=%lu milliseconds"
               , dwTime, dwAvg);

            GetWindowText(hWnd, szTitle, sizeof(szTitle));
            MessageBox(hWnd, szTime, szTitle, MB_OK);
            break;

         default:
            return (DefWindowProc(hWnd, iMsg, wParam, lParam));
         }

      return 0L;
      }
```

IDATAOBJ.CPP

```
/*
 * IDATAOBJ.CPP
 * Data Object for Chapter 6
 *
 * Implementation of the IDataObject interface for CDataObject.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */


#include "dataobj.h"


CImpIDataObject::CImpIDataObject(LPVOID pObj, LPUNKNOWN punkOuter)
   {
   m_cRef=0;
   m_pObj=pObj;
   m_punkOuter=punkOuter;
   return;
   }

CImpIDataObject::~CImpIDataObject(void)
   {
   return;
   }


STDMETHODIMP CImpIDataObject::QueryInterface(REFIID riid, LPVOID FAR *ppv)
```

```
    {
    return m_punkOuter->QueryInterface(riid, ppv);
    }


STDMETHODIMP_(ULONG) CImpIDataObject::AddRef(void)
    {
    ++m_cRef;
    return m_punkOuter->AddRef();
    }

STDMETHODIMP_(ULONG) CImpIDataObject::Release(void)
    {
    --m_cRef;
    return m_punkOuter->Release();
    }


STDMETHODIMP CImpIDataObject::GetData(LPFORMATETC pFE, LPSTGMEDIUM
pSTM)
    {
    LPCDataObject   pObj=(LPCDataObject)m_pObj;
    UINT            cf=pFE->cfFormat;

    //Check the aspects we support.
    if (!(DVASPECT_CONTENT & pFE->dwAspect))
        return ResultFromScode(DATA_E_FORMATETC);

    switch (cf)
        {
        case CF_METAFILEPICT:
            if (!(TYMED_MFPICT & pFE->tymed))
                break;

            return pObj->RenderMetafilePict(pSTM);

        case CF_BITMAP:
            if (!(TYMED_GDI & pFE->tymed))
                break;

            return pObj->RenderBitmap(pSTM);

        case CF_TEXT:
            if (!(TYMED_HGLOBAL & pFE->tymed))
                break;

            return pObj->RenderText(pSTM);
```

```
      default:
         break;
      }

   return ResultFromScode(DATA_E_FORMATETC);
   }



STDMETHODIMP CImpIDataObject::GetDataHere(LPFORMATETC pFE,
LPSTGMEDIUM pSTM)
   {
   //We don't implement this now.
   return ResultFromScode(E_NOTIMPL);
   }



STDMETHODIMP CImpIDataObject::QueryGetData(LPFORMATETC pFE)
   {
   LPCDataObject   pObj=(LPCDataObject)m_pObj;
   UINT            cf=pFE->cfFormat;
   BOOL            fRet=FALSE;

   //Check the aspects we support.
   if (!(DVASPECT_CONTENT & pFE->dwAspect))
      return ResultFromScode(DATA_E_FORMATETC);

   switch (cf)
      {
      case CF_METAFILEPICT:
         fRet=(BOOL)(pFE->tymed & TYMED_MFPICT);
         break;

      case CF_BITMAP:
         fRet=(BOOL)(pFE->tymed & TYMED_GDI);
         break;

      case CF_TEXT:
         fRet=(BOOL)(pFE->tymed & TYMED_HGLOBAL);
         break;

      default:
         fRet=FALSE;
         break;
```

```
      }

   return fRet ? NOERROR : ResultFromScode(S_FALSE);
   }



STDMETHODIMP CImpIDataObject::GetCanonicalFormatEtc(LPFORMATETC pFEIn
   , LPFORMATETC pFEOut)
   {
   return ResultFromScode(DATA_S_SAMEFORMATETC);
   }



STDMETHODIMP CImpIDataObject::SetData(LPFORMATETC pFE, STGMEDIUM FAR
*pSTM
   , BOOL fRelease)
   {
   //We don't handle SetDatas here.
   return ResultFromScode(DATA_E_FORMATETC);
   }



STDMETHODIMP CImpIDataObject::EnumFormatEtc(DWORD dwDir
   , LPENUMFORMATETC FAR *ppEnum)
   {
   LPCDataObject   pObj=(LPCDataObject)m_pObj;

   /*
    * We only support ::GetData in this object so we return NULL for
    * DATADIR_SET.  Otherwise we instantiate one of our CEnumFormatEtc
    * objects which has all the appropriate functions; we only need to
    * tell it where our arrays live.
    *
    * The m_punkOuter passed to the enumerator allows the enumerator
    * to AddRef and Release our controlling unknown as it receives
    * reference counting calls.  By calling AddRef itself, it can
    * insure that the object that holds the data context will stick
    * around as long as the enumerator itself is around.  Otherwise
    * it could be enumerating through bogus data.
    */

   switch (dwDir)
      {
      case DATADIR_GET:
```

```
            *ppEnum=(LPENUMFORMATETC)new CEnumFormatEtc(m_punkOuter
               , pObj->m_cfeGet, pObj->m_rgfeGet);
             break;

         case DATADIR_SET:
            *ppEnum=NULL;
            break;

         default:
            *ppEnum=NULL;
            break;
         }

      if (NULL==*ppEnum)
         return ResultFromScode(E_FAIL);
      else
         (*ppEnum)->AddRef();

      return NOERROR;
      }



STDMETHODIMP CImpIDataObject::DAdvise(LPFORMATETC pFE, DWORD dwFlags
      , LPADVISESINK pIAdviseSink, LPDWORD pdwConn)
      {
      LPCDataObject   pObj=(LPCDataObject)m_pObj;
      HRESULT         hr;

      if (NULL==pObj->m_pIDataAdviseHolder)
         {
         hr=CreateDataAdviseHolder(&pObj->m_pIDataAdviseHolder);

         if (FAILED(hr))
            return ResultFromScode(E_OUTOFMEMORY);
         }

      hr=pObj->m_pIDataAdviseHolder->Advise((LPDATAOBJECT)this, pFE
         , dwFlags, pIAdviseSink, pdwConn);

      return hr;
      }



STDMETHODIMP CImpIDataObject::DUnadvise(DWORD dwConn)
```

```
   {
   LPCDataObject   pObj=(LPCDataObject)m_pObj;
   HRESULT         hr;

   if (NULL==pObj->m_pIDataAdviseHolder)
      return ResultFromScode(E_FAIL);

   hr=pObj->m_pIDataAdviseHolder->Unadvise(dwConn);

   return hr;
   }



STDMETHODIMP CImpIDataObject::EnumDAdvise(LPENUMSTATDATA FAR
*ppEnum)
   {
   LPCDataObject   pObj=(LPCDataObject)m_pObj;
   HRESULT         hr;

   if (NULL==pObj->m_pIDataAdviseHolder)
      return ResultFromScode(E_FAIL);

   hr=pObj->m_pIDataAdviseHolder->EnumAdvise(ppEnum);
   return hr;
   }
```

RENDER.CPP

```
/*
 * RENDER.CPP
 * Data Object for Chapter 6
 *
 * CDataObject::Render* functions to create text, bitmaps, and metafiles
 * in a variety of sizes.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */



#include "dataobj.h"
#include <string.h>



HRESULT CDataObject::RenderText(LPSTGMEDIUM pSTM)
```

```
{
DWORD       cch;
HGLOBAL     hMem;
LPSTR       psz;
UINT        i;

//Get the size of data we're dealing with
switch (m_iSize)
   {
   case DOSIZE_SMALL:
      cch=CCHTEXTSMALL;
      break;

   case DOSIZE_MEDIUM:
      cch=CCHTEXTMEDIUM;
      break;

   case DOSIZE_LARGE:
      cch=CCHTEXTLARGE;
      break;

   default:
      return ResultFromScode(E_FAIL);
   }

hMem=GlobalAlloc(GMEM_SHARE | GMEM_MOVEABLE, cch);

if (NULL==hMem)
   return ResultFromScode(STG_E_MEDIUMFULL);

psz=(LPSTR)GlobalLock(hMem);

for (i=0; i < cch-1; i++)
   *(psz+i)=' ' + (i % 32);

*(psz+i)=0;

GlobalUnlock(hMem);

pSTM->hGlobal=hMem;
pSTM->tymed=TYMED_HGLOBAL;
return NOERROR;
}
```

```
HRESULT CDataObject::RenderBitmap(LPSTGMEDIUM pSTM)
   {
   HBITMAP     hBmp;
   UINT        cxy;
   HDC         hDC, hDCSrc, hDCDst;

   //Get the size of bitmap we're dealing with
   switch (m_iSize)
      {
      case DOSIZE_SMALL:
         cxy=CXYBITMAPSMALL;
         break;

      case DOSIZE_MEDIUM:
         cxy=CXYBITMAPMEDIUM;
         break;

      case DOSIZE_LARGE:
         cxy=CXYBITMAPLARGE;
         break;

      default:
         return ResultFromScode(E_FAIL);
      }

   //Get two memory DCs between which to BitBlt.
   hDC=GetDC(NULL);
   hDCSrc=CreateCompatibleDC(hDC);
   hDCDst=CreateCompatibleDC(hDC);
   ReleaseDC(NULL, hDC);

   if (NULL==hDCSrc || NULL==hDCDst)
      {
      if (NULL!=hDCDst)
         DeleteDC(hDCDst);

      if (NULL!=hDCSrc)
         DeleteDC(hDCSrc);

      return ResultFromScode(STG_E_MEDIUMFULL);
      }

   SelectObject(hDCSrc, m_rghBmp[m_iSize-DOSIZE_SMALL]);

   hBmp=CreateCompatibleBitmap(hDCSrc, cxy, cxy);

   if (NULL==hBmp)
```

```
    {
    DeleteDC(hDCDst);
    DeleteDC(hDCSrc);

    if (NULL!=hBmp)
        DeleteObject(hBmp);

    return ResultFromScode(STG_E_MEDIUMFULL);
    }

  //Copy from the source to destination
  SelectObject(hDCDst, hBmp);
  BitBlt(hDCDst, 0, 0, cxy, cxy, hDCSrc, 0, 0, SRCCOPY);

  DeleteDC(hDCDst);
  DeleteDC(hDCSrc);

  pSTM->hGlobal=(HGLOBAL)hBmp;
  pSTM->tymed=TYMED_GDI;
  return NOERROR;
  }



HRESULT CDataObject::RenderMetafilePict(LPSTGMEDIUM pSTM)
  {
  HDC           hDC;
  HGLOBAL       hMem;
  HMETAFILE     hMF;
  LPMETAFILEPICT  pMF;
  HBRUSH        hBrush;
  HGDIOBJ       hBrT;
  UINT          cRec;
  int           x, y, dxy;
  RECT          rc;

  switch (m_iSize)
    {
    case DOSIZE_SMALL:
      cRec=CRECMETAFILESMALL;
      break;

    case DOSIZE_MEDIUM:
      cRec=CRECMETAFILEMEDIUM;
      break;
```

```
        case DOSIZE_LARGE:
            cRec=CRECMETAFILELARGE;
            break;

        default:
            return ResultFromScode(E_FAIL);
        }


    hDC=(HDC)CreateMetaFile(NULL);

    if (NULL!=hDC)
        {
        /*
         * Draw something into the metafile.  For this object we
         * draw some number of rectangles equal to the number of
         * records we want.  So take the square root of the number
         * of records and iterate over that number in both x & y.
         */

        dxy=(int)1024/cRec;

        //This creates a blue shading from light (top) to dark (bottom)
        for (y=1024; y >=0; y-=dxy)
            {
            hBrush=CreateSolidBrush(RGB(0, 0, (1024-y)/4));
            hBrT=SelectObject(hDC, (HGDIOBJ)hBrush);

            for (x=0; x < 1024; x+=dxy)
                {
                SetRect(&rc, x, y, x+dxy, y+dxy);
                FillRect(hDC, &rc, hBrush);
                }

            SelectObject(hDC, hBrT);
            DeleteObject(hBrush);
            }

        hMF=CloseMetaFile(hDC);
        }

    if (NULL==hMF)
        return ResultFromScode(STG_E_MEDIUMFULL);

    //Allocate the METAFILEPICT structure.
    hMem=GlobalAlloc(GMEM_SHARE | GMEM_MOVEABLE,
sizeof(METAFILEPICT));
```

```
  if (NULL==hMem)
     {
     DeleteMetaFile(hMF);
     return ResultFromScode(STG_E_MEDIUMFULL);
     }

  pMF=(LPMETAFILEPICT)GlobalLock(hMem);

  pMF->hMF=hMF;
  pMF->mm=MM_ANISOTROPIC;
  pMF->xExt=1024;
  pMF->yExt=1024;

  GlobalUnlock(hMem);

  pSTM->hGlobal=hMem;
  pSTM->tymed=TYMED_MFPICT;
  return NOERROR;
  }
```

DDATAOBJ.RC (or EDATAOBJ.RC)

```
[Other entries omitted from listing]

IDB_16BY16      BITMAP      16.bmp
IDB_64BY64      BITMAP      64.bmp
IDB_256BY256    BITMAP      256.bmp


IDR_MENU MENU MOVEABLE DISCARDABLE
  BEGIN
   POPUP "&Iterations"
    BEGIN
     MENUITEM "&A 16",              IDM_ADVISEITERATIONS16
     MENUITEM "&B 64",              IDM_ADVISEITERATIONS64
     MENUITEM "&C 144",             IDM_ADVISEITERATIONS144
     MENUITEM "&D 256",             IDM_ADVISEITERATIONS256
     MENUITEM "&E 400",             IDM_ADVISEITERATIONS400
     MENUITEM "&F 572",             IDM_ADVISEITERATIONS572
    END
  #endif
  END
```

16.BMP, 64.BMP, 256.BMP

Each module actually supports three different CLSIDs where each CLSID represents a different data set. Each class has its own CLSID but use exactly the same entries for InProcServer and LocalServer in the registration database. Each object provides text, a bitmap, and a metafile in different sizes as shown in the table below:

CLSID	Text (chars)	Bitmap (x*y)	Metafile (FillRect records)
CLSID_DataSmall	64	16*16	16
CLSID_DataMedium	1024	64*64	128
CLSID_DataLarge	16384	256*256	1024

The text contains a repeating sequence of characters (starting at ASCII 32), the bitmap is just a sampling of a few bitmaps laying around my hard drive, and the metafile is composed of bands of different blue shaded rectangles. The large metafile produces an effect similar to that generated by the Windows SDK setup tools or the title screen of WinHelp.

DDATAOBJ and EDATAOBJ are good examples of implementing multiple classes a single module. In DDATAOBJ, the DllGetClassObject function uses the same class factory implementation for all three classes, only differentiating them with an extra parameter to the class factory constructor:

```
HRESULT __export FAR PASCAL DllGetClassObject(REFCLSID rclsid, REFIID riid
   , LPVOID FAR *ppv)
   {
   if (!IsEqualIID(riid, IID_IUnknown)
      && !IsEqualIID(riid, IID_IClassFactory))
      return ResultFromScode(E_NOINTERFACE);

   *ppv=NULL;

   if (IsEqualCLSID(rclsid, CLSID_DataObjectSmall))
      *ppv=(LPVOID)new CDataObjectClassFactory(DOSIZE_SMALL);

   if (IsEqualCLSID(rclsid, CLSID_DataObjectMedium))
      *ppv=(LPVOID)new CDataObjectClassFactory(DOSIZE_MEDIUM);

   if (IsEqualCLSID(rclsid, CLSID_DataObjectLarge))
      *ppv=(LPVOID)new CDataObjectClassFactory(DOSIZE_LARGE);

   if (NULL==*ppv)
      return ResultFromScode(E_OUTOFMEMORY);

   ((LPUNKNOWN)*ppv)->AddRef();
   return NOERROR;
   }
```

The DOSIZE_* values are defined for these samples as 0, 1, and 2 and are used to identify the data set in the data object. The EDATAOBJ program creates three class factories on startup, one for each data size, and registers each separately maintaining a separate registration key for each.

```
for (i=0; i < DOSIZE_CSIZES; i++)
   {
   m_rgpIClassFactory[i]=(LPCLASSFACTORY)new CDataObjectClassFactory(i);

   if (NULL==m_rgpIClassFactory[i])
      return FALSE;

   m_rgpIClassFactory[i]->AddRef();
   }

hr=CoRegisterClassObject(CLSID_DataObjectSmall
   , (LPUNKNOWN)m_rgpIClassFactory, CLSCTX_LOCAL_SERVER
```

```
    , REGCLS_MULTIPLEUSE, &m_rgdwRegCO[0]);

  hr2=CoRegisterClassObject(CLSID_DataObjectMedium
    , (LPUNKNOWN)m_rgpIClassFactory, CLSCTX_LOCAL_SERVER
    , REGCLS_MULTIPLEUSE, &m_rgdwRegCO[1]);

  hr3=CoRegisterClassObject(CLSID_DataObjectLarge
    , (LPUNKNOWN)m_rgpIClassFactory, CLSCTX_LOCAL_SERVER
    , REGCLS_MULTIPLEUSE, &m_rgdwRegCO[2]);

  if (FAILED(hr) || FAILED(hr2) || FAILED(hr3))
    return FALSE;
```

In both DLL and EXE implementations the class factories identically create the data object using a class CDataObject, passing the DOSIZE_* value to identify the data set. Other than these changes, the DLL and EXE housings are the same as those used in Chapter 4.

Some CDataObject Features

The data object has a number of interesting features worth mentioning briefly. First, each data object carries an array of FORMATETCs that describe the data obtainable from ::GetData only, since ::SetData is not handled by these objects. Filling out such an array can be tedious, so here we make use of our SETFormatEtc and SETDefFormatEtc macros in INC\BOOKGUID.H. Note the ordering here starts with CF_METAFILEPICT and drops to CF_TEXT as for this example we consider the metafile the richest, most descriptive format:

```
//Initialize the FORMATETCs arrays we use for ::EnumFormatEtc
m_cfeGet=CFORMATETCGET;

SETDefFormatEtc(m_rgfeGet[0], CF_METAFILEPICT, TYMED_MFPICT);
SETDefFormatEtc(m_rgfeGet[1], CF_BITMAP, TYMED_GDI);
SETDefFormatEtc(m_rgfeGet[2], CF_TEXT, TYMED_HGLOBAL);
```

When ::EnumFormatEtc is called, CDataObject passes the count of FORMATETCs, m_cfeGet, and the pointer to the FORMATETC array, m_rgfeGet, to the CEnumFormatEtc constructor. The enumerator copies the array and maintains it internally until the enumerator object itself is destroyed.

The second most interesting part of our data object implementation it that the object creates its own window, regardless of whether it lives in a DLL or EXE. These are small overlapped windows with a thin border and no system menu–only a caption, minimize box, and a menu. The menu on these windows are used for demonstrating notifications as discussed in "Advising and Notification with Data Objects" below, so we won't belabor them here. These windows do not have a system menu as they should always be present when the data object itself is present. For that reason the window procedure for the window class eats WM_CLOSE such that the windows cannot be removed from the Task Manager or with a malevolent Alt+F4.

Implementing IDataObject

The IDataObject implementation for our data object is contained in the file IDATAOBJ.CPP as shown in Listing 6-2. You will also find a skeletal IDataObject implementation and class definition in INTERFACE\IDATAOBJ.*. Most of the implementation is quite straightforward, and in fact some member functions don't need implementations at all. Let's therefore just point out the interesting parts, leaving specific details about less used members to the OLE 2.0 Programmer's Reference.

IDataObject::GetData, as well as ::GetDataHere and ::SetData, must not only check the clipboard format desired by the caller but all other fields in the FORMATETC. In particular, you must insure that you render the requested aspect, if possible, and comply with the requested storage medium if possible. If you do not render specifically for any device, then you can ignore the ptd field altogether. You only need to check that field if you make any special cases for particular devices. If you cannot comply with the FORMATETC, then return an HRESULT containing DATA_E_FORMATETC. If, when you try to render the data and allocations fail, return STG_E_MEDIUMFULL. You can see this in the code in RENDER.CPP.

It's very important that your implementation of ::GetData is aware that it must fill in the entire STGMEDIUM, that is, you must fill in the *tymed* field as well as providing the actual reference to the data. Failure to do so will generally cause marshaling of the data to fail, since OLE 2.0 would otherwise not know what to marshal.

::GetDataHere does not need to reallocate any storage medium in order to fulfill the request of the caller, that is, you don't have to try to reallocate bitmaps or metafile–you can just flat out fail. In fact, implementations of IDataObject::GetDataHere inside OLE 2.0 flat out fail if you pass a medium other than

TYMED_ISTORAGE, TYMED_ISTREAM, or TYMED_FILE. These three are generally why ::GetDataHere exists in the first place, so they are all you really need to support if you bother supporting any at all. If writing to a disk-based storage fails, then return STG_E_MEDIUMFULL. Note also that our implementation of ::GetDataHere returns E_NOTIMPL instead of DATA_E_FORMATETC to let callers know we just don't do this as opposed to telling them they sent the wrong FORMATETC. Telling callers that *they* have a problem when we don't implement a function is likely to drive them stark-raving mad.

::QueryGetData is very similar in structure to ::GetData since we execute most of the same checks on the FORMATETC passed to us; just instead of rendering the data we return NOERROR if we have the format, S_FALSE if we don't. All other checks on the aspect, target device, and storage medium are the same.

::GetCanonicalFormatEtc is easy to implement if you never do any device-specific renderings: just return an HRESULT with DATA_S_SAMEFORMATETC. Note that this is not a failure SCODE as it contains information (thus the _S_). Be sure not to compare the return value of ::GetCanonicalFormatEtc to NOERROR to test success, use the SUCCEEDED or FAILED macro instead. This function should ignore the *tymed* fields in both FORMATETCs as they have no relevance here.

::EnumFormatEtc simply creates one of our CEnumFormatEtc objects, passing to it the count and array of FORMATETCs that our ::GetData function supports. ::EnumFormatEtc takes a parameter described as the 'data direction' which is either Get or Set. If the caller asks for a Set enumerator, we return E_FAIL since we don't allow ::SetData at all.

Finally, you might notice a number of API calls in the ::DAdvise, ::DUnadvise, and ::EnumDAdvise implementations–these functions make use of an object called the "Data Advise Holder" for reasons that will become thankfully obvious in "Advising and Notification with DataObjects" below.

A [Component] Data Object User

Through one of a number of protocols, a piece of code that we can classify as a data consumer received a IDataObject pointer to a data object. The interface members available through that pointer represent all data manipulation APIs for the consumer such as retrieving data or checking that formats are available. The DATAUSER program shown in Listing 6-3 obtains three data objects using CoCreateInstance: one for the small data set (CLSID_DataObjectSmall), one for the medium set (CLSID_DataObjectMedium), and one for the large data set (CLSID_DataObjectLarge). Initially it instantiates the three data objects from DDATAOBJ.DLL by specifying CLSCTX_INPROC_SERVER in CoCreateInstance.

DATAUSER.CPP

```
/*
 * DATAUSER.CPP
 *
 * A user of the Data Objects for Chapter 6
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */


#define INITGUIDS
#include "datauser.h"


//These are for displaying clipboard formats textually.
static char * rgszCF[13]={"Unknown", "CF_TEXT", "CF_BITMAP",
"CF_METAFILEPICT"
            , "CF_SYLK", "CF_DIF", "CF_TIFF", "CF_OEMTEXT", "CF_DIB"
            , "CF_PALETTE", "CF_PENDATA", "CF_RIFF", "CF_WAVE"};


static char szSuccess[]    ="succeeded";
static char szFailed[]     ="failed";
static char szExpected[]   ="expected";
static char szUnexpected[] ="unexpected!";


int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hInstPrev
   , LPSTR pszCmdLine, int nCmdShow)
   {
   MSG        msg;
   LPAPPVARS   pAV;

   SetMessageQueue(96);

   //Create and initialize the application.
   pAV=new CAppVars(hInst, hInstPrev, nCmdShow);
```

```
   if (NULL==pAV)
      return -1;

   if (pAV->FInit())
      {
      while (GetMessage(&msg, NULL, 0,0 ))
         {
         TranslateMessage(&msg);
         DispatchMessage(&msg);
         }
      }

   delete pAV;
   return msg.wParam;
   }


LRESULT FAR PASCAL __export DataUserWndProc(HWND hWnd, UINT iMsg
   , WPARAM wParam, LPARAM lParam)
   {
   HRESULT        hr;
```

Listing 6-3: The DATAUSER program that uses the Component Data Objects implemented in the previous section.

```
   LPAPPVARS      pAV;
   HMENU          hMenu;
   FORMATETC      fe;
   WORD           wID;

   pAV=(LPAPPVARS)GetWindowLong(hWnd, DATAUSERWL_STRUCTURE);

   switch (iMsg)
      {
      case WM_NCCREATE:
         pAV=(LPAPPVARS)((LONG)((LPCREATESTRUCT)lParam)->lpCreateParams);
         SetWindowLong(hWnd, DATAUSERWL_STRUCTURE, (LONG)pAV);
         return (DefWindowProc(hWnd, iMsg, wParam, lParam));

      case WM_DESTROY:
         PostQuitMessage(0);
         break;

      case WM_PAINT:
         pAV->Paint();
         break;
```

```
      case WM_COMMAND:
         SETDefFormatEtc(fe, 0, TYMED_HGLOBAL | TYMED_GDI |
TYMED_MFPICT);

         hMenu=GetMenu(hWnd);
         wID=LOWORD(wParam);

         switch (wID)
            {
            case IDM_OBJECTUSEDLL:
               if (!pAV->m_fEXE)
                  break;

               pAV->m_fEXE=FALSE;
               pAV->FReloadDataObjects(TRUE);
               break;

            case IDM_OBJECTUSEEXE:
               if (pAV->m_fEXE)
                  break;

               pAV->m_fEXE=TRUE;
               pAV->FReloadDataObjects(TRUE);
               break;

            case IDM_OBJECTDATASIZESMALL:
            case IDM_OBJECTDATASIZEMEDIUM:
            case IDM_OBJECTDATASIZELARGE:
               CheckMenuItem(hMenu, IDM_OBJECTDATASIZESMALL,
MF_UNCHECKED);
               CheckMenuItem(hMenu, IDM_OBJECTDATASIZEMEDIUM,
MF_UNCHECKED);
               CheckMenuItem(hMenu, IDM_OBJECTDATASIZELARGE,
MF_UNCHECKED);
               CheckMenuItem(hMenu, wID,  MF_CHECKED);

               //Kill old advise.
               if (NULL!=pAV->m_pIDataObject || 0!=pAV->m_dwConn)
                  pAV->m_pIDataObject->DUnadvise(pAV->m_dwConn);

               if (IDM_OBJECTDATASIZELARGE==wID)
                  pAV->m_pIDataObject=pAV->m_pIDataLarge;
               else if (IDM_OBJECTDATASIZEMEDIUM==wID)
                  pAV->m_pIDataObject=pAV->m_pIDataMedium;
               else
                  pAV->m_pIDataObject=pAV->m_pIDataSmall;
```

```
        //Setup new advise.
        fe.cfFormat=pAV->m_cfAdvise;
        pAV->m_pIDataObject->DAdvise(&fe, ADVF_NODATA
            , pAV->m_pIAdviseSink, &pAV->m_dwConn);

        break;

    case IDM_OBJECTQUERYGETDATA:
        if (NULL==pAV->m_pIDataObject)
            break;

        fe.tymed=TYMED_HGLOBAL | TYMED_GDI | TYMED_MFPICT;

        pAV->TryQueryGetData(&fe, CF_TEXT,         TRUE,  0);
        pAV->TryQueryGetData(&fe, CF_BITMAP,       TRUE,  1);
        pAV->TryQueryGetData(&fe, CF_DIB,          FALSE, 2);
        pAV->TryQueryGetData(&fe, CF_METAFILEPICT, TRUE,  3);
        pAV->TryQueryGetData(&fe, CF_WAVE,         FALSE, 4);
        break;


    case IDM_OBJECTGETDATATEXT:
    case IDM_OBJECTGETDATABITMAP:
    case IDM_OBJECTGETDATAMETAFILEPICT:
        if (NULL==pAV->m_pIDataObject)
            break;

        //Clean up whatever we currently have.
        pAV->m_cf=0;
        ReleaseStgMedium(&pAV->m_stm);

        if (IDM_OBJECTGETDATATEXT==wID)
            SETDefFormatEtc(fe, CF_TEXT, TYMED_HGLOBAL);

        if (IDM_OBJECTGETDATABITMAP==wID)
            SETDefFormatEtc(fe, CF_BITMAP, TYMED_GDI);

        if (IDM_OBJECTGETDATAMETAFILEPICT==wID)
            SETDefFormatEtc(fe, CF_METAFILEPICT, TYMED_MFPICT);

        pAV->m_stm.tymed=fe.tymed;
        hr=pAV->m_pIDataObject->GetData(&fe, &(pAV->m_stm));

        if (SUCCEEDED(hr))
            pAV->m_cf=fe.cfFormat;
```

```
            InvalidateRect(hWnd, NULL, TRUE);
            UpdateWindow(hWnd);
            break;

         case IDM_OBJECTEXIT:
            PostMessage(hWnd, WM_CLOSE, 0, 0L);
            break;


         case IDM_ADVISETEXT:
         case IDM_ADVISEBITMAP:
         case IDM_ADVISEMETAFILEPICT:
            if (NULL==pAV->m_pIDataObject)
               break;

            //Terminate the old connection
            if (0!=pAV->m_dwConn)
               pAV->m_pIDataObject->DUnadvise(pAV->m_dwConn);

            CheckMenuItem(hMenu, pAV->m_cfAdvise+IDM_ADVISEMIN,
MF_UNCHECKED);
            CheckMenuItem(hMenu, wID, MF_CHECKED);

            //New format is wID-IDM_ADVISEMIN
            pAV->m_cfAdvise=(UINT)(wID-IDM_ADVISEMIN);
            fe.cfFormat=pAV->m_cfAdvise;
            pAV->m_pIDataObject->DAdvise(&fe, ADVF_NODATA
               , pAV->m_pIAdviseSink, &pAV->m_dwConn);

            break;

         case IDM_ADVISEGETDATA:
            pAV->m_fGetData=!pAV->m_fGetData;
            CheckMenuItem(hMenu, wID, pAV->m_fGetData ? MF_CHECKED :
MF_UNCHECKED);
            break;

         case IDM_ADVISEREPAINT:
            pAV->m_fRepaint=!pAV->m_fRepaint;
            CheckMenuItem(hMenu, wID, pAV->m_fRepaint ? MF_CHECKED :
MF_UNCHECKED);
            break;

         default:
            break;
         }
```

```
        break;

      default:
        return (DefWindowProc(hWnd, iMsg, wParam, lParam));
      }

   return 0L;
   }


CAppVars::CAppVars(HINSTANCE hInst, HINSTANCE hInstPrev, UINT nCmdShow)
   {
   m_hInst       =hInst;
   m_hInstPrev   =hInstPrev;
   m_nCmdShow    =nCmdShow;

   m_hWnd        =NULL;
   m_fEXE        =FALSE;

   m_pIAdviseSink =NULL;
   m_dwConn       =0;
   m_cfAdvise     =0;
   m_fGetData     =FALSE;
   m_fRepaint     =FALSE;

   m_pIDataSmall =NULL;
   m_pIDataMedium=NULL;
   m_pIDataLarge =NULL;
   m_pIDataObject=NULL;

   m_cf=0;
   m_stm.tymed=TYMED_NULL;
   m_stm.lpszFileName=NULL;      //Initializes union contents to NULL.
   m_stm.pUnkForRelease=NULL;

   m_fInitialized=FALSE;
   return;
   }


CAppVars::~CAppVars(void)
   {
   //This releases the data object interfaces and advises
   FReloadDataObjects(FALSE);

   ReleaseStgMedium(&m_stm);
```

```
   if (NULL!=m_pIAdviseSink)
      m_pIAdviseSink->Release();

   if (IsWindow(m_hWnd))
      DestroyWindow(m_hWnd);

   if (m_fInitialized)
      CoUninitialize();

   return;
   }


BOOL CAppVars::FInit(void)
   {
   WNDCLASS    wc;
   DWORD       dwVer;
   BOOL        fRet;

   dwVer=CoBuildVersion();

   if (rmm!=HIWORD(dwVer))
      return FALSE;

   if (FAILED(CoInitialize(NULL)))
      return FALSE;

   m_fInitialized=TRUE;

   if (!m_hInstPrev)
      {
      wc.style         = CS_HREDRAW | CS_VREDRAW;
      wc.lpfnWndProc   = DataUserWndProc;
      wc.cbClsExtra    = 0;
      wc.cbWndExtra    = CBWNDEXTRA;
      wc.hInstance     = m_hInst;
      wc.hIcon         = LoadIcon(m_hInst, "Icon");
      wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
      wc.hbrBackground  = (HBRUSH)(COLOR_WINDOW + 1);
      wc.lpszMenuName   = MAKEINTRESOURCE(IDR_MENU);
      wc.lpszClassName  = "DATAUSER";

      if (!RegisterClass(&wc))
         return FALSE;
      }
```

```
   m_hWnd=CreateWindow("DATAUSER", "Data Object User"
      , WS_OVERLAPPEDWINDOW,35, 35, 350, 250, NULL, NULL, m_hInst, this);

   if (NULL==m_hWnd)
      return FALSE;

   ShowWindow(m_hWnd, m_nCmdShow);
   UpdateWindow(m_hWnd);

   m_pIAdviseSink=new CImpIAdviseSink(this);

   if (NULL==m_pIAdviseSink)
      return FALSE;

   m_pIAdviseSink->AddRef();

   CheckMenuItem(GetMenu(m_hWnd), IDM_OBJECTUSEDLL,        MF_CHECKED);
   CheckMenuItem(GetMenu(m_hWnd), IDM_OBJECTDATASIZESMALL,
MF_CHECKED);

   //Load the initial objects
   fRet=FReloadDataObjects(TRUE);
   m_pIDataObject=m_pIDataSmall;

   return fRet;
   }



BOOL CAppVars::FReloadDataObjects(BOOL fReload)
   {
   HRESULT     hr1, hr2, hr3;
   DWORD       dwClsCtx;
   HCURSOR     hCur, hCurT;
   HMENU       hMenu;
   UINT        uTempD, uTempE;

   //Clean out any data we're holding
   m_cf=0;
   ReleaseStgMedium(&m_stm);

   //Turn off whatever data connection we have
   if (NULL!=m_pIDataObject && 0!=m_dwConn)
      m_pIDataObject->DUnadvise(m_dwConn);

   if (NULL!=m_pIDataLarge)
      m_pIDataLarge->Release();
```

```
if (NULL!=m_pIDataMedium)
   m_pIDataMedium->Release();

if (NULL!=m_pIDataSmall)
   m_pIDataSmall->Release();

m_pIDataObject=NULL;
CoFreeUnusedLibraries();

//Exit if we just wanted to free.
if (!fReload)
   return FALSE;


hCur=LoadCursor(NULL, MAKEINTRESOURCE(IDC_WAIT));
hCurT=SetCursor(hCur);
ShowCursor(TRUE);

dwClsCtx=(m_fEXE) ? CLSCTX_LOCAL_SERVER : CLSCTX_INPROC_SERVER;

hr1=CoCreateInstance(CLSID_DataObjectSmall, NULL, dwClsCtx
   , IID_IDataObject, (LPVOID FAR *)&m_pIDataSmall);

hr2=CoCreateInstance(CLSID_DataObjectMedium, NULL, dwClsCtx
   , IID_IDataObject, (LPVOID FAR *)&m_pIDataMedium);

hr3=CoCreateInstance(CLSID_DataObjectLarge, NULL, dwClsCtx
   , IID_IDataObject, (LPVOID FAR *)&m_pIDataLarge);

ShowCursor(FALSE);
SetCursor(hCurT);

//If anything fails, recurse to clean up...
if (FAILED(hr1) || FAILED(hr2) || FAILED(hr3))
   return FReloadDataObjects(FALSE);

//Reset the state of the menus for Small, no advise, no options.
hMenu=GetMenu(m_hWnd);
CheckMenuItem(hMenu, IDM_OBJECTDATASIZESMALL,  MF_CHECKED);
CheckMenuItem(hMenu, IDM_OBJECTDATASIZEMEDIUM, MF_UNCHECKED);
CheckMenuItem(hMenu, IDM_OBJECTDATASIZELARGE,  MF_UNCHECKED);

m_pIDataObject=m_pIDataSmall;
CheckMenuItem(hMenu, m_cfAdvise+IDM_ADVISEMIN, MF_UNCHECKED);
```

```
   uTempE=m_fEXE  ? MF_CHECKED : MF_UNCHECKED;
   uTempD=!m_fEXE ? MF_CHECKED : MF_UNCHECKED;

   CheckMenuItem(hMenu, IDM_OBJECTUSEDLL, uTempD);
   CheckMenuItem(hMenu, IDM_OBJECTUSEEXE, uTempE);

   CheckMenuItem(hMenu, IDM_ADVISEGETDATA, MF_UNCHECKED);
   CheckMenuItem(hMenu, IDM_ADVISEREPAINT, MF_UNCHECKED);

   m_fGetData=FALSE;
   m_fRepaint=FALSE;

   //Cannot request data using async advises, so disable these.
   uTempE=m_fEXE  ? MF_DISABLED | MF_GRAYED : MF_ENABLED;
   EnableMenuItem(hMenu,  IDM_ADVISEGETDATA, uTempE);
   EnableMenuItem(hMenu, IDM_ADVISEREPAINT, uTempE);

   return TRUE;
   }



void CAppVars::TryQueryGetData(LPFORMATETC pFE, UINT cf, BOOL fExpect, UINT
y)
   {
   char       szTemp[80];
   LPSTR      psz1;
   LPSTR      psz2;
   UINT       cch;
   HRESULT    hr;
   HDC        hDC;

   if (0!=cf)
      pFE->cfFormat=cf;

   hr=m_pIDataObject->QueryGetData(pFE);
   psz1=(NOERROR==hr) ? szSuccess : szFailed;
   psz2=((NOERROR==hr)==fExpect) ? szExpected : szUnexpected;

   hDC=GetDC(m_hWnd);
   SetTextColor(hDC, GetSysColor(COLOR_WINDOWTEXT));
   SetBkColor(hDC, GetSysColor(COLOR_WINDOW));

   if (CF_WAVE < cf || 0==cf)
      cch=wsprintf(szTemp, "QueryGetData on %d %s (%s).", cf, psz1, psz2);
   else
      {
```

```
      cch=wsprintf(szTemp, "QueryGetData on %s %s (%s)."
         , (LPSTR)rgszCF[cf], psz1, psz2);
      }

   //Don't overwrite other painted display.
   SetBkMode(hDC, TRANSPARENT);
   TextOut(hDC, 0, 16*y, szTemp, cch);

   ReleaseDC(m_hWnd, hDC);

   return;
   }


void CAppVars::Paint(void)
   {
   PAINTSTRUCT     ps;
   HDC             hDC;
   HDC             hMemDC;
   LPMETAFILEPICT  pMF;
   LPSTR           psz;
   RECT            rc;
   FORMATETC       fe;

   GetClientRect(m_hWnd, &rc);

   hDC=BeginPaint(m_hWnd, &ps);

   //May need to retrieve the data with EXE objects
   if (m_fEXE)
      {
      if (TYMED_NULL==m_stm.tymed && 0!=m_cf)
         {
         SETDefFormatEtc(fe, m_cf, TYMED_HGLOBAL | TYMED_MFPICT |
TYMED_GDI);

         if (NULL!=m_pIDataObject)
            m_pIDataObject->GetData(&fe, &m_stm);
         }
      }

   switch (m_cf)
      {
      case CF_TEXT:
         psz=(LPSTR)GlobalLock(m_stm.hGlobal);
```

```
        if (NULL==psz)
            break;

        SetTextColor(hDC, GetSysColor(COLOR_WINDOWTEXT));
        SetBkColor(hDC, GetSysColor(COLOR_WINDOW));

        DrawText(hDC, psz, lstrlen(psz), &rc, DT_LEFT | DT_WORDBREAK);
        GlobalUnlock(m_stm.hGlobal);
        break;


    case CF_BITMAP:
        hMemDC=CreateCompatibleDC(hDC);

        if (NULL!=SelectObject(hMemDC, (HGDIOBJ)m_stm.hGlobal))
            {
            BitBlt(hDC, 0, 0, rc.right-rc.left, rc.bottom-rc.top
                , hMemDC, 0, 0, SRCCOPY);
            }

        DeleteDC(hMemDC);
        break;


    case CF_METAFILEPICT:
        pMF=(LPMETAFILEPICT)GlobalLock(m_stm.hGlobal);

        if (NULL==pMF)
            break;

        SetMapMode(hDC, pMF->mm);
        SetWindowOrgEx(hDC, 0, 0, NULL);
        SetWindowExtEx(hDC, pMF->xExt, pMF->yExt, NULL);

        SetViewportExtEx(hDC, rc.right-rc.left, rc.bottom-rc.top, NULL);

        PlayMetaFile(hDC, pMF->hMF);
        GlobalUnlock(m_stm.hGlobal);
        break;


    default:
        break;
    }

EndPaint(m_hWnd, &ps);
return;
```

```
   }
```

DATAUSER.H

```
/*
 * DATAUSER.H
 *
 * Definitions for an user of the Data objects for Chapter 6
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */

#ifndef _DATAUSER_H_
#define _DATAUSER_H_

#include <windows.h>
#include <ole2.h>
#include <ole2ver.h>
#include <bookguid.h>

//Menu Resource ID and Commands
#define IDR_MENU                1

#define IDM_OBJECTUSEDLL            100
#define IDM_OBJECTUSEEXE            101
#define IDM_OBJECTDATASIZESMALL     102
#define IDM_OBJECTDATASIZEMEDIUM    103
#define IDM_OBJECTDATASIZELARGE     104
#define IDM_OBJECTQUERYGETDATA      105
#define IDM_OBJECTGETDATATEXT       106
#define IDM_OBJECTGETDATABITMAP     107
#define IDM_OBJECTGETDATAMETAFILEPICT   108
#define IDM_OBJECTEXIT              109

#define IDM_ADVISEMIN               200
#define IDM_ADVISETEXT              (IDM_ADVISEMIN+CF_TEXT)
#define IDM_ADVISEBITMAP            (IDM_ADVISEMIN+CF_BITMAP)
#define IDM_ADVISEMETAFILEPICT      (IDM_ADVISEMIN+CF_METAFILEPICT)
#define IDM_ADVISEGETDATA           300
#define IDM_ADVISEREPAINT           301

//DATAUSER.CPP
LRESULT FAR PASCAL __export DataUserWndProc(HWND, UINT, WPARAM,
LPARAM);
```

```
class __far CImpIAdviseSink;

/*
 * Application-defined classes and types.
 */

class __far CAppVars
   {
   friend LRESULT FAR PASCAL __export DataUserWndProc(HWND, UINT,
WPARAM, LPARAM);
   friend class CImpIAdviseSink;

   protected:
      HINSTANCE      m_hInst;          //WinMain parameters
      HINSTANCE      m_hInstPrev;
      UINT           m_nCmdShow;

      HWND           m_hWnd;           //Main window handle
      BOOL           m_fEXE;           //For tracking menu selection.

      LPADVISESINK   m_pIAdviseSink;   //Our CImpIAdviseSink
      DWORD          m_dwConn;         //Advise connection
      UINT           m_cfAdvise;       //Advise format
      BOOL           m_fGetData;       //::GetData on data change?
      BOOL           m_fRepaint;       //Repaint on data change?

      LPDATAOBJECT   m_pIDataSmall;
      LPDATAOBJECT   m_pIDataMedium;
      LPDATAOBJECT   m_pIDataLarge;

      LPDATAOBJECT   m_pIDataObject;   //Current selection
      UINT           m_cf;
      STGMEDIUM      m_stm;            //Current rendering we obtained

      BOOL           m_fInitialized;   //Did CoInitialize work?

   public:
      CAppVars(HINSTANCE, HINSTANCE, UINT);
      ~CAppVars(void);
      BOOL FInit(void);
      BOOL FReloadDataObjects(BOOL);
      void TryQueryGetData(LPFORMATETC, UINT, BOOL, UINT);
      void Paint(void);
   };

typedef CAppVars FAR * LPAPPVARS;
```

```
#define CBWNDEXTRA              sizeof(LONG)
#define DATAUSERWL_STRUCTURE    0

//This lives with the application to get OnDataChange notifications.

class __far CImpIAdviseSink : public IAdviseSink
   {
   protected:
      ULONG             m_cRef;     //Interface reference count.
      LPAPPVARS         m_pAV;      //Back pointer to the application

   public:
      CImpIAdviseSink(LPAPPVARS);
      ~CImpIAdviseSink(void);

      STDMETHODIMP QueryInterface(REFIID, LPVOID FAR *);
      STDMETHODIMP_(ULONG) AddRef(void);
      STDMETHODIMP_(ULONG) Release(void);

      //We only implement OnDataChange for now.
      STDMETHODIMP_(void)  OnDataChange(LPFORMATETC, LPSTGMEDIUM);
      STDMETHODIMP_(void)  OnViewChange(DWORD, LONG);
      STDMETHODIMP_(void)  OnRename(LPMONIKER);
      STDMETHODIMP_(void)  OnSave(void);
      STDMETHODIMP_(void)  OnClose(void);
   };

typedef CImpIAdviseSink FAR * LPIMPIADVISESINK;

#endif //_DATAUSER_H_
```

The file IADVSINK.CPP was purposely left out of Listing 6-3; that particular file is given in the context of "Advising and Notification with Data Objects" below.

While running, DATAUSER keeps one of its three objects as the 'current' one on which all calls are made. By default this will be the small data set object. To change the current object you can use DATAUSER's "Data Object" menu that provides other functions as well:

1. Switch between DLL objects and EXE objects. The switch involves releasing the current objects, creating the new ones using CoCreateInstance with CLSCTX_INPROC_SERVER (DLL) or CLSCTX_LOCAL_SERVER (EXE), and resetting the state of the application.
2. Change which of the three objects is current, either small, medium, or large.
3. Call ::QueryGetData on the current object for the formats CF_TEXT, CF_BITMAP, CF_DIB, CF_METAFILEPICT, and CF_WAVE, displaying the results (and what whether that was the *expected* result) to DATAUSER's window. This is the equivalent of calling IsClipboardFormatAvailable for such a data transfer, only we must pass a FORMATETC instead of a single clipboard format. The call is isolated in a function CAppVars::TryQueryGetData.
4. Call ::GetData on the current object to retrieve one of the data formats supported by the data objects. DATAUSER releases its old data (ReleaseStgMedium) and holds on to the new data to use

in repaints. This is similar to calling GetClipboardData except that we can describe our desired data using a FORMATETC and we receive the data back in a STGMEDIUM structure to which we pass a pointer. Once the data is returned to us we become responsible. A typical session with DATAUSER is shown in Figure 6-1 where the background of the window is painted with the metafile from the large data set.

Figure 6-1: The DATAUSER program painted with a large metafile showing the Data Object menu.

DATAUSER also has an "Advise" menu through which it sets up advisory connections with the current object. Such connections are important enough to warrant their own section. But before that I wanted to mention one small detail about IDataObject::SetData. You might notice that you have a *tymed* field in both the FORMATETC and the STGMEDIUM that you pass in this call. When calling ::SetData, make sure that both of these are identical.

Advising and Notification with Data Objects

One question I'm frequently asked is just how fast data updates can be sent from a data object to the user of that object. DATAUSER and both data object implementations were constructed to not only demonstrate data objects but to also answer this question. The IDataObject interface as described in the previous section has three member function called DAdvise, DUnadvise, and EnumDAdvise[1] that deal with notifications.

The overall notification mechanism of OLE 2.0 is used not only for data objects but for two other object types as well. Whatever agent is interested in being notified about data changes implements an object with the IAdviseSink interface:[2]

```
DECLARE_INTERFACE_(IAdviseSink, IUnknown)
  {
  [IUnknown methods included]

  //IAdviseSink methods
  STDMETHOD_(void,OnDataChange)(THIS_ FORMATETC FAR* pFormatetc,
    STGMEDIUM FAR* pStgmed) PURE;
  STDMETHOD_(void,OnViewChange)(THIS_ DWORD dwAspect, LONG lindex) PURE;
  STDMETHOD_(void,OnRename)(THIS_ LPMONIKER pmk) PURE;
  STDMETHOD_(void,OnSave)(THIS) PURE;
  STDMETHOD_(void,OnClose)(THIS) PURE;
  };
typedef    IAdviseSink FAR* LPADVISESINK;
```

This is a significant change from the way Windows has handled notification, or 'callbacks' up to this point. Traditionally any time you wanted some sort of notification, be it a window procedure, dialog procedure, an enumeration of clipboard formats or metafile records, DDE messages, and so forth, you were required to pass a pointer to a **single** function in your code to some other Windows API, usually sandwiched between calls to MakeProcInstance and FreeProcInstance. For example, to invoke a typical dialog box you must pass a pointer to your dialog procedure to a function like DialogBox:

```
DLGPROC   pfn;

pfn=(DLGPROC)MakeProcInstance((FARPROC)AboutProc, hInstance);
DialogBox(hInstance, MAKEINTRESOURCE(IDD_ABOUT), hWnd, pfn);
FreeProcInstance((FARPROC)pfn);
```

In all truth, a dialog procedure or any other window procedure is just a notification sink where the notifications are in the form of messages. In OLE 2.0 we want to expose the equivalent type of functionality for notification sinks through Windows Objects. Therefore instead of passing a single callback function to some API, you pass an interface pointer, that is a pointer to an **array of callback functions**, to the object from which you want to receive notifications. Instead of having one callback and a notification code, there is one member function of the advise sink object for each type of notification. This has the obvious advantage that each member function can have a different set of parameters and as many parameters as necessary to describe the event instead of forcing everything into the extreme bottleneck of wParams and lParams. Instead of there being one function associated with the connection and with all events, there is one object associated with the connection and one member function in that object associated with every event.

It's interesting to point out here that your trusted window procedure will eventually evolve into a bunch of member functions on something like your "Window Procedure" object (a *Windows Object*, of course). The Microsoft Foundation Classes already provide this mechanism today through their C++ class called CFrameWnd for which you define a 'message map' that associated today's messages, like WM_SIZE, with a member function in the class, like OnSize. Again, the benefit of the object-oriented approach is first that the parameters to OnSize can be separate variables for the new *cx* and *cy* instead of having them packed into lParam, and second that within OnSize you have a *this* pointer which you can use to immediately access all your window-related variables. Within a message procedure, you generally have to use GetWindowWord or GetWindowLong to retrieve variables associated with a particular hWnd.

All of this is why so much attention has been given to the "Notification" technology of OLE 2.0, because it does represent a significant departure, and improvement, from the old ways of traditional Windows programming. It also shows why the Microsoft Foundation Classes help you to adjust to this new style of

1Again, the names containing "D" identify these as belonging to IDataObject to eliminate conflicts with IOleObject::Advise, ::Unadvise, and ::EnumAdvise.

2There is also an interface called IAdviseSink2 which was added late in the development cycle of OLE 2.0 such that Microsoft could not change IAdviseSink itself. This additional interface has one addition member called OnLinkSourceChange.

programming, although MFC is still based on C++ classes and not on reusable component Windows Objects and must still be considered a tool and not the native representation of Windows' system capabilities.

Enough lecture–let's get back to some code. The only two members of IAdviseSink that are of interest to us now are OnDataChange and OnViewChange (see "View Objects"). The others are related to compound document objects that we'll see in Chapter 9 when we implement IAdviseSink as part of a compound document container. But for now, any general piece of code that wants data change notifications implements a stand-alone object with an implementation IAdviseSink. Again, the INTERFAC directory in the sample code for this book provides a generic CImpIAdviseSink. The DATAUSER application uses a slightly modified version of this that code to implement IAdviseSink as an interface on it's application object, CAppVars. The code for the interface implementation is shown in Listing 6-4.

IADVSINK.CPP

```
/*
 * IADVSINK.CPP
 *
 * Implementation of the IAdviseSink interface for the Data User, Chapter 6
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 *.

#include "datauser.h"


CImpIAdviseSink::CImpIAdviseSink(LPAPPVARS pAV)
   {
   m_cRef=0;
   m_pAV=pAV;
   return;
   }

CImpIAdviseSink::~CImpIAdviseSink(void)
   {
   return;
   }



STDMETHODIMP CImpIAdviseSink::QueryInterface(REFIID riid, LPVOID FAR *ppv)
   {
   *ppv=NULL;

   //Any interface on this object is the object pointer.
   if (IsEqualIID(riid, IID_IUnknown) || IsEqualIID(riid, IID_IAdviseSink))
      *ppv=(LPVOID)this;

   /*
    * If we actually assign an interface to ppv we need to AddRef it
```

```
    * since we're returning a new pointer.
    */
   if (NULL!=*ppv)
      {
      ((LPUNKNOWN)*ppv)->AddRef();
      return NOERROR;
      }

   return ResultFromScode(E_NOINTERFACE);
   }

STDMETHODIMP_(ULONG) CImpIAdviseSink::AddRef(void)
   {
   return ++m_cRef;
   }
```

Listing 6-4: DATAUSER's IAdviseSink implementation that's only interested in OnDataChange.

```
STDMETHODIMP_(ULONG) CImpIAdviseSink::Release(void)
   {
   ULONG   cRefT;

   cRefT=--m_cRef;

   if (0==cRefT)
      delete this;

   return cRefT;
   }



STDMETHODIMP_(void) CImpIAdviseSink::OnDataChange(LPFORMATETC pFE
   , LPSTGMEDIUM pSTM)
   {
   BOOL       fUsable=TRUE;
   UINT       cf;
   STGMEDIUM   stm;

   /*
    * We first check that the changed data is, in fact, a format we're
    * interested in, either CF_TEXT, CF_BITMAP, or CF_METAFILEPICT, then
    * only in the aspects we want.  We then check if pSTM->tymed is
    * TYMED_NULL or something else.  If NULL, we just exit so the
    * data object can time ADVF_NODATA transactions.  Otherwise we
```

```
 * verify that the data is useful and repaint.
 *
 * If there is data in pSTM we are responsible for it.
 */

//Ignore the m_fGetData flag for EXE objects (we can't GetData)
if (!m_pAV->m_fGetData && !m_pAV->m_fEXE)
   return;

//See if we're interested
cf=pFE->cfFormat;

if ((CF_TEXT!=cf && CF_BITMAP!=cf && CF_METAFILEPICT!=cf)
   || !(DVASPECT_CONTENT & pFE->dwAspect))
   return;

//Check media types
switch (cf)
   {
   case CF_TEXT:
      fUsable=(BOOL)(TYMED_HGLOBAL & pFE->tymed);
      break;

   case CF_BITMAP:
      fUsable=(BOOL)(TYMED_GDI & pFE->tymed);
      break;

   case CF_METAFILEPICT:
      fUsable=(BOOL)(TYMED_MFPICT & pFE->tymed);
      break;

   default:
      break;
   }

if (!fUsable)
   return;

if (NULL==m_pAV->m_pIDataObject)
   return;

/*
 * When dealing with EXE objects, invalidate ourselves after setting
 * TYMED_NULL in our STGMEDIUM that causes CAppVars::Paint to request new
 * data.  We cannot call ::GetData in here because this is an async call
 * when we're dealing with an EXE.
 */
```

```
   if (m_pAV->m_fEXE)
      {
      ReleaseStgMedium(&(m_pAV->m_stm));
      m_pAV->m_cf=cf;
      m_pAV->m_stm.tymed=TYMED_NULL;

      InvalidateRect(m_pAV->m_hWnd, NULL, TRUE);
      return;
      }

   if (FAILED(m_pAV->m_pIDataObject->GetData(pFE, &stm)))
      return;

   //Get rid of old data and update.
   ReleaseStgMedium(&(m_pAV->m_stm));

   m_pAV->m_cf=cf;
   m_pAV->m_stm=stm;

   InvalidateRect(m_pAV->m_hWnd, NULL, TRUE);

   if (m_pAV->m_fRepaint)
      UpdateWindow(m_pAV->m_hWnd);

   return;
   }



STDMETHODIMP_(void) CImpIAdviseSink::OnViewChange(DWORD dwAspect, LONG
lindex)
   {
   return;
   }

STDMETHODIMP_(void) CImpIAdviseSink::OnRename(LPMONIKER pmk)
   {
   return;
   }

STDMETHODIMP_(void) CImpIAdviseSink::OnSave(void)
   {
   return;
   }

STDMETHODIMP_(void) CImpIAdviseSink::OnClose(void)
```

```
{
return;
}
```

An Object User Providing an Object?

DATAUSER's implementation of the IAdviseSink interface on it's CAppVars class is the first instance we've seen thus far showing how the user of one object on occasion needs to implement an object of its own. In this case, the user of the data object implements the advise sink object and hands that advise sink object to the data object. The data object, in turn, becomes a user itself of that advise sink object, and it obtained the pointer through a call to one of its **own** member functions. Such a pattern is common in OLE 2.0 applications where such applications are both object user and object implementor. How we classify applications into things like Containers and Servers depends on what interfaces and objects those applications implement and how they expose them. In any case, all objects implemented in any type of application are still Windows Objects. This is why this and all other chapters use the words 'object' and 'object user' instead of the far-too-specific 'container' and 'server' as you may have seen in generally use in the OLE 2.0 Programmer's Reference.

Establishing an Advisory Connection

When you are interested in knowing when data changes you need to follow a few straightforward steps:

1. Implement the IAdviseSink interface on an appropriate object. This may be a site in which the data lives, the document into which the data is integrated, or the application as a whole. IAdviseSink should live wherever you wish to see the notifications. The only member function you need worry about at this point is ::OnDataChange.
2. When you wish to set up an advise, instantiate the advise sink and pass it to IDataObject::DAdvise along with the FORMATETC describing the data you're interested in an flags indicating how you want the advise to happen. ::DAdvise will return a DWORD key to identify your connection. In ::Advise the data object will ::AddRef your advise sink.
3. When you wish to terminate the connection, pass the key from ::DAdvise to ::DUnadvise. The data object will ::Release your advise sink.

An advise in OLE 2.0 is specifically associated with an aspect in the FORMATETC that you pass to ::DAdvise. The data object can refuse to establish an advise if it does not support that format and uses the aspect to determine whether or not to send you a notification. For that reason you will receive an ::OnDataChange whenever the data object affects the rendering of an aspect, not specifically the rendering of an aspect in one clipboard format. In other words you will receive notifications regardless of the specific format you are using for a given aspect.

DATAUSER instantiates its IAdviseSink interface implementation on startup before creating any objects:

```
m_pIAdviseSink=new CImpIAdviseSink(this);

if (NULL==m_pIAdviseSink)
   return FALSE;

m_pIAdviseSink->AddRef();
```

The extra AddRef on the interface accounts for the pointer in m_pIAdviseSink because the CImpIAdviseSink itself initializes its reference count to zero. When DATAUSER wants to establish an advise it fills a FORMATETC and calls the ::DAdvise for the current data object:

```
m_pIDataObject->DAdvise(&fe, ADVF_NODATA, m_pIAdviseSink, &m_dwConn);
```

where m_pIDataObject is the current object, fe is a FORMATETC structure containing DVASPECT_CONTENT, ADVF_NODATA is one of a number of inclusive advise flags shown in Table 6-1, m_pIAdviseSink is the advise sink object in which we've implemented OnDataChange, and &m_dwConn is the address in which the data object should store the connection key. When we later terminate the connection we just pass this key to DUnadvise:

```
m_pIDataObject->DUnadvise(m_dwConn);
```

DATAUSER passes ADVF_NODATA to IDataObject::DAdvise to prevent the data object from rendering the data and sending it to the IAdviseSink::OnDataChange function in the LPSTGMEDIUM parameter. By default, data objects will go ahead and provide the updated data, so if you normally would ask for it every time it changed, you can omit this flag an have the data given to you automatically.

Flag	Meaning
ADVF_NODATA	Prevents the data object from sending data along with the OnDataChange notification. This is equivalent to a DDE 'warm link'; absence of this flag is equivalent to a DDE 'hot link.' In any case, when ADVF_NODATA is used the tymed field of the LPSTGMEDIUM passed to OnDataChange will usually contain TYMED_NULL. Some data objects may still send data and tymed will contain another value, and in that case OnDataChange is still responsible for the STGMEDIUM. Be sure to check.
ADVF_ONLYONCE	Automatically terminates the advisory connection after the first call to OnDataChange. It is not necessary to call DUnadvise when you use this flag. You may still, however, call DUnadvise if you have not yet received a notification.
ADVF_PRIMEFIRST	Causes an initial OnDataChange call even when the data has not changed

	from its present state. If you combine ADVF_PRIMEFIRST with ADVF_ONLYONCE you create a single asynchronous IDataObject::GetData call.
ADVF_DATAONSTOP	When provided with ADVF_NODATA causes the last OnDataChange sent from the data object to actually provide the data, that is, pSTM->tymed will be a value other than TYMED_NULL. This flag is meaningless without ADVF_NODATA.

Table 6-1: The advise flags usable with IDataObject::DAdvise.

DATAUSER allows you to control when and how it handles the OnDataChange notifications through its Advise menu as shown in Figure 6-2 (the image is the large bitmap). The three format items indicate which format DATAUSER will work with in OnDataChange. By default, OnDataChange will do nothing, enabling us to test how many notifications per second are possible with a do-nothing advise sink. The other two menu items turn on options for OnDataChange: "GetData on Change" causes OnDataChange to call IDataObject::GetData for every notification (to test the speed of rendering the data) and "Repaint on Change" will go the extra step of forcing a full window repaint on each notification (DATAUSER always invalidates the window by doesn't always call UpdateWindow). These two options together would allow us to watch dynamic changes to the data as they occur, although the data objects we have for this chapter don't bother changing the data at all. Since DATAUSER allows menu control over when and if the data is actually rendered for each notification, it initially passes ADVF_NODATA to IDataObject::DAdvise to prevent the data objects from automatically re-rendering the data on every ::OnDataChange. This way we can measure the performance of notifications with and without ::GetData calls.

Figure 6-2: The DATAUSER program painted with a large bitmap showing the Advise menu options

Sending Notifications as a Data Object

To provide for notification within the implementation of a data object you must implement ::DAdvise, ::DUnadvise, and ::EnumDAdvise of IDataObject. But there is a catch. OLE 2.0 specifies that any data object may receive *multiple* calls to ::DAdvise and that the object is then responsible for sending notifications to each and every one of those advise sinks. In other words the data object is responsible for tracking the advise sinks:

In addition the data object must determine who to notify when a particular aspect changes, that is, go back through its list of advise sinks, find those that asked for a specific aspect, and call those IAdviseSink::OnDataChange functions. All of this can be a royal pain, so OLE 2.0 provides an object called the "Data Advise Holder" that implements an interface called IDataAdviseHolder:

```
DECLARE_INTERFACE_(IDataAdviseHolder, IUnknown)
   {
   [The omnipresent IUnknown methods included]

   //IDataAdviseHolder methods
   STDMETHOD(Advise)(THIS_ LPDATAOBJECT pDataObject, FORMATETC FAR* pFetc,
      DWORD advf, LPADVISESINK pAdvise, DWORD FAR* pdwConnection) PURE;
   STDMETHOD(Unadvise)(THIS_ DWORD dwConnection) PURE;
   STDMETHOD(EnumAdvise)(THIS_ LPENUMSTATDATA FAR* ppenumAdvise) PURE;

   STDMETHOD(SendOnDataChange)(THIS_ LPDATAOBJECT pDataObject,
      DWORD dwReserved, DWORD advf) PURE;
   };

typedef     IDataAdviseHolder FAR* LPDATAADVISEHOLDER;
```

The data advise holder trivializes implementation of a data object's ::Advise members. When the data object sees its first ::DAdvise, (or when initially created), it can create a data advise holder and delegate the initial ::DAdvise request on to the holder:

From then on the data object delegates any ::DUnadvise and ::EnumDAdvise calls to the same data advise holder which will manage all the advise sinks for us.

```
STDMETHODIMP CImpIDataObject::DAdvise(LPFORMATETC pFE, DWORD dwFlags
   , LPADVISESINK pIAdviseSink, LPDWORD pdwConn)
   {
   LPCDataObject   pObj=(LPCDataObject)m_pObj;
   HRESULT         hr;

   if (NULL==pObj->m_pIDataAdviseHolder)
      {
      hr=CreateDataAdviseHolder(&pObj->m_pIDataAdviseHolder);

      if (FAILED(hr))
         return ResultFromScode(E_OUTOFMEMORY);
      }

   hr=pObj->m_pIDataAdviseHolder->Advise((LPDATAOBJECT)this, pFE
      , dwFlags, pIAdviseSink, pdwConn);

   return hr;
   }

STDMETHODIMP CImpIDataObject::DUnadvise(DWORD dwConn)
   {
   LPCDataObject   pObj=(LPCDataObject)m_pObj;
   HRESULT         hr;

   if (NULL==pObj->m_pIDataAdviseHolder)
      return ResultFromScode(E_FAIL);

   return pObj->m_pIDataAdviseHolder->Unadvise(dwConn);
```

```
    }

STDMETHODIMP CImpIDataObject::EnumDAdvise(LPENUMSTATDATA FAR *ppEnum)
    {
    LPCDataObject   pObj=(LPCDataObject)m_pObj;
    HRESULT         hr;

    if (NULL==pObj->m_pIDataAdviseHolder)
        return ResultFromScode(E_FAIL);

    return pObj->m_pIDataAdviseHolder->EnumAdvise(ppEnum);
    }
```

When the data object wants to send notifications, it asks the data advise holder to perform the honors by calling IDataAdviseHolder::SendOnDataChange. To this function you pass a pointer to the data object (so the holder can call ::GetData as necessary), a zero (a reserved value), and any combination of the flags shown in Table 6-1. By passing the ADVF_NODATA you can prevent the holder from asking you for data before it notifies interested advise sinks. What you end up with by using the data advise holder is a little loss of performance (for a few more function calls) with the return of great convenience.

The data objects, both in DLL and EXE, create small "Advisor" windows, one for each instantiated objects as shown in Figure 6-3. Each Advisor window sports an "Iterations" menu that allow you to fire off a number of continuous notifications from 16 to 572 (that is, $16*n^2$ where n progresses from 1 to 6). These advisor windows are created from within the data objects themselves–there is no problem creating a window from within a DLL in this manner.

Figure 6-3: The DATAUSER program with Advisor windows from three data objects in use.

Whenever you select one of the numbers from the Iterations menu, the object enters a loop in its AdvisorWndProc (that receives the WM_COMMAND for that menu item). Within this loop it calls the IDataAdviseHolder::SendOnDataChange a given number of times then reports the number of milliseconds it took for all the calls along with an average time for each call:

```
DWORD   dwTime, dwAvg;
char    szTime[128], szTitle[80];
UINT    i;

[In a WM_COMMAND message case]

if (NULL==m_pIDataAdviseHolder)
    break;

dwTime=GetTickCount();

i=0;
while (TRUE)
    {
    pDO->m_pIDataAdviseHolder->SendOnDataChange(
        pDO->m_pIDataObject, 0, ADVF_NODATA);

    if (++i >= iAdvise)
        break;
    }

dwTime=GetTickCount()-dwTime;
dwAvg=dwTime/iAdvise;

wsprintf(szTime, "Total\t=%lu milliseconds\n\rAverage\t=%lu milliseconds"
    , dwTime, dwAvg);

GetWindowText(hWnd, szTitle, sizeof(szTitle));
MessageBox(hWnd, szTime, szTitle, MB_OK);
```

I encourage you to play with both DLL and EXE data objects to see just how fast DATAUSER can receive notifications from both on your particular machine. You will notice immediately that notifications from the EXE object are slower simply due to the LRPC layer in between and how the presence of that layer affects notification.

Special Considerations for Remoted Notifications

When a data object lives in an EXE as opposed to a DLL, there are some significant performance penalties due to the marshaling of the IAdviseSink::OnDataChange calls between the data object and the user. Blasting data rapidly in the code shown above works fine for a DLL but will eventually cause the message queue in the user application to overflow if you exceed the size of that application's message queue (which is generally why OLE 2.0 applications should use a message queue size of 96). To prevent this from happening, and to give the advise sink application a change to catch up, the EXE version of our data object, EDATAOBJ, has a slight modification to the loop shown above that differs from that shown in Listing 6-1. Note that this explicit yielding is only necessary under Windows 3.1 and not under Windows NT since the latter is a preemptive system to begin with:

```
i=0;
while (TRUE)
   {
  #ifdef EXEDATAOBJECT
  #ifndef WIN32
   MSG    msg;

   if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
      {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
      }
   else
  #endif
  #endif
      {
      pDO->m_pIDataAdviseHolder->SendOnDataChange(
         pDO->m_pIDataObject, 0, ADVF_NODATA);

      if (++i >= iAdvise)
         break;
      }
   }
```

When a call to IAdviseSink::OnDataChange is made from one application to another, that is, across a process boundary where marshaling is necessary, the component object library generates an asynchronous call. Most other marshaled function calls are synchronous, but notifications between applications are designed asynchronous since the source application generally doesn't want to wait around for possible consumers to use that data. This is not, however, without some impact on the consumer's implementation of IAdviseSink, and for that we need to look specifically at an ::OnDataChange implementation.

Inside the Advise Sink

The DATAUSER program implements an advise sink object with the IAdviseSink interface but since DATAUSER only ever hands a pointer to this object to IDataObject::DAdvise, it only needs to implement the single IAdviseSink::OnDataChange member.

You may have already noticed while running DATAUSER that it has two items in the "Advise" menu in addition to those that allow you to select the advise format: "GetData on Change" and "Paint on Change: that change DATAUSER's behavior inside ::OnDataChange:

- When you toggle "GetData on Change" you toggle the m_fGetData flag in the instantiated CAppVars (C++) object. If this flag is not set inside IAdviseSink::OnDataChange, DATAUSER returns immediately so as to execute the member function as fast as possible:

```
STDMETHODIMP_(void) CImpIAdviseSink::OnDataChange(LPFORMATETC pFE
   , LPSTGMEDIUM pSTM)
   {
   if (!m_pAV->m_fGetData)
```

```
return;
```

- If m_fGetData is set, the we check the format of the data specified in the FORMATETC parameter to ::OnDataChange, and if we can use it, we fire off a call to IDataObject::GetData, store that data in the CAppVars object, and invalidate (but not update) the main window to cause an eventual repaint:

```
if (m_pAV->m_fRepaint)
   UpdateWindow(m_pAV->m_hWnd);
```

This is precisely why DATAUSER sets up an advise with ADVF_NODATA: the m_fGetData flag determines whether or not the data object should render the data on each notification.

- When you toggle "Paint on Change" you toggle the CAppVars member m_fRepaint. This flag has no affect if m_fGetData is not set; only when ::OnDataChange has retrieved the data and invalidated the window, m_fRepaint determines whether or not to immediately call UpdateWindow:

```
if (m_pAV->m_fRepaint)
   UpdateWindow(m_pAV->m_hWnd);
```

Now we have a slight complication when we are using the EXE version EDATAOBJ: notifications are asynchronous whereas a notification from a DLL is always synchronous (since it's a direct function call in all cases). The complication, which will be familiar to readers who worked in OLE 1.0, is that during an asynchronous call the callee, in this case IAdviseSink::OnDataChange, cannot make any other function calls that might require remoting. So if we are using the EXE-based data object, our IAdviseSink implementation cannot call IDataObject::GetData from within our ::OnDataChange. Bummer.

To account for this, DATAUSER disables the "GetData on Change" and "Paint on Change" menu items when you are using EXE data objects, that is, the CAppVars member m_fEXE is TRUE. The implementation of IAdviseSink::OnDataChange uses m_fEXE to decide if it should call IDataObject::GetData. So first of all, since m_fGetData is meaningless when m_fEXE is TRUE, the code shown above for dealing with m_fGetData requires a slight modification:

```
if (!m_pAV->m_fGetData && !m_pAV->m_fEXE)
   return;
```

If we are within an asynchronous ::OnDataChange call and we do attempt to call IDataObject::GetData, we'll get a return SCODE of RPC_E_CANTCALLOUT_INASYNCCALL which tells you exactly what's going on. Therefore DATAUSER has to implement somewhat different behavior when data changes such that the actual call to IDataObject::GetData is delayed until after we've exited IAdviseSink::OnDataChange:

```
if (m_pAV->m_fEXE)
   {
   ReleaseStgMedium(&(m_pAV->m_stm));  //Clean up any old data.
   m_pAV->m_cf=cf;
   m_pAV->m_stm.tymed=TYMED_NULL;

   InvalidateRect(m_pAV->m_hWnd, NULL, TRUE);
   return;
   }
```

This code in itself means very little until you examine the top of the CAppVars::Paint function in DATAUSER.CPP:

```
hDC=BeginPaint(m_hWnd, &ps);

 if (m_fEXE)
    {
    if (TYMED_NULL==m_stm.tymed && 0!=m_cf)
       {
       SETDefFormatEtc(fe, m_cf
          , TYMED_HGLOBAL | TYMED_MFPICT | TYMED_GDI);

       if (NULL!=m_pIDataObject)
          m_pIDataObject->GetData(&fe, &m_stm);
       }
    }

 [Other code to perform painting]
```

What happens here is that in IAdviseSink::OnDataChange, DATAUSER stores the clipboard format that changed in the CAppVars object but stores a TYMED_NULL to indicate that it actually has no data on hand with which to paint. Since ::OnDataChange still invalidates the window, we'll eventually see a WM_PAINT message but that message will occur outside the asynchronous LRPC call. Therefore our WM_PAINT handling in the code above can call IDataObject::GetData if it sees that we're using the EXE data object and that we don't already have data on hand (m_stm.tymed is TYMED_NULL). This call to GetData will always succeed (at least the calling, not necessarily the rendering) so we can then repaint with that updated data.

IDataObject as a Standard for Object Data Transfer

As I've mentioned already, IDataObject is the standardized interface for data exchange. It therefore makes sense that any object capable of performing any type of data transfer do as much as they can through IDataObject instead of any custom interfaces. We can see this in practice with the Polyline component object we've been developing here. The versions of the IPolyline interface up to this point (those in chapters 4 and 5) contain a number of custom data transfer functions:

```
DECLARE_INTERFACE_(IPolyline5, IUnknown)
    {
    [Other members.]

    STDMETHOD(DataSet)     (THIS_ LPPOLYLINEDATA, BOOL, BOOL) PURE;
    STDMETHOD(DataGet)     (THIS_ LPPOLYLINEDATA) PURE;
    STDMETHOD(DataSetMem)   (THIS_ HGLOBAL, BOOL, BOOL, BOOL) PURE;
    STDMETHOD(DataGetMem)   (THIS_ HGLOBAL FAR *) PURE;
    STDMETHOD(RenderBitmap) (THIS_ HBITMAP FAR *) PURE;
    STDMETHOD(RenderMetafile) (THIS_ HMETAFILE FAR *) PURE;
    STDMETHOD(RenderMetafilePict) (THIS_ HGLOBAL FAR *) PURE;

    [Other members.]
    }
```

In other words, IPolyline has so far had two separate functions for exchange of its native data in both Get and Set directions: DataSet and DataGet operate on a pointer medium (which is used internally in the Polyline for the most part), DataSetMem and DataGetMem operate on an HGLOBAL medium. This interface has also maintained three separate member functions to render specific formats. But since we can express any data format through a FORMATETC, and since we can express storage mediums like HGLOBAL and HBITMAP, we can replace all of these functions by implementing IDataObject on Polyline, just as we replaced the specific file storage functions in Chapter 5 with IPersistStorage.

In addition, Polyline has been traveling along with another interface IPolylineAdviseSink with members like OnDataChange, OnPointChange, etc. Here as well we remove the OnDataChange member from this interface and replace it with use of the standard IAdviseSink. Therefore anyone using this object that is unaware of the custom IPolyline or IPolylineAdviseSink interfaces can still ask for data change notifications using the standard interfaces.

Overall the changes to both Polyline and its user, Component Schmoo are listed below with the revised IPolyline interface, IPolyline6, defined in INC\IPOLY6.H. I won't give code listings here since none of the code explains anything that hasn't already been covered in both DATAUSER and DDATAOBJ above.

- Polyline has two additional files, IDATAOBJ.CPP and IENUMFE.CPP, which implement IDataObject (on the CPolyline object) and IEnumFORMETETC. CoSchmoo adds IADVSINK.CPP in which it receives OnDataChange notifications from the Polyline. CoSchmoo implements the IAdviseSink as part of it's CSchmooDoc object and has removed the OnDataChange member from its implementation of IPolylineAdviseSink.
- Polyline internally maintains a DataAdviseHolder object for tracking IAdviseSink pointers it sees through IDataObject. Where it used to call IPolylineAdviseSink::OnDataChange it now calls IDataAdviseHolder::SendOnDataChange.
- The IPolyline::DataSet and ::DataGet members, since they are used internally by the CPolyline object in it's IPersistStorage implementation, are been moved out of the public interface into private member functions of the C++ CPolyline class.
- Polyline now uses a registered clipboard format to identify its native data rather than a separate function call. The string is defined as SZPOLYLINECLIPFORMAT in INC\IPOLY6.H.
- The internal CPolyline class carries FORMATETC arrays for both IDataObject::GetData and IDataObject::SetData. GetData handles Polyline's native format in TYMED_HGLOBAL, CF_METAFILEPICT in TYMED_MFPICT, and CF_BITMAP in TYMED_GDI, in that order. SetData only

handles Polyline's native format in TYMED_HGLOBAL. These structures are initialized in CPolyline::CPolyline.

- Component Schmoo implements a set of IUnknown members on it's CSchmooDoc class in order to support the interface implementation of IAdviseSink.
- All of Component Schmoo's clipboard handling in CSchmooDoc::RenderFormat and CSchmooDoc::FPaste now work through the IDataObject interface on its Polyline object. Since Component Schmoo only maintains an IPolyline6 pointer at all times, it obtains the IDataObject pointer when necessary through QueryInterface.

All these changes again move Polyline closer to becoming a compound document object in Chapter 10. With both IPersistStorage and IDataObject in place, Polyline is already halfway towards that goal. With an implementation of IOleObject it will be embeddable in any compound document container as well as still be usable by Component Schmoo, for containers will communicate with Polyline through standard interfaces whereas Component Schmoo, which knows Polyline in somewhat of the biblical sense, will continue to communicate through the custom IPolyline interface. This again reinforces a great benefit of interfaces and the QueryInterface mechanism: Polyline can support two different types of users (component user with inside knowledge and a compound document container with no extra knowledge) by implementing additional interfaces which do not interfere with one another.

View Objects and the IViewObject Interface

A view object is very similar to a data object in that it's almost always just another way of treating some other more complex object. For that reason you obtain an IViewObject pointer almost invariably by calling QueryInterface on another object. In this case, a 'view object' is any object that you can use through an IViewObject interface pointer which allows you to ask the object to draw itself or otherwise manage details about its graphical presentations. The interface exists so that an object user can ask an object to render directly onto a device instead of into some transfer medium:

```
DECLARE_INTERFACE_(IViewObject, IUnknown)
  {
  [IUnknown methods included]

  STDMETHOD(Draw) (THIS_ DWORD dwDrawAspect, LONG lindex,
        void FAR * pvAspect, DVTARGETDEVICE FAR * ptd,
        HDC hicTargetDev, HDC hdcDraw,
        const LPRECTL lprcBounds, const LPRECTL lprcWBounds,
        BOOL (CALLBACK * pfnContinue) (DWORD),
        DWORD dwContinue) PURE;

  STDMETHOD(GetColorSet) (THIS_ DWORD dwDrawAspect, LONG lindex,
        void FAR* pvAspect, DVTARGETDEVICE FAR * ptd,
        HDC hicTargetDev, LPLOGPALETTE FAR* ppColorSet) PURE;

  STDMETHOD(Freeze)(THIS_ DWORD dwDrawAspect, LONG lindex,
        void FAR* pvAspect, DWORD FAR* pdwFreeze) PURE;
  STDMETHOD(Unfreeze) (THIS_ DWORD dwFreeze) PURE;

  STDMETHOD(SetAdvise) (THIS_ DWORD aspects, DWORD advf,
        LPADVISESINK pAdvSink) PURE;
  STDMETHOD(GetAdvise) (THIS_ DWORD FAR* pAspects, DWORD FAR* pAdvf,
        LPADVISESINK FAR* ppAdvSink) PURE;
  };

typedef   IViewObject FAR* LPVIEWOBJECT;
```

WARNING: IViewObject, although a standard interface, does not have marshaling support, that is, it is not possible to move an IViewObjectPointer across a process boundary under Windows. The primary problem is that the hDC parameter to IViewObject::Draw cannot be passed to another process because device contexts are only meaningful in the context of one process. Therefore IViewObject can only be implemented from objects in server DLLs or object handlers. In the case where an object in a server EXE requires use of IViewObject it must provide an object handler DLL to work in with the server EXE.

Asking an object "can you draw yourself an hDC I provide?" is the same as calling QueryInterface with IID_IViewObject. If the object is not capable, or if the object is in an EXE, the answer will be "no." If, however, you are given the IViewObject pointer, you can do a number of things with it, the most common of which will be to ask the object to draw itself. When the object is asked to draw, it's responsible for leaving

the hDC in the same state as it was received, which is what common sense would dictate. If you need to change the mapping mode or any other aspect about the hDC, be sure to SaveDC on entry to IViewObject::Draw and RestoreDC on exit.

You may have noticed already that three of the parameters to IViewObject::Draw are an aspect (*dwAspect*), a piece index (*lindex*), and a pointer to a target device structure (*ptd*). These are three of the same fields in a FORMATETC and are used pretty much in the same way to identify the data you want to draw, much like you identify the data to render through IDataObject::GetData. There is no clipboard format nor is there a storage medium involved with ::Draw, however, since we are not trying to get the data ourselves but rather want it placed on an hDC which can be thought here to identify both the format and the medium.

IViewObject::Draw

The Draw member function in IViewObject probably has the longest parameter list in all of OLE 2.0 because it encompasses such rich functionality. It allows you to tell an object exactly what to draw, where to draw it, how to draw it, and even lets you supply a callback function (yes, we still use a few of them) if you want to have the ability to break out of long repaints. Since this function covers so much ground, let's first examine the simplest use of this function which would be in the form:

```
pIViewObject->Draw(DVASPECT_CONTENT, -1, NULL, NULL, 0, hDC, &rcl
  , NULL, NULL, 0);
```

This line of code means "draw the full rendering of the object (DVASPECT_CONTENT, -1) on a rectangle (rcl) on this hDC." The object will draw whatever is meant by its 'full representation' directly to the given hDC, scaled to the rectangle you provide. Always express the rectangle in the units of the current mapping mode in the hDC. The hDC may be the screen, a printer, a metafile, or a memory device context and the object will not really care. So an object that typically generates bitmaps may implement this function with a simple StretchBlt call, or if it normally uses a metafile would call PlayMetafile after setting the window and viewport extents properly for your rectangle.

As a view object user, whenever you changed the size of the object's display area you would simply repaint the object passing a different rectangle to ::Draw. When you drew the object on the screen while processing WM_PAINT, you would pass the hDC from BeginPaint; if you were printing whatever document contained the object you would pass the hDC from a CreateDC call on the printer device. You might also have a reason to choose a different aspect as the first parameter, such as DVASPECT_THUMBNAIL if you were drawing the object in a print preview mode, or DVASPECT_ICON if you preferred a small, compact presentation. So in the most simple uses of IViewObject::Draw, you are concerned with the following four parameters:

- *dwAspect* (DWORD): One of DVASPECT_CONTENT, DVASPECT_THUMBNAIL, DVASPECT_ICON, or DVASPECT_DOCPRINT with identical definitions as the same field in FORMATETC.
- *lindex* (LONG): The index of the piece as in FORMATETC. In OLE 2.0 this should always be -1.
- *hDC* (HDC): The device context on which to draw. Must not be NULL.
- *lprcBounds* (RECTL FAR *): A pointer to the RECTL that contains the rectangle in which the object should draw, scaling the presentation as necessary to fit this rectangle. This parameter canoot be NULL. Be sure not to pass a pointer to a typical Windows RECT structure under Windows 3.1–a RECTL is twice as large, so passing a RECT is living dangerously. Under Windows NT, a RECT is identical to a RECTL.

Note also that the *pvAspect* parameter exists to provide further information for a specific *dwAspect*. Under OLE 2.0, this parameter is not used as none of the standard aspects support additional information of this nature. Therefore this parameter is always NULL and will not be discussed further.

OLE 2.0 provides the API function **OleDraw** which streamlines this most common case of calling IViewObject::Draw. OleDraw is defined in OLE2.H and lives in OLE2.DLL. The exact implementation of the function is not hard to guess or reproduce:

```
STDAPI OleDraw(LPUNKNOWN pUnknown, DWORD dwAspect, HDC hdcDraw
  , LPCRECT lprcBounds);
  {
  HRESULT     hr;
  LPVIEWOBJECT pIViewObject;

  if (NULL!=pUnknown)
    {
    hr=pUnknown->QueryInterface(IID_IViewObject
```

```
        , (LPVOID FAR *)&pIViewObject);

      if (SUCCEEDED(hr))
        {
        pIViewObject->Draw(DVASPECT_CONTENT, -1, NULL, NULL, 0
          , hDC, lprcBounds, NULL, NULL, 0);

        pIViewObject->Release();
        }
      }

    return;
    }
```

OleDraw is provided as an API because you normally don't hold on to an IViewObject pointer for any other reason, and it would be cumbersome if you always had to QueryInterface yourself every time you just want to draw normally. However, there are many times where you want more control over the exact rendering such that OleDraw becomes a straight jacket in which case you are always free to QueryInterface for IViewObject and call ::Draw yourself. Let's look at some of the cases where you might want more control.

Rendering for a Specific Device

Many applications, especially high-end graphics or desktop-publishing packages are very concerned about getting the best possible output on a printer as well as the fastest possible output. These applications will then want to tell all view object about the intended device when calling ::Draw. Two parameters of ::Draw handle this consideration:

- *ptd* (LPDVTARGETDEVICE): A pointer to a target device structure identical to the one in FORMATETC that describes the exact device, generally a printer, for which the object is to render its image. A NULL means "display."
- *hicTargetDev* (HDC): An **information** context for the target device described by *ptd* from which the object can extract device metrics and test the capabilities of that device. If the *ptd* parameter is NULL the object should not use this parameter, regardless of its actual value.

There are two primary cases where you might pass non-NULL values to these functions. The first and most obvious is when hDC is a printer device context and you want the objects to render as accurately as possible for that printer. In this case you can describe the printer device in *ptd* and pass either an information context you have on hand or the same hDC to which you are printing as the *hicTargetDev* parameter. You must pass something in *hicTargetDev* so you tell the object that you are, in fact, going to a printer, since in a printing case you will **always** have an *hDC* that can be treated like an information context. An example of when you would use both parameters is when you are printing to, say, a PostScript printer, and you want to let any object you're printing optimize for PostScript as well. The object, knowing this fact and knowing that hDC is a real printer DC, may choose to send PostScript commands directly to the printer (via Escape) instead of using GDI functions. The result will be highly optimized output for the printer and will generally show better performance.

The second case applies to situations like Print Preview modes, where the application wishes to draw the object on the screen but wants the object to act like it was drawing to the printer. In this case *ptd* will point to a valid DVTARGETDEVICE structure but *hicTargetDev* is NULL, meaning that the object should call GDI functions on *hDC* to draw what it would show on the given device. For example, if the object would normally show a magenta shading on the screen it may not want to draw the same shading to a printer that does not support color. By telling the object about your intended target device it may choose to either eliminate the shading or to draw in black and white dithering with a black and white device.

Drawing Into a Metafile

A metafile device context is a rather special beast when it comes to drawing the object at a specific location within that metafile. The object in this case needs to know both the window extents and the window origin if it has any hope of drawing it's representation in the correct context of the metafile hDC.

To accommodate metafiles, the *lprcWBounds* parameter contains the window extent and the window origin, not a real rectangle. The window origin is the point (*lprcWBounds->left*, *lprcWBounds->top*). The horizontal window extent is in *lprcWBounds->right* and the vertical window extent is in *lprcWBounds->bottom*. Objects in DLLs that implement IViewObject::Draw should "account" for these values if the caller provides them. By "account" for them I mean that you should scale or place your graphics appropriately: do not call SetWindowOrgEx or SetWindowExtEx or else you will place records into the metafile that could

affect other records that you do no own. You should only use these values to correctly place whatever GDI calls you happen to make, which, for the most part means you can ignore them altogether and just use *lprcBounds*.

When you do have to pay attention is if you are playing metafile yourself as your method for generating the presentation (like if you happen to be a metafile editor). For example, the OLE 2.0 default handler usually implements IViewObject::Draw by pulling a metafile from its internal cache and drawing that on the hDC. However, each record in its cached metafile assumes a specific origin, usually (0,0)\, so for OLE 2.0 to draw that metafile into another metafile it must play one record at a time and modify the scaling and the origin coordinates for each record such that it shows properly in the outer metafile. So *lprcWBounds* is required to be able to place one metafile within another. Only when you play a metafile in IViewObject::Draw do you need to worry about this parameter.

Aborting Long Repaints

The two other parameters to IViewObject::Draw allow an application the ability to break out of a long repaint or otherwise lengthy operation:

- *pfnContinue* (BOOL (CALLBACK *) (DWORD)): A pointer to a callback function that is called periodically during a painting process. The function returns TRUE to continue drawing or FALSE to abort the operation which causes ::Draw to return DRAW_E_ABORT.
- *dwContinue* (DWORD): An extra 32-bit value to pass as the parameter to *pfnContinue*. Typically this would be a pointer to some application-defined structure needed inside the callback function.

A typical user of a view object might implement a function that would test the status of the ESC key to determine whether to continue the painting, allowing end-users to end torturous long waits for ridiculously complicated drawings:

```
BOOL CALLBACK DrawAbort(DWORD dwContinue)
  {
  return (GetKeyState(VK_ESCAPE) < 0);
  }
```

How often the callback function is actually called depends on the implementation of the object, of course, as well as the actual means to draw the data. As a general guideline, call the function for every 16 operations where an operation is either a GDI call or playing a metafile record. If you know you are drawing more than one large bitmap, then call the function after each BitBlt or StretchBlt. If you are only transferring one bitmap or know that your operation is quite fast, you can ignore this function altogether.

Other IViewObject Member Functions

Of course, there are other member functions in IViewObject other than ::Draw, which are, on the average, used far less than ::Draw. This section will describe ::GetColorSet, ::Freeze, and ::Unfreeze. ::SetAdvise and ::GetAdvise are described in the "IViewObject and Notification" section below.

IViewObject::GetColorSet exists to allow the user of a view object to obtain the logical palette that the object would use when drawn such that the object user can try to match that palette as best it can. ::GetColorSet takes the same *dwAspect, lindex, pvAspect, ptd,* and *hicTargetDev* parameters as ::Draw. If the object uses a palette, it should fill the LOGPALETTE structure pointed to by *ppColorSet* with the colors it would use if ::Draw were called with the same parameters; otherwise it should return S_FALSE. In general, the palette here should be identical to what the object might pass to the Windows APIs SetPaletteEntries or CreatePalette.

::Freeze and ::Unfreeze (which I always thought should have been called ::Thaw) lets the object user to control whether or not the object is allowed to change it's visual representation on subsequent calls to ::Draw. ::Freeze works on one aspect at a time, that is, freezing DVASPECT_CONTENT still allows ::Draw called with DVASPECT_ICON to change the actual presentation. Calling ::Freeze returns a DWORD key which you later pass to ::Unfreeze to bring the object back from the Ice Age.

Freezing a view object can be thought of as creating a bitmap copy of the current view of the object and always using that bitmap to show the object. Underneath, the actual data might have changed, but the image does not, that is, since we're always using the snapshot bitmap to show the object, calls to ::Draw don't show any changes.

IViewObject and Notification

The final two members of IViewObject, ::SetAdvise and ::GetAdvise, work with IAdviseSink::OnViewChange independently of IAdviseSink::OnDataChange. For the most

part, ::OnViewChange should be used by an advise sink to know when an object's presentation changes as opposed to its underlying data which may, in fact, not change anything about the actual display of that data. For example, if a data object is attached to a spreadsheet, then a change in one of the cells only precipitates a view change for DVASPECT_CONTENT if that cell is visible in the full content rendering. A change to the spreadsheet does not, however, change DVASPECT_ICON because the icon, and its label, are quite unrelated to the actual data underneath them. A change to the actual spreadsheet filename, on the other hand, may change the icon aspect (if the filename is the icon's label) but not the content aspect which only shows the spreadsheet cells.

Notifications from a view object are commonly used from a compound document container that needs to know when to repaint a compound document object. Therefore containers will call IViewObject::SetAdvise with the aspect shown for that object (be it content, icon, etc.), advise flags (the same ADVF_* flags as those for IDataObject::DAdvise), and a pointer to the container's IAdviseSink interface. Whenever that aspect, and that aspect only, changes, the advise sink will be notified through ::OnViewChange, in which case the container will generally force a repaint on the object and a call to IViewObject::Draw.

The only advise flags that are actually relevant here are ADVF_PRIMEFIRST and ADVF_ONLYONCE, where the former will immediately generate an ::OnViewChange in your IAdviseSink and the latter will perform release your advise sink as soon as ::OnViewChange is called once. Of course, you can pass a zero for the flags.

You may wonder about when you have an advise established on an object through both the IDataObject and IViewObject interfaces on that object. Which IAdviseSink member will be called first? The answer is undefined as it depends completely on the object's implementation. You should therefore not try to depend on any particular ordering of the notifications.

A view object may not, of course, support any advises in which case IViewObject::SetAdvise will return NOTIFY_E_ADVISENOTSUPPORTED. Even when advising is supported, view objects only maintain one advise sink pointer at a time, that is, there is no ViewAdviseHolder like there is a DataAdviseHolder. Therefore a call to ::SetAdvise will terminate notifications for whoever previously called ::SetAdvise; a call with a NULL IAdviseSink will simply terminat the current connection. For this reason as well, you can call ::GetAdvise to retrieve the current IAdviseSink pointer to which the object is sending notifications.

Freeloading from OLE2.DLL

I want to skip ahead a little bit and show an application of the view object implementation inside OLE2.DLL, but to do that we need to look at compound documents a little. This library, besides providing all of the Ole* APIs, is also known as the 'default handler,' that is, the object handler that is always used for compound document object if a specific handler does not exist. Compound document objects are generically seen as some sort of locked box where inside that box is the object's native data; the object's server is the only agent that has the key to the box. Compound documents is all about putting these little boxes into documents and moving them around with the documents in which they exist.

The big deal is that only the server who originally created the object knows how to unlock the box to generate a picture for the object, that is, it's the only code that can draw anything to actually **show** or print in a document. Everything's cool as long as the server is around when you want to view or print that document. Well, since the little box moves around with it's container document, and since that document may move around from machine to machine, the object eventually becomes detached from it's server. So the document is left with this box of goo with nothing to show for it because there's nothing around that knows how to open the box.

For this reason, OLE 2.0 maintains one or more cached presentations for every object at the discretion of the container. That is, the container controls what is and is not cached. By default, OLE caches a metafile rendered for the screen for every object it handles (that is, for every object using OLE2.DLL as a handler). Whenever the container asks for a presentation to be cached for an object, OLE asks the object's server to render that presentation and tucks it away for later use. When the container saves the object it generates a call to the implementation of IPersistStorage::Save in the handler. If the handler is OLE2.DLL, then all presentations in the cache are also saved. When the container reloads the object, it generates a call to IPersistStorage::Load which pulls those presentations out of storage and into the cache. Furthermore, whenever the object is loaded, that is, its presentation cache has been initialized, the container can call IViewObject::Draw which tells the default handler to take the best presentation it can find in the cache and draw it to an hDC.

Why I'm bringing this up is because I wanted to make my Patron sample capable of containing ordinary bitmaps and metafiles before we go as far to contain compound document objects (we actually do this next

chapter). Many word processors and other document-oriented applications also share this capability or desire, because sometimes you have a picture lying around on disk that you want to integrate into a document, and you might not have an application that knows anything about such presentations. But when I thought about adding the feature to Patron I started feeling a tad query: I would have to write code to first of all draw metafiles and bitmaps (not too bad), but then also to save that data to a disk file and load it back in again. I don't know about you, but serializing graphics to a file and figuring out how to load them in again is a certain degree of tedium I avoid like bubonic plague.

Then I suddenly realized that in OLE2.DLL's role as the default handler, it had to somewhere contain the code to do exactly what I needed: drawing and serialization. I just had to figure out how to convince that DLL to do my work for me. It was a struggle at first, until I found two entries in the registration database that struck me:

```
{00000316-0000-0000-C0000-000000000046} = Device Independent Bitmap
   InprocServer=ole2.dll
{00000315-0000-0000-C0000-000000000046} = Metafile
   InprocServer=ole2.dll
```

What I found is that OLE2.DLL is a server for what are called "static" compound document objects, which are presentations without any known source. As a DLL server for any compound document object, OLE2.DLL is responsible for providing IDataObject, IViewObject, and IPersistStorage interfaces, among others, on these object. What came out of my experiment to see just how I could make use of all this neat functionality was the program called FREELOAD shown in Figure 6-4. This handy little application can copy or paste any metafile or bitmap from the clipboard and load or save it to a compound file with the .FRE extension. FREELOAD is a simple application built on the sample code CLASSLIB (as are Patron and Schmoo) where each document can contain one graphic. The only code relevant to our discussion here is in the DOCUMENT.CPP file shown in Listing 6-4.

Figure 6-4: The FREELOAD program with three open presentations.

DOCUMENT.CPP

```
/*
 * DOCUMENT.CPP
 *
 * Implementation of the CFreeloaderDoc derivation of CDocument.
 * We create a default handler object and use it for drawing, data
 * caching, and serialization.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */


#include "freeload.h"


CFreeloaderDoc::CFreeloaderDoc(HINSTANCE hInst)
   : CDocument(hInst)
   {
   m_pIStorage=NULL;
   m_pIUnknown=NULL;
   m_dwConn=0;
```

```
   return;
   }


CFreeloaderDoc::~CFreeloaderDoc(void)
   {
   ReleaseObject();
```

Listing 6-4: The DOCUMENT.CPP file of the FREELOAD program that handles compound files and clipboard operations with graphic presentations.

```
   if (NULL!=m_pIStorage)
      m_pIStorage->Release();

   return;
   }



void CFreeloaderDoc::ReleaseObject(void)
   {
   LPOLECACHE      pIOleCache;
   HRESULT         hr;

   if (0!=m_dwConn)
      {
      hr=m_pIUnknown->QueryInterface(IID_IOleCache
         , (LPVOID FAR *)&pIOleCache);

      if (SUCCEEDED(hr))
         {
         pIOleCache->Uncache(m_dwConn);
         pIOleCache->Release();
         }
      }

   if (NULL!=m_pIUnknown)
      m_pIUnknown->Release();

   CoFreeUnusedLibraries();

   m_dwConn=0;
   m_pIUnknown=NULL;
   return;
   }
```

```
BOOL CFreeloaderDoc::FMessageHook(HWND hWnd, UINT iMsg, WPARAM wParam
   , LPARAM lParam, LRESULT FAR *pLRes)
   {
   PAINTSTRUCT     ps;
   HDC             hDC;
   RECT            rc;
   RECTL           rcl;
   LPVIEWOBJECT    pIViewObject;
   HRESULT         hr;

   if (WM_PAINT!=iMsg)
      return FALSE;

   hDC=BeginPaint(hWnd, &ps);
   GetClientRect(hWnd, &rc);

   /*
    * To draw the object we can either QueryInterface for an IViewObject,
    * call IViewObject::Draw, and IViewObject::Release, or we can use
    * OleDraw which does exactly the same three steps, only calling
    * ::Draw with defaults.  OleDraw does exactly what is done here.
    */

   if (NULL!=m_pIUnknown)
      {
      hr=m_pIUnknown->QueryInterface(IID_IViewObject
         , (LPVOID FAR *)&pIViewObject);

      if (SUCCEEDED(hr))
         {
        #ifndef WIN32
         rcl.left  =MAKELONG(rc.left, 0);
         rcl.right =MAKELONG(rc.right, 0);
         rcl.top   =MAKELONG(rc.top, 0);
         rcl.bottom=MAKELONG(rc.bottom, 0);
        #else
         //Win32 rectangles are already LONGs.
         rcl=rc;
        #endif

         pIViewObject->Draw(DVASPECT_CONTENT, -1, NULL, NULL, 0, hDC
            , &rcl, NULL, NULL, 0);
        pIViewObject->Release();
         }
      }
```

```
   EndPaint(hWnd, &ps);

   return FALSE;
   }



UINT CFreeloaderDoc::ULoad(BOOL fChangeFile, LPSTR pszFile)
   {
   HRESULT            hr;
   CLSID              clsID;
   LPSTORAGE          pIStorage;
   LPUNKNOWN          pIUnknown;
   LPPERSISTSTORAGE   pIPersistStorage;
   DWORD              dwMode=STGM_TRANSACTED | STGM_READWRITE |
STGM_SHARE_EXCLUSIVE;

   if (NULL==pszFile)
      {
      //Create a new temp file.
      hr=StgCreateDocfile(NULL, dwMode | STGM_CREATE |
STGM_DELETEONRELEASE
         , 0, &pIStorage);

      if (FAILED(hr))
         return DOCERR_COULDNOTOPEN;

      m_pIStorage=pIStorage;

      FDirtySet(FALSE);
      Rename(NULL);
      return DOCERR_NONE;
      }

   //Attempt to open the storage.
   hr=StgOpenStorage(pszFile, NULL, dwMode, NULL, 0, &pIStorage);

   if (FAILED(hr))
      return DOCERR_COULDNOTOPEN;

   /*
    * When we previously called IPersistStorage::Save, OLE2.DLL kindly
    * placed a CLSID into the IStorage for us, either CLSID_StaticMetafile
    * or CLSID_StaticDib.  All we have to do is load this CLSID, create
    * the object for that ID (using CoCreateInstance, see ::FPaste below).
    */
```

```
    hr=ReadClassStg(pIStorage, &clsID);

    //See if we know about it.
    if (FAILED(hr) || !(IsEqualCLSID(clsID, CLSID_StaticMetafile)
        || IsEqualCLSID(clsID, CLSID_StaticDib)))
        {
        pIStorage->Release();
        return DOCERR_READFAILURE;
        }

    //Go create an object, then tell *it* to load the data.
    hr=CoCreateInstance(clsID, NULL, CLSCTX_INPROC_SERVER, IID_IUnknown
        , (LPVOID FAR *)&pIUnknown);

    if (FAILED(hr))
        {
        pIStorage->Release();
        return DOCERR_READFAILURE;
        }

    //Get IPersistStorage for the data we hold.
    pIUnknown->QueryInterface(IID_IPersistStorage, (LPVOID FAR *)&pIPersistStorage);

    hr=pIPersistStorage->Load(pIStorage);
    pIPersistStorage->Release();

    if (FAILED(hr))
        {
        pIUnknown->Release();
        pIStorage->Release();
        return DOCERR_READFAILURE;
        }

    m_pIStorage=pIStorage;
    m_pIUnknown=pIUnknown;

    Rename(pszFile);
    FDirtySet(FALSE);
    return DOCERR_NONE;
    }



UINT CFreeloaderDoc::USave(UINT uType, LPSTR pszFile)
    {
```

```
   HRESULT           hr;
   LPSTORAGE         pIStorage;
   LPPERSISTSTORAGE  pIPersistStorage;
   CLSID             clsID;

   //If we have no data object, there's nothing to save.
   if (NULL==m_pIUnknown)
      return DOCERR_WRITEFAILURE;

   //Get IPersistStorage for the data we hold.
   hr=m_pIUnknown->QueryInterface(IID_IPersistStorage, (LPVOID FAR
*)&pIPersistStorage);

   if (FAILED(hr))
      return DOCERR_WRITEFAILURE;

   //Save or Save As with the same file is just a commit.
   if (NULL==pszFile || (NULL!=pszFile && 0==lstrcmpi(pszFile, m_szFile)))
      {
      pIPersistStorage->Save(m_pIStorage, TRUE);
      m_pIStorage->Commit(STGC_ONLYIFCURRENT);

      pIPersistStorage->SaveCompleted(m_pIStorage);
      pIPersistStorage->Release();

      FDirtySet(FALSE);
      return DOCERR_NONE;
      }

   /*
    * When we're given a name, open the storage, creating it new if
    * it does not exist or overwriting the old one.  Then ::CopyTo from the
    * current to the new, ::Commit the new, then ::Release the old.
    */

   hr=StgCreateDocfile(pszFile, STGM_TRANSACTED | STGM_READWRITE
      | STGM_CREATE | STGM_SHARE_EXCLUSIVE, 0, &pIStorage);

   if (FAILED(hr))
      return DOCERR_COULDNOTOPEN;

   //Insure the image is up to date, then tell it we're changing
   pIPersistStorage->Save(m_pIStorage, TRUE);
   pIPersistStorage->HandsOffStorage();

   //Save the class, bitmap or metafile
   pIPersistStorage->GetClassID(&clsID);
```

```
    hr=WriteClassStg(m_pIStorage, clsID);

    hr=m_pIStorage->CopyTo(NULL, NULL, NULL, pIStorage);

    if (FAILED(hr))
        {
        pIPersistStorage->SaveCompleted(m_pIStorage);
        pIPersistStorage->Release();
        pIStorage->Release();
        return DOCERR_WRITEFAILURE;
        }

    pIStorage->Commit(STGC_ONLYIFCURRENT);

    /*
     * Revert changes on the original storage.  If this was a temp file,
     * it's deleted since we used STGM_DELETEONRELEASE.
     */
    m_pIStorage->Release();

    //Make this new storage current
    m_pIStorage=pIStorage;
    pIPersistStorage->SaveCompleted(m_pIStorage);
    pIPersistStorage->Release();

    Rename(pszFile);
    FDirtySet(FALSE);
    return DOCERR_NONE;
    }



BOOL CFreeloaderDoc::FClip(HWND hWndFrame, BOOL fCut)
    {
    BOOL            fRet=TRUE;
    static UINT     rgcf[3]={CF_METAFILEPICT, CF_DIB, CF_BITMAP};
    const UINT      cFormats=3;
    UINT            i;
    HGLOBAL         hMem;

    if (NULL==m_pIUnknown)
        return FALSE;

    if (!OpenClipboard(hWndFrame))
        return FALSE;
```

```
   //Clean out whatever junk is in the clipboard.
   EmptyClipboard();

   for (i=0; i < cFormats; i++)
      {
      hMem=RenderFormat(rgcf[i]);

      if (NULL!=hMem)
         {
         SetClipboardData(rgcf[i], hMem);
         fRet=TRUE;
         break;
         }
      }

   //Free clipboard ownership.
   CloseClipboard();

   //If we're cutting, clean out the cache and the object we hold.
   if (fRet && fCut)
      {
      ReleaseObject();
      InvalidateRect(m_hWnd, NULL, TRUE);
      UpdateWindow(m_hWnd);
      FDirtySet(TRUE);
      }

   return fRet;
   }



HGLOBAL CFreeloaderDoc::RenderFormat(UINT cf)
   {
   LPDATAOBJECT      pIDataObject;
   FORMATETC         fe;
   STGMEDIUM         stm;

   if (NULL==m_pIUnknown)
      return NULL;

   //We only have to ask the data object (cache) for the data.
   switch (cf)
      {
      case CF_METAFILEPICT:
         stm.tymed=TYMED_MFPICT;
         break;
```

```
    case CF_DIB:
        stm.tymed=TYMED_HGLOBAL;
        break;

    case CF_BITMAP:
        stm.tymed=TYMED_GDI;
        break;

     default:
        return NULL;
     }

  stm.hGlobal=NULL;
  SETFormatEtc(fe, cf, DVASPECT_CONTENT, NULL, stm.tymed, -1);

  m_pIUnknown->QueryInterface(IID_IDataObject, (LPVOID FAR *)&pIDataObject);
  pIDataObject->GetData(&fe, &stm);
  pIDataObject->Release();

  return stm.hGlobal;
  }



BOOL CFreeloaderDoc::FQueryPaste(void)
  {
  return IsClipboardFormatAvailable(CF_BITMAP)
     || IsClipboardFormatAvailable(CF_DIB)
     || IsClipboardFormatAvailable(CF_METAFILEPICT);
  }



BOOL CFreeloaderDoc::FPaste(HWND hWndFrame)
  {
  UINT              cf=0;
  BOOL              fRet=FALSE;
  HRESULT           hr;
  DWORD             dwConn;
  LPUNKNOWN         pIUnknown;
  LPOLECACHE        pIOleCache;
  LPPERSISTSTORAGE  pIPersistStorage;
  FORMATETC         fe;
  STGMEDIUM         stm;
  CLSID             clsID;
```

```
if (!OpenClipboard(hWndFrame))
   return FALSE;

/*
 * Try to get data in order of metafile, dib, bitmap.  We set stm.tymed
 * up front so if we actually get something a call to ReleaseStgMedium
 * will clean it up for us.
 */

stm.tymed=TYMED_MFPICT;
stm.hGlobal=GetClipboardData(CF_METAFILEPICT);

if (NULL!=stm.hGlobal)
   cf=CF_METAFILEPICT;

if (0==cf)
   {
   stm.tymed=TYMED_HGLOBAL;
   stm.hGlobal=GetClipboardData(CF_DIB);

   if (NULL!=stm.hGlobal)
      cf=CF_DIB;
   }

if (0==cf)
   {
   stm.tymed=TYMED_GDI;
   stm.hGlobal=GetClipboardData(CF_BITMAP);

   if (NULL!=stm.hGlobal)
      cf=CF_BITMAP;
   }

CloseClipboard();

//Didn't get anything?  Then we're finished.
if (0==cf)
   return FALSE;

//This now describes the data we have.
SETFormatEtc(fe, cf, DVASPECT_CONTENT, NULL, stm.tymed, -1);


/*
 * Create an object to deal with this data.  There's two ways to
 * do this:  CoCreateInstance (CoGetClassObject) or
```

```
    * OleCreateDefaultHandler.  The first will go through all the exercises
    * of looking up the CLSID in the regDB, finding OLE2.DLL (registered
    * for these classes), getting a class factory, and using
    * IClassFactory::CreateInstance.
    *
    * The second method directly into ole2.dll which creates an object
    * identical to that created with CoCreateInstnace.  This call is more
    * terse and generally faster.
    *
    * This function always uses OleCreateDefaultHandler.  For an
    * example of CoCreateInstance, see ::ULoad above.
    */

   if (CF_METAFILEPICT==cf)
      clsID=CLSID_StaticMetafile;
   else
      clsID=CLSID_StaticDib;

   hr=OleCreateDefaultHandler(clsID, NULL, IID_IUnknown, (LPVOID FAR
*)&pIUnknown);

   if (FAILED(hr))
      {
      ReleaseStgMedium(&stm);
      return FALSE;
      }

   /*
    * Our contract says we provide storage through IPersistStorage::InitNew.
    * We know that the object we're dealing with supports IPersistStorage
    * and IOleCache, so we don't bother to check return values.
    * HACK:  Living Dangerously?
    */
   pIUnknown->QueryInterface(IID_IPersistStorage, (LPVOID FAR *)&pIPersistStorage);
   pIPersistStorage->InitNew(m_pIStorage);
   pIPersistStorage->Release();

   //Now that we have the cache object, shove the data into it.
   pIUnknown->QueryInterface(IID_IOleCache, (LPVOID FAR *)&pIOleCache);
   pIOleCache->Cache(&fe, ADVF_PRIMEFIRST, &dwConn);

   hr=pIOleCache->SetData(&fe, &stm, TRUE);
   pIOleCache->Release();

   if (FAILED(hr))
      {
```

```
    ReleaseStgMedium(&stm);
    pIUnknown->Release();
    return FALSE;
    }

  //Now that that's all done, replace our current with the new.
  ReleaseObject();
  m_pIUnknown=pIUnknown;
  m_dwConn=dwConn;

  FDirtySet(TRUE);

  InvalidateRect(m_hWnd, NULL, TRUE);
  UpdateWindow(m_hWnd);
  return TRUE;
  }
```

The most interesting functions in DOCUMENT.CPP are those dealing with the clipboard: CFreeloaderDoc::FClip, ::RenderFormat, and ::FPaste. The ::ULoad and ::USave functions are written according to the IPersistStorage contract between an object and the object user, and since we've already nailed the coffin on that topic in Chapter 5 I won't resurrect it again. ::FMessageHook is only interesting in that it shows a real working example of calling IViewObject::Draw, pretty much in the simple use where OleDraw would suffice. I'm using IViewObject explicitly here for demonstration: OleDraw would be perfectly sufficient.

In all truth, ::FClip and ::RenderFormat look perfectly innocent as well. ::FClip simply places a format or two on the clipboard and ::RenderFormat simply asks the object in the document for a metafile or bitmap through IDataObject. The question is, again, how on earth did we originally obtain the pointer to the object? Our answer lies in ::FPaste.

The first half of ::FPaste looks like a reasonably normal piece of Windows code that gets a graphic image off the clipboard. The only slightly odd thing is that I'm storing the data handle directly into a STGMEDIUM (because it's there!). I also initialize a FORMATETC with the description of what data I actually pasted. Then I do something really really weird:

```
HRESULT     hr;
LPUNKNOWN   pIUnknown;
CLSID       clsID;
UINT        cf;

...

//cf is the format we got from the clipboard.
if (CF_METAFILEPICT==cf)
   clsID=CLSID_StaticMetafile;
else
   clsID=CLSID_StaticDib;

hr=OleCreateDefaultHandler(clsID, NULL, IID_IUnknown
   , (LPVOID FAR *)&pIUnknown);
```

OleCreateDefaultHandler is a special function designed to create an object within OLE2.DLL to service the given CLSID. Since OLE2.DLL is also registered as the handler for both CLSID_StaticMetafile[1] and CLSID_StaticDib, OleCreateDefaultHandler is actually equivalent to calling CoCreateInstance with CLSCTX_INPROC_HANDLER. OleCreateDefaultHandler is meant to be called from another specific application handler in order for that specific handler to delegate calls to the default handler. We'll see this

1Both these CLSIDs are defined in the standard OLEGUID.H include files but are not included in either OLE2.LIB or COMPOBJ.LIB. Therefore you must define these yourself as done in FREELOAD.H: DEFINE_OLEGUID(CLSID_StaticMetafile, 0x00000315, 0, 0); and DEFINE_OLEGUID(CLSID_StaticDib, 0x00000316, 0, 0);

API put to it's "proper" use in Chapter 11. For now, let's see how we can further talk it into doing our dirty work for us which only requires two simple steps.

First, we have to fulfill our contractual obligations to the object's IPersistStorage by calling IPersistStorage::InitNew with some storage object. When the document was first opened, CFreeloaderDoc::ULoad was called which either created or opened a root storage. This storage remains open for the lifetime of the document window, so we can just pass it to InitNew here.

All that remains now is to somehow take the data we obtained from the clipboard and get the object to remember it and maintain it for us. To do this you must stuff that data into the object's cache by using the IOleCache interface that object handlers must implement for all compound document objects:

```
HRESULT     hr;
LPOLECACHE   pIOleCache;

...

pIUnknown->QueryInterface(IID_IOleCache, (LPVOID FAR *)&pIOleCache);
pIOleCache->Cache(&fe, ADVF_PRIMEFIRST, &dwConn);

hr=pIOleCache->SetData(&fe, &stm, TRUE);
pIOleCache->Release();
```

The IOleCache interface has a number of member functions only two of which are of interest to us here. ::Cache instructs the object (that is, the handler) to hold on to renderings for a specific FORMATETC. ::SetData is our way of actually handing some data to the object which it places in the cache.

Well, that's really all there is to it! Now that we have some data in the cache, we can call IViewObject::Draw to show that data on the screen or to dump it to a printer, or we could call IDataObject::GetData to get that graphic back again ourselves. We can also call IPersistStorage::Save to serialize that data to a storage., and to reload the data from the storage, we only need to recreate the object and call IPersistStorage::Load which internally will call IOleCache::SetData for us. You can see loading in CFreeloaderDoc::ULoad which creates the object using CoCreateInstance just to show the equivalence between that and OleCreateDefaultHandler.

If you ask me, this was much simpler than the code I could have written. All compliments of OLE 2.0.

IDataObject and DDE

Way back in the early stages of this chapter we saw how the member functions of IDataObject were very similarly, in some cases identical, to functions available through other transfer protocols. As we'll see in Chapter 7, the Windows clipboard APIs are entirely subsumed by a few new OLE 2.0 APIs and IDataObject. Later chapters address the drag-drop and compound document protocols. But where's DDE in all this?

IDataObject and IAdviseSink encompass the same functionality as DDE, with the exception of DDE execute which is more than matched with OLE Automation. The table below shows the parallels between DDE messages and IDataObject member functions:

DDE	IDataObject
WM_DDE_POKE	IDataObject::SetData
WM_DDE_REQUEST	IDataObject::GetData
WM_DDE_ADVISE	IDataObject::DAdvise
WM_DDE_UNADVISE	IDataObject::DUnadvise
WM_DDE_DATA	IAdviseSink::OnDataChange

On the surface, it appears that one could write a simple mapping layer to expose a DDE conversation through IDataObject, or to generate DDE messages from events in an IDataObject and IAdviseSink. When I first began writing this book, I intended to include a DLL to do just this, coming full circle with the notion of Uniform Data Transfer by showing how you might isolate yourself from any DDE awareness. But there are some significant differences that I will only briefly outline here, and I will not attempt to offer solutions; such is the job of later OLE revisions.

The first difference is that DDE is inherently asynchronous, and DDE applications have come to depend on this fact. OLE 2.0 interface functions are inherently synchronous with the exception of IAdviseSink calls from a data object in an EXE. If you call IDataObject::DAdvise with ADVF_ONLYONCE | ADVF_PRIMEFIRST you can get the equivalent of an asynchronous IDataObject::GetData, but that's a special case only.

The second question is how to obtain an IDataObject pointer that represents the data you want. Under

DDE you initiate a conversation attempting to reach a specific service and topic, such as the service "Excel" and the topic "NASDAQ.XLS". OLE 2.0's mechanisms allow you do pretty much the same in a more roundabout fashion, that is, CoCreateInstance on Excel's CLSID asking for IPersistFile, then call IPersistFile::Load("NASDAQ.XLS"), followed by a QueryInterface on IPersistFile for IDataObject (assuming, of course, that Excel knew that this was how such a connection sequence would happen). So now, like you would have a DDE conversation handle, you have an IDataObject representing the entire NASDAQ.XLS spreadsheet.

This brings up the third difference. With a DDE conversation, you can request data specifying an item name and a clipboard format. This causes the DDE server to dynamically locate the data identified with the item name and to render that data in the requested format. With an IDataObject pointer, however, you can specify the format you want in FORMATETC, but how can you identify the item? The problem here is that IDataObject doesn't have a standard field in which the caller specifies the subset of data. The *lindex* field of FORMATETC could fill this void, but it's intended to be used for object layout negotiation and not as a general place-holder for something like an atom (which DDE uses to hold an item name).

What this all means is that there is no standard protocol to replace DDE with IDataObject. The capability is there, but standards are not. And how can you facilitate data exchange between applications without a standard? You don't, and so we'll talk no further about DDE transfers in the context of OLE 2.0. We can concentrate better on the existing standards of clipboard, drag-drop, and compound document exchanges.

Summary

A data object is really any other type of object as seen through IDataObject-colored glasses. IDataObject is a standard OLE 2.0 interface that combines all the data transfer capabilities of the existing transfer protocols in Windows: clipboard, DDE, and OLE 1.0, as well as providing the transfer functions for drag-drop operations as will be demonstrated in Chapter 8.

A significant improvement in data transfer provided in OLE 2.0 data objects is the ability to describe data not only with a clipboard format, but also with a target device, an indication of how much detail is contained in the rendering, and the storage device on which it lives. Before OLE 2.0, all data transfers had to go through global memory which was not suitable for all kinds of data. OLE 2.0 opens up transfers to use not only global memory but also storage on disk, either in a traditional file or in storage or stream objects.

Implementing a data object is the matter of implementing the IDataObject interface on top of the data set it represents. Through a pointer to IDataObject the object user can request data, set data, enumerate or query available formats, and set up an advisory connection between the data object and an advise sink object implemented by the object user with the IAdviseSink interface. This shows how the user of the data object is the implementor of an advise sink object and that the implementor of a data object is the user of an advise sink object. Object use and object implementation occur everywhere and this is the first real example of how objects are used in two-way communication.

Through IAdviseSink the user of a data object can receive notifications when specific data formats in the data object change. Advisory connections can be established with a variety of options, such as whether or not the object should send data with the notification.

Alongside data objects are view objects which again are any object as seen through the IViewObject interface. This interface allows you to ask an object to draw itself an otherwise manage visual presentations of its data. With some special tricks you can also get OLE2.DLL to maintain a view object for any given metafile or bitmap such that you can use the view object to draw those graphics, eliminating sometimes tedious code from your own application. In this case as well, you can coax OLE2.DLL to even save and load those graphics to and from a compound file.

The IDataObject interface and DDE have many parallels but some important differences which also prevents a complete replacement of DDE with data objects. Such a replacement is, however, forthcoming and sooner or later will become integrated into Windows. When that happens, the full circle of Uniform Data Transfer will close, where you can use any existing protocol, be it clipboard, DDE, drag-drop, or compound documents, to move a data object pointer between some source of data and some consumer of data, where once that pointer has been communicated, neither side needs to care how that communication took place.