

6. Data & Presentation Transfer & Caching

A central feature of creating compound documents using embedded or linked objects is the ability to obtain a *presentation* of an object that can be shown in its container when the container is opened. These presentations are *cached* locally in the container's document, since they must be obtainable without the expense of running the object's application.

OLE 1 had this capability, of course. At the time an object was created (with OleCreate(), etc.), the client specified what sort of presentation it needed from the object:

"OLESTATUS **OleCreate**(lpszProtocol, lpClient, lpszClass, lhClientDoc, lpszObjname, lplpObject, renderopt, cfFormat)
...

renderopt Specifies the client's preference for presentation data for the object. This parameter can be one of the following values:

Value	Meaning
olerender_none	The client library does not obtain any presentation data and does not draw the object.
olerender_draw	The client calls the OleDraw function, and the library obtains and manages presentation data.
olerender_format	The client calls the OleGetData function to retrieve data in a specific format. The library obtains and manages the data in the requested format, as specified by the cfFormat parameter.

cfFormat Specifies the clipboard format when the renderopt parameter is olerender_format. This clipboard format is used in a subsequent call to the OleGetData function. If this clipboard format is CF_METAFILEPICT, CF_DIB, or CF_BITMAP, the library manages the data and draws the object. The library does not support drawing for any other formats."

This scheme accommodated the ability for the object client to use the object either by drawing it as a picture or by getting data from it in a certain clipboard format. However, it suffers from a major drawback:

- Only one presentation or data cache can be maintained in an object. In particular, a container cannot request that both a cache of the screen presentation and of a printer presentation of the object be maintained. Further, the container must choose between getting a drawing presentation *or* getting a data presentation; it cannot have both.

This problem is addressed in OLE 2; the present chapter discusses the solution in detail. OLE 2 provides the capability to maintain inside an object drawing presentations for one or more specific target devices in addition to the screen. Further, OLE-maintained drawing presentations are automatically converted as appropriate as an object in a document is transported from one platform to another. For example, an object transported from Windows to the Macintosh will have metafile presentations converted to PICTs.

We begin this chapter by discussing some of the supporting types and constants used in what follows. Next, we present the interfaces IDataObject and IViewObject, the two central interfaces involved in data and presentation transfer respectively. Following this, we examine exactly how all these interfaces are pieced together and used by containers, handlers, and servers.

6.1. Supporting Types and Constants

A few types and constants play a central rôle in the supporting data and presentation transfer and caching.

6.1.1. FORMATETC

A FORMATETC is a generalization of a simple clipboard format, enhanced to encompass a target device, an "aspect," and a "storage medium;" wherever one might expect to find a clipboard format, a FORMATETC is used instead. Its primary use is in parameters passed to functions in IDataObject interface, such as IDataObject::GetData() where it is used to indicate exactly what kind of data the caller is requesting. IDataObject is discussed below.


```
typedef struct tagFORMATETC {
    CLIPFORMAT          cfFormat;   // the format in which data is conveyed.
    DVTARGETDEVICE *    ptd;        // the target device for which it has been composed.
    DWORD               dwAspect;   // disambiguates multiple uses of same format.
    LONG                lindex;     // interpreted according to dwAspect.
    DWORD               tymed;      // the kind of storage medium on which it the data is conveyed.
    } FORMATETC;
```

6.1.1.1. FORMATETC::cfFormat

The *cfFormat* member of FORMATETC indicates the particular clipboard format of interest. This is an occurrence of a concept already familiar to Windows programmers; see the Windows SDK documentation for a further discussion of clipboard formats and related functions, such as RegisterClipboardFormat().

6.1.1.2. FORMATETC::ptd

The *ptd* member indicates for which target device the data is to be composed. This member is a pointer to a DVTARGETDEVICE, which is described below. The ptd member may sometimes be NULL; NULL can be used whenever the data format is insensitive to target device or when the caller simply doesn't care what device is used. In the latter case, if the data still requires a target device, then the object should pick an appropriate default device (often the display for visual objects). Data obtained from an object with a NULL target device (especially in the former case where the data format is insensitive to the device) can be thought of as an alternate form of the native representation of the object: a representation that can be used for data interchange. The data that results is usually the same as would result if the user did a **File / Save As...** and specified an interchange format.

6.1.1.3. FORMATETC::dwAspect

The *dwAspect* member enables the caller to request data that represents different aspects, roles, or views of the object, yet are all communicated using the same clipboard format. For example, the caller might want to request an iconic picture of the object vs. its content picture, yet retrieve both in a metafile clipboard format. Values from this parameter are taken from the enumeration DVASPECT:

```
typedef enum tagDVASPECT {
    DVASPECT_CONTENT        = 1,
    DVASPECT_THUMBNAIL      = 2,
    DVASPECT_ICON           = 4,
    DVASPECT_DOCPRINT       = 8,
    } DVASPECT;
```

(Even though these values are individual flag bits, a value in FORMATETC::dwAspect must specify *exactly one value*. Use of *multiple* DVASPECT values is found in the few variables named "grfAspects," such as the parameter to IViewObject::SetAdvise(), which is discussed below. The single exception to this is found in IDataObject::DAdvise().)

The semantics of each of these values is as follows.

Value	Description
DVASPECT_CONTENT	Give a representation of the object as appropriate for display as an *embedded object* inside its container. For use with compound document objects, this is by far the most common value. Note that it would be appropriate to use this value to get a presentation of the embedded object either for rendering on the screen *or* on a printer; DVASPECT_DOCPRINT, by contrast, concerns the look of the object *as if it were printed from the top level.*
DVASPECT_THUMBNAIL	Give a thumbnail representation of the object, a picture appropriate for showing in a browsing tool. The **File / Find File ...** and **Insert / Picture ...** dialogs in Microsoft Word for Windows 2.0 are examples of such browsers.

DVASPECT_ICON	Give an iconic representation of the object. See also "Labeled-icon Metafile" below.
DVASPECT_DOCPRINT	Give a representation of the object as if it were printed, i.e.: as if **File / Print...** were chosen from its menus. The described data represents a sequence of pages.

6.1.1.4. FORMATETC::lindex

The interpretation of this field is governed by the current value of FORMATETC::dwAspect. The relationship is as follows:

Value	Description
DVASPECT_CONTENT	lindex must be -1.
DVASPECT_THUMBNAIL	lindex is ignored.
DVASPECT_ICON	lindex is ignored.
DVASPECT_DOCPRINT	The lindex parameter controls which pages of the document are referred to. A value of -1 indicates all the pages are of interest. A positive value indicates a particular page number in the document. Page 1 is the first page.

6.1.1.5. FORMATETC::tymed

The *tymed* member of FORMATETC indicates by what mechanism data is to be conveyed in a particular data transfer. In addition to being passed through global memory, the data can be passed though a disk file or through an instance of one of the OLE 2 storage-related interfaces (see the chapter on "Persistent Storage for Objects" for a detailed discussion of these interfaces). When data is actually transferred, a generalization of an HGLOBAL known as a STGMEDIUM is used to pass the data.

This member is an or'ing together of values taken from the enumeration TYMED, which have the following meanings:

```
typedef enum tagTYMED {          // Data provided on a storage medium consisting of ...
    TYMED_HGLOBAL = 1,           //     ... a global memory handle
    TYMED_FILE = 2,              //     ... a file name indicating a disk file
    TYMED_ISTREAM = 4,           //     ... a pointer to an instance of IStream interface
    TYMED_ISTORAGE = 8,          //     ... a pointer to an instance of IStorage interface
    TYMED_GDI = 16,              //     ... a GDI object; needs special release behaviour
    TYMED_MFPICT = 32,           //     ... a CF_METAFILEPICT: contains a nested global handle
    TYMED_NULL = 0,              //     (not actually a medium; indicates none is passed).
    } TYMED;
```

Value	Description
TYMED_HGLOBAL	the data is to be passed in a global memory handle, much as data is passed today using DDE.
TYMED_FILE	the data is to be passed in the contents of a file on the disk.
TYMED_ISTREAM	the data is to be passed using an instance of IStream interface. The data passed is the data available through IStream::Read() calls.
TYMED_ISTORAGE	the data is to be passed using an instance of IStorage interface. The data passed is the streams and storage objects nested beneath the IStorage.
TYMED_GDI	the data is passed in a global memory handle, but is in fact a GDI object, which requires special release behavior: DeleteObject() instead of GlobalFree().
TYMED_MFPICT	the data passed is in CF_METAFILEPICT format. It too requires special release behaviour.

The general philosophy of the use of the tymed member is as follows:

Any particular clipboard format has a natural expression as either a *flat* format or a *structured* hierarchical one. All traditional formats are the former case; examples of the latter are the OLE 2 embedded object formats (e.g.: the CF_EMBEDDEDOBJECT and CF_EMBEDSOURCE formats, as described in the "Drag

& Drop and the Clipboard" chapter). We have three types of flat media (hglobal, stream, and file) and one type of structured media (storage).

It is always appropriate to ask for a particular format on a flat or a structured medium, as appropriate for the natural expression of the format. In addition, is plausible to ask for a format whose natural expression is a structured format to be provided on a flat format: the structured-to-flat mapping is that provided by the Compound File[54] IStorage implementation on top of an ILockBytes. For example, passing a structured format in a file uses StgCreateDocfile() to build an IStorage on top of it, then fills in the IStorage as determined by the semantics of the format.

It is not appropriate to ask for a "flat" format on a structured medium. CF_TEXT, for example, cannot be passed on TYMED_ISTORAGE.

6.1.2. STGMEDIUM

In much the same way as a FORMATETC is a generalization of a clipboard format, a STGMEDIUM is a generalization of a global memory handle: wherever one might expect to find an HGLOBAL involved in a data transfer, in OLE 2 a STGMEDIUM is used in its place.

```
typedef struct tagSTGMEDIUM {
    DWORD       tymed;          // ONE value from the enumeration TYMED. Indicates currently valid union member.
    union {
        HANDLE  hGlobal;
        LPSTR   lpszFileName;
        IStream*  pstm;
        IStorage* pstg;
        };
    IUnknown *  punkForRelease;     // If non-NULL, ReleaseStgMedium() uses this to release the STGMEDIUM.
    } STGMEDIUM;

HRESULT ReleaseStgMedium(pmedium);
```

A STGMEDIUM is a tagged union whose members correspond to the enumeration TYMED: each different type of medium has a matching member of STGMEDIUM through which occurrences of that medium are passed.

A very common thing to do with a STGMEDIUM is to pass it from one body of code to another. For example, in IDataObject::GetData(), the callee can allocate a medium and return it to the caller (see below). In such situations we wish to have flexibility as to whether the receiving body of code now owns the medium (and thus can free the resources it contains at will) or whether when the receiving body of code is done with the medium it needs to inform the code that provided the medium in the first place in order that it can, for example, free up other hidden resources that are being maintained on the medium's behalf.

The provider of the medium indicates its choice of these two ownership scenarios in the value it provides in punkForRelease. A NULL value indicates the former scenario, where the receiving body of code can free the medium itself. If a non-NULL pointer is specified, then IUnknown::Release() will be invoked on it in order to free the medium.

The receiving body of code never itself examines punkForRelease; instead, it simply calls ReleaseStgMedium() to free the medium. See that function for a description of how punkForRelease is used.

6.1.2.1. ReleaseStgMedium

HRESULT ReleaseStgMedium(pmedium)

Free the given storage medium. After this call, the medium is invalid and can no longer be used.

If the original provider of the medium wished to maintain control of the freeing of the medium, the following is done according to the type of storage medium, after which pmedium->punkForRelease->Release() is invoked.

---

[54] Historically, Compound Files were called "Docfiles." The two terms are synonymous.

Medium	**Done before calling pmedium->punkForRelease->Release()**
TYMED_HGLOBAL, TYMED_MFPICT, TYMED_GDI	Nothing is done.
TYMED_FILE	Nothing is done to the actual disk file. Frees the file name string by using the standard memory management paradigm.
TYMED_ISTREAM	Calls IStream->Release().
TYMED_ISTORAGE	Calls IStorage->Release().

If by contrast ownership was transferred to the receiver of the data (which is indicated to ReleaseStgMedium() by punkForRelease being NULL), then the following is done.

Medium	**Done in order to free the data**
TYMED_HGLOBAL	Calls GlobalFree() on the handle.
TYMED_GDI	Calls DeleteObject() on the handle.
TYMED_MFPICT	The hMF that it contains is deleted with DeleteMetaFile(), then the handle itself passed to GlobalFree().
TYMED_FILE	Frees the disk file by deleting it. Frees the file name string by using the standard memory management paradigm.
TYMED_ISTREAM	Calls IStream->Release().
TYMED_ISTORAGE	Calls IStorage->Release().

The arguments to this function are as follows:

Argument	Type	Description
pmedium	STGMEDIUM *	the storage medium which is to be freed
return value	HRESULT	S_OK

6.1.3. DVTARGETDEVICE

DVTARGETDEVICE contains enough information about a Windows target device such that an HDC can be created on the device using CreateDC(). This structure is very much like the OLE 1 OLETARGETDEVICE structure, but it contains an initial size count so that it can be copied more easily. In addition, it removes some obsolete functionality having to do with "environments."

```
typedef struct tagDVTARGETDEVICE {
    DWORD  tdSize;
    WORD   tdDriverNameOffset;
    WORD   tdDeviceNameOffset;
    WORD   tdPortNameOffset;
    WORD   tdExtDevmodeOffset;
    BYTE   tdData[1];
    } DVTARGETDEVICE;
```

Member	Description
tdSize	the size of the DVTARGETDEVICE structure in bytes.
tdDeviceNameOffset	specifies the offset from the beginning of the target device structure to the name of the device.
tdDriverNameOffset	specifies the offset from the beginning of the target device structure to the name of the device driver.
tdPortNameOffset	specifies the offset from the beginning of the target device structure to the name of the port.
tdExtDevmodeOffset	specifies the offset from the beginning of the target device structure to a DEVMODE structure retrieved by the ExtDeviceMode() function.
tdData	specifies an array of bytes containing data for the target device.

The strings indirectly indicated by tdDeviceNameOffset, tdDriverNameOffset, and tdPortNameOffset should be NULL-terminated.

6.1.4. Labeled-icon Metafile

In OLE2, when an embedded object is shown as an icon in its container, the icon has a user-defined label below it. Thus, when an object is asked for an icon, it cannot return an actual windows HICON; rather, it returns a metafile that when played would draw an icon with a label below. To support this, and to support the container's ability to edit the label of the icon, we define a particular style of metafile. This is a normal metafile, but with some restrictions and interpretations on its records.

1. The initial records in the metafile can in fact be anything except the comment denoted in 2. Most typically, however, the initial records should be:
   a. SetWindowOrg(0,0) and SetWindowExt(max of text and icon width, icon+text height+4). This gets us the scaling that we need.
   b. draw the icon centered horizontally at the top of the metafile
2. Next is a metafile comment containing the string "IconOnly". This delimits the icon-drawing part of the metafile from the rest of it. Any ExtTextOut() calls following this record are in fact part of the label. The first and second comment following this one are also semantically significant. This "IconOnly" comment may be omitted if there is no label, and if no icon-source information is provided.
3. Typically now follow calls to set the text color and the background mode, and to select a font. Anything but ExtTextOut or comment records are allowed.
4. Now follow one or more consecutive ExtTextOut calls, the text of which concatenated together in order comprises the label.
5. The metafile comment next following the one in step 2 contains a string which is the full file name of the .EXE or .DLL from which this icon was extracted. No metafile comments may intervene between this one and the one in step 2.
6. The next following metafile comment contains a string which is the index of the icon used from within the file indicated in step 5. No metafile comments may intervene between this one and the one in step 5.

When, in IDataObject::GetData() or IDataObject::GetDataHere(), an object is asked for DVASPECT_ICON and metafile format, it should respond with a metafile as just described.

6.2. IDataObject interface

IDataObject interface provides the ability to pass data to and from an object using SetData() and GetData(). The data that is passed is arranged according to a particular format denoted by a clipboard format. Optionally, the data is tagged as being composed or laid-out according to the characteristics of a particular target device. The data being transferred can be conveyed by one of several different media.

The set of formats, etc., that can legally be passed to and from an object can be enumerated with EnumFormatEtc(). In addition, an advisory connection can be made to the data object whereby it will notify a caller when data it contains changes.

```
interface IDataObject : IUnknown {
    virtual    HRESULT    GetData(pformatetc, pmedium) = 0;
    virtual    HRESULT    GetDataHere(pformatetc, pmedium) = 0;
    virtual    HRESULT    QueryGetData(pformatetc) = 0;
    virtual    HRESULT    GetCanonicalFormatEtc(pformatetcIn, pformatEtcOut) = 0;
    virtual    HRESULT    SetData(pformatetc, pmedium, fRelease) = 0;
    virtual    HRESULT    EnumFormatEtc(wDirection, ppenumFormatEtc) = 0;
```

```
virtual     HRESULT     DAdvise(pformatetc, grfAdvf, pAdvSink, pdwConnection) = 0;
virtual     HRESULT     DUnadvise(dwConnection) = 0;
virtual     HRESULT     EnumDAdvise(ppenumAdvise) = 0;
};
```

6.2.0.1. IDataObject::GetData

HRESULT IDataObject::GetData(pformatetc, pmedium)

Retrieve data for a certain aspect of the object in a certain clipboard format formatted for a certain target device conveyed on a certain storage medium. The information as to what is to be retrieved and how it is to be passed is indicated in the parameter pformatetc.

pformatetc->tymed may indicate that the caller is willing to receive the data on one of several media. The callee decides if it can support one of the media requested by the caller. If it cannot, then it returns DATA_E_FORMATETC. If it can, then it returns the actual data on a medium passed back through the pmedium parameter. pmedium is an conceptually an out parameter: the STGMEDIUM structure is allocated by the caller, but filled by the callee.

The callee gets to decide who is responsible for releasing the resources maintained on behalf of the medium: itself, or the caller. The callee indicates its decision through the value it returns through function pointer pmedium->punkForRelease(), as was described above. The caller always frees the returned medium by simply calling ReleaseStgMedium() (then, of course, freeing the STGMEDIUM structure itself).

It is not presently possible to transfer ownership of a root-level IStorage from process to another, though this will be rectified in later releases. Therefore, at present, use of GetData() with TYMED_ISTORAGE requires that the callee retain ownership of the data, that is, that it use a non-NULL pUnkForRelease. Alternatively, callers are encouraged to instead use GetDataHere(), as in general it is more efficient.

Argument	Type	Description
pformatetc	FORMATETC *	the format, etc., in which the caller would like to obtain the returned data.
pmedium	STGMEDIUM *	a place in which the medium containing the returned data is communicated.
return value	HRESULT	S_OK, DATA_E_FORMATETC.

6.2.0.2. IDataObject::GetDataHere

HRESULT IDataObject::GetDataHere(pformatetc, pmedium)

This function is almost identical to IDataObject::GetData(), but the caller provides the medium instead of the callee; the callee just copies data into the medium that the caller provides. Since the caller allocates the storage medium, then (of course) it is also responsible for freeing it.

The callee must fill in the actual medium provided by the caller: in the hGlobal case, for example, the callee *cannot* allocate a new hGlobal, but must put its data in the one the caller provided. If the caller-provided medium is not large enough for the data, then STG_E_MEDIUMFULL should be returned.[55]

It will always be the case that on entry the caller sets pmedium->tymed == pformatetc->tymed; pformatetc->tymed can only indicate one medium.

Argument	Type	Description
pformatetc	FORMATETC *	the format, etc., in which the caller would like to obtain the returned data.
pmedium	STGMEDIUM *	a place in which the medium containing the returned data is communicated.
return value	HRESULT	S_OK, DATA_E_FORMATETC, STG_E_MEDIUMFULL

[55] Notice that, unlike the other media, in the HGLOBAL case, there is no direct way for the callee to indicate the number of valid bytes returned. However, since the Windows memory allocator has historically *always* rounded the size of allocated global memory blocks, data that can validly passed in an HGLOBAL must internally be self-sizing. Thus, there is no problem.

6.2.0.3. IDataObject::QueryGetData

HRESULT IDataObject::QueryGetData(pformatetc)

Answer as to whether if this FORMATETC were to be passed to IDataObject::GetData() then data would be successfully retrieved. (There is no way to directly query whether IDataObject::GetDataHere() would be successful.) pformatEtc here is as in IDataObject::GetData().

Argument	Type	Description
pformatetc	FORMATETC *	as in IDataObject::GetData().
return value	HRESULT	S_OK, S_FALSE

6.2.0.4. IDataObject::GetCanonicalFormatEtc

HRESULT IDataObject::GetCanonicalFormatEtc(pformatetcIn, pformatetcOut)

It will sometimes be the case that a given data object will return exactly the same data for more than one requested FORMATETC. This will be particularly common for target devices: it will often be the case that the data returned is insensitive to the particular target device in question. In order that callers may store data they obtain from objects more efficiently, this function provides a mechanism whereby the object can communicate to the caller which FORMATETCs produce the same output data. Conceptually, the sets of FORMATETCs for which the same data is returned partition the space of FORMATETCs into equivalence classes; GetCanonicalFormatEtc() returns a distinguished member of the equivalence class of which pformatetcIn is a member.

In this function, the tymed member of each FORMATETC is not significant and is to be ignored.

The callee should pick a canonical representative of the set of FORMATETCs equivalent to the one passed by the caller in *pformatetcIn and return that though pformatetcOut. pformatetcOut is allocated by the caller; the callee merely fills it in.

If the returned FORMATETC is different than the passed one then S_OK is returned. If the returned FORMATETC is the same as the passed one then DATA_S_SAMEFORMATETC is returned, and no value need be filled in in pformatetcOut, with the exception of pformatetcOut->ptd, which must be NULLed. Thus, the simplest implementation of this function is one that merely sets *(pformatetcOut->ptd) = NULL and returns the constant DATA_S_SAMEFORMATETC.

Argument	Type	Description
pformatetcIn	FORMATETC *	the format, etc., in which the caller would like to obtain the returned data.
pformatetcOut	FORMATETC *	The place at which the canonical equivalent to pformatetcIn is to be returned. Caller allocates, callee fills in. If DATA_S_SAMEFORMATETC is returned from the function, then pformatetcOut->ptd must be set to NULL by the callee before exiting.
return value	HRESULT	S_OK, DATA_S_SAMEFORMATETC

6.2.0.5. IDataObject::SetData

HRESULT IDataObject::SetData(pformatetc, pmedium, fRelease)

Send data in a specified format, etc., to this object. As in IDataObject::GetData(), pformatetc indicates the format, aspect, etc., on which the data is being passed. The actual data is passed through the caller-allocated pmedium parameter.[56]

The caller decides who, itself or the callee, is responsible for releasing the resources allocated on behalf of the medium. It indicates its decision in the fRelease parameter. If false, then the caller retains ownership, and the callee may only use the storage medium for the duration of the call. If true, then the callee takes ownership, and must itself free the medium when it is done with it. The callee should *not* consider itself as

[56] It will always be the case here that pformatetc->tymed == medium->tymed.

having taken ownership of the data unless it successfully consumes it (i.e.: does not return DATA_E_FORMATETC or some other error). If it does take ownership, the callee frees the medium by calling ReleaseStgMedium(); see that function for a discussion of how the medium is actually freed.

Argument	Type	Description
pformatetc	FORMATETC *	the format, etc., in which to interpret the data contained in the medium.
pmedium	STGMEDIUM *	the actual storage medium (an in-parameter only).
fRelease	BOOL	indicates who has ownership of the medium after the call completes.
return value	HRESULT	S_OK, DATA_E_FORMATETC.

6.2.0.6. IDataObject::EnumFormatEtc

HRESULT IDataObject::EnumFormatEtc(wDirection, ppenumFormatEtc)

Enumerate the forms in which data can be stored into or retrieved from this object with SetData() and GetData() respectively. This is *not* a guarantee of support: the formats, etc., which are acceptable can in general change over time, and so the information returned by the enumeration must be treated as a hint by the caller as to what can in reality be passed. In practice, it will be a very good hint, but a hint nevertheless.

wDirection indicates the operation whose legal data transfer forms the caller wishes to enumerate. It is a value taken from the enumeration DATADIR, which is defined as follows:

```
typedef enum tagDATADIR {
    DATADIR_GET = 1,
    DATADIR_SET = 2,
    } DATADIR;
```

These values have the following semantics:

Value	Description
DATADIR_GET	enumerate those forms which can be passed in GetData().
DATADIR_SET	enumerate those forms which can be passed in SetData().

ppenum indicates where the resulting enumerator should be returned. The returned enumerator is of type IEnumFORMATETC, which is defined as:

```
typedef Enum<FORMATETC> IEnumFORMATETC;     // See Chapter 4 on enumerations.
```

(See the section in Chapter 4 regarding enumerations for a description of how this template notation in fact specifies the complete IEnumFORMATETC interface.)

The FORMATETCs returned by the enumeration usually (perhaps always) indicate a NULL target device, FORMATETC::ptd. This is appropriate since, unlike the other members of FORMATETC, the target device does not participate in the object's decision as to whether it can accept (provide) the data in a SetData() call (respectively, GetData() call). Also, the FORMATETC::tymed member may often indicate that more than one kind of storage medium is acceptable.

Argument	Type	Description
wDirection	WORD	a value from the enumeration DATADIR; see above.
ppenumFormatEtc	IEnumFORMATETC**	the place at which the instantiated enumerator is to be returned.
return value	HRESULT	S_OK.

6.2.0.7. IDataObject::DAdvise

HRESULT IDataObject::DAdvise(pformatetc, grfAdvf, pAdvSink, pdwConnection)

Set up an advisory connection between the data object and an advisory sink through which the sink can be informed when data provided by the object later changes. Not all uses of IDataObject support this DAdvise() functionality; this function can return OLE_E_ADVISENOTSUPPORTED. However, specifically,

all embeddings *must* support advise capability: if IOleObject is supported on an object, then advises on the IDataObject obtained via QueryInterface() from that IOleObject must be supported.

As in GetData(), pformatetc indicates the format, etc., in which the caller wishes to receive the data. When the data change occurs in the object, the object pAdvSink should be informed of the change using IAdviseSink::OnDataChange(). The implementation of IDataAdviseHolder provided by CreateDataAdviseHolder() is extremely helpful in supporting this functionality; see below.

If an advisory connection is successfully set up, the callee returns a non-zero value through pdwConnection (if a connection fails to be established, zero is returned). The connection so established can be torn down by the caller by passing this non-zero token back to this object in IDataObject::DUnadvise().

A group of flags are passed in the grfAdvf parameter to control the advisory connection. The constants that can be or'd together and passed in this parameter are found in the enumeration ADVF:

```
typedef enum tagADVF {
    ADVF_NODATA = 1,                // supress transfer of data in notification.
    ADVF_ONLYONCE = 2,              // only notify once, then do an automatic unadvise.
    ADVF_PRIMEFIRST = 4,            // send additional initial notification.
    ADVF_DATAONSTOP = 64,           // send data before shutdown
    ADVFCACHE_NOHANDLER = 8,        // see IOleCache::Cache().
    ADVFCACHE_FORCEBUILTIN = 16,    // see IOleCache::Cache().
    ADVFCACHE_ONSAVE = 32,          // see IOleCache::Cache().
    } ADVF;
```

(The values ADVFCACHE_NOHANDLER, ADVFCACHE_FORCEBUILTIN, and ADVFCACHE_ONSAVE are not applicable here, only in the grfAdvf parameter to the IOleCache::Cache() function. See that function for a description of the semantics of these flags).

A request for an advise with a FORMATETC with the following values:

```
formatetc.cfFormat == NULL
formatetc.ptd == NULL
formatetc.dwAspect == -1L;
formatetc.lindex == -1L;
formatetc.tymed == -1L;
```

and a grfAdvf which includes ADVF_NODATA is in fact to be interpreted as generic request to be informed of *any* change in the IDataObject. (Such "wildcard" advises are the only situation in which a dwAspect with more than one aspect is allowed in a FORMATETC::dwAspect.) See also IDataAdviseHolder::SendOnDataChange().

Value	Description
ADVF_NODATA	Do not actually return the data bits on the OnDataChange(), merely indicate that the data in the specified format has changed. The caller can then at its leisure retrieve the latest value with a GetData() call. OnDataChange() is still called; however, the medium used is TYMED_NULL. At its choice, the data object may choose to provide the data anyway, which it may wish to do particularly in the case where more than one advisory connection has been made specifying the same FORMATETC.
ADVF_ONLYONCE	Automatically tear down the advisory connection after sending the first OnDataChange() to the sink. If this flag is specified, the sink will not receive two OnDataChange() calls. A non-zero *pdwConnection is still returned if the connection is established, even if this flag is given. This allows the caller to tear down the connection *before* the first OnDataChange() occurs.
ADVF_PRIMEFIRST	Send an additional initial OnDataChange() to the sink as soon as the data is available, without waiting for it to change from its present value. Notice that the combination DAdvise(..., ADVF_ONLYONCE | ADVF_PRIMEFIRST, ...) provides, in effect, an asynchronous GetData() call.
ADVF_DATAONSTOP	If used with ADVF_NODATA, then this flag indicates that in the event that the source is about to shut down and it has changed at all since the advisory connection was established, then before shutting down it should make an additional OnData-

Change() call that *actually provides the data*.[57] Sinks that specify this flag should in OnDataChange() look to see if data is in fact being passed and take it if so; they may not get another chance.

Almost all uses of ADVF_DATAONSTOP, will, in fact, also specify ADVF_NODATA. However, if ADVF_DATAONSTOP is used *without* also specifying ADVF_NODATA, then, at the data source's option, the behaviour can be one of the following:

- the data source may behave as if ADVF_DATAONSTOP had not been indicated at all.
- the data source may behave as if ADVF_NODATA had also been given in addition to ADVF_DATAONSTOP.
- the data source may choose to *only* send a notification (if something's changed) in the shutdown case (of course one carrying the data); that is, the data source can choose to omit sending notifications as data changes during editing (ADVF_PRIMEFIRST, though, would still force an *initial* notification.)

This approach allows the sink to indicate that it is willing to do a little more work in dealing with the notifications that it receives in order to allow the data source the opportunity to optimize communications costs.

The parameters to this function have the following meanings.

Argument	Type	Description
pformatetc	FORMATETC*	the format, etc., in which the occurrence of changes should be reported to the specified sink.
grfAdvf	DWORD	a group of flags from the enumeration ADVF.
pAdvSink	IAdviseSink*	the sink which should be informed of changes.
pdwConnection	DWORD*	if an advisory connection is successfully established by this call, then through here is returned a token that can be passed to DUnadvise() in order to tear the connection down.
return value	HRESULT	S_OK, DATA_E_FORMATETC, OLE_E_ADVISENOTSUPPORTED

6.2.0.8. IDataObject::DUnadvise

HRESULT IDataObject::DUnadvise(dwConnection)

Tear down an advisory connection setup previously with IDataObject::DAdvise(). The dwConnection parameter here is a non-zero value returned through pdwConnection in the DAdvise().

Argument	Type	Description
dwConnection	DWORD	a non-zero value previously returned from IDataObject::DAdvise().
return value	HRESULT	S_OK, E_FAIL.

6.2.0.9. IDataObject::EnumDAdvise

HRESULT IDataObject::EnumDAdvise(ppenumAdvise)

Enumerate the advisory connections currently found on this object. While an enumeration is in progress, the effect of registering or revoking advisory connections on what is later is enumerated is undefined. In the event that there are presently no connections on the object, NULL is returned through ppenumAdvise.

The returned enumerator is of type IEnumSTATDATA, which is defined as:

```
typedef Enum<STATDATA>    IEnumSTATDATA;  // See Chapter 4 on enumerations.
```

[57] If we didn't have a flag of this sort, by the time that the normal non-data-carrying OnDataChange() reaches the sink, the source may have completed shutting down, and so the data may not be retrievable. In short, without this flag, ADVF_NODATA would be of little use.

(See the section in Chapter 4 regarding enumerations for a description of how this template notation in fact specifies the complete IEnumSTATDATA interface.)

STATDATA is a structure defined as follows:

```
typedef struct tagSTATDATA {
    FORMATETC       formatetc;
    DWORD           grfAdvf;
    IAdviseSink*    pAdvSink;
    DWORD           dwConnection;
    } STATDATA;
```

The arguments to this function are as follows:

Argument	Type	Description
ppenumAdvise	IEnumSTATDATA*	the place at which the new enumerator should be returned. NULL is a legal return value; it indicates that there are presently not any connections.
return value	HRESULT	S_OK, OLE_E_ADVISENOTSUPPORTED.

6.3. IViewObject interface

IViewObject interface provides the ability to ask an object to provide a pictorial representation of itself by drawing on a caller-provided device context. Independently of the drawing device context, the caller can specify the target device for which it would like the object to compose the picture. The picture can thus be composed as if it were drawn on one target device but in fact actually be drawn on a device context belonging to another device. Different kinds of pictures can be produced from the object: a caller can ask for its content representation (for embedding), an iconic representation, etc. In addition, the caller can ask to be informed when in the future the picture produced by the object changes. IViewObject interface is very much like IDataObject interface, but operates in the context of drawing pictures instead of getting data.

Due to Windows architectural considerations, it is not possible to remote an instance of IViewObject between processes.

```
interface IViewObject : IUnknown {
    virtual     HRESULT     Draw(dwAspect, lindex, pvAspect, ptd, hicTargetDev, hdcDraw, lprcBounds, lprcWBounds,
                                pfnContinue, dwContinue) = 0;
    virtual     HRESULT     GetColorSet(dwAspect, lindex, pvAspect, ptd, hicTargetDev, ppColorSet) = 0;
    virtual     HRESULT     Freeze(dwAspect, lindex, pvAspect, pdwFreeze) = 0;
    virtual     HRESULT     Unfreeze(dwFreeze) = 0;
    virtual     HRESULT     SetAdvise(grfAspects, grfAdvf, pAdvSink) = 0;
    virtual     HRESULT     GetAdvise(pgrfAspects, pgrfAdvf, ppAdvSink) = 0;
    };

HRESULT OleDraw(pUnk, dwAspect, hdcDraw, lprcBounds);
```

6.3.0.1. IViewObject::Draw

HRESULT IViewObject::Draw(dwAspect, lindex, pvAspect, ptd, hicTargetDev,
hdcDraw, lprcBounds, lprcWBounds, pfnContinue, dwContinue)

Draw the indicted piece of the indicated aspect of this object on the device context hdcDraw with formatting, font selection, and other compositional decisions made as if the object were going to be drawn on the target device indicated by ptd.

hicTargetDev is an information context on the ptd target device. It may in fact be a full device context instead of a mere information context, but the callee cannot rely on that. hicTargetDev is passed by the caller for the convenience of the callee, since callees almost always need an information context for the target device anyway, and callers usually have a device context available. hicTargetDev may not be NULL.

dwAspect indicates what kind of picture of the object is being requested. Legal values for this parameter are taken from the enumeration DVASPECT, which was previously described. Only one value may be

specified from this enumeration. The relationship between the dwAspect parameter and the lindex and lprcBounds parameters is as follows:

Value	Description
DVASPECT_CONTENT	This is by far the most common value. lindex must be -1. The compositional size of the object is as determined by the layout negotiation that has taken place: IOleObject::SetExtent(), etc.
DVASPECT_ICON, DVASPECT_THUMBNAIL	lindex is ignored. If the object supports IOleObject interface,[58] then the client can suggest the compositional size of the drawing with IOleObject::SetExtent().[59] Otherwise, the compositional size of the drawing is implicitly determined by the object itself; the client has no control over it. lprcBounds specifies the rectangle on hdcDraw into which the icon should be mapped.
DVASPECT_DOCPRINT	The lindex parameter controls which pages of the document are drawn. A value of -1 indicates all the pages should be drawn; StartPage(), EndPage(), etc., should be used as usual. A positive value indicates that that particular page number should be drawn. The lprcBounds parameter, as usual, indicates where on hdcDraw the drawing should be done. The *compositional* size and other characteristics of the page should be taken from hicTargetDev; the mapping into lprcBounds causes, perhaps, scaling to occur.

Object handlers (such as the Default Handler) which implement IViewObject::Draw() by playing a metafile have to treat SetPaletteEntries metafile records in a special way due to a bug in Windows. The Windows function PlayMetaFile() sets these palette entries to the foreground, which is incorrect; they instead need to be set to the background palette; use EnumMetaFile() and act accordingly.

The arguments to this function have the following meanings:

Argument	Type	Description
dwAspect	DWORD	a value taken from the enumeration DVASPECT. See above.
lindex	DWORD	indicates which piece of the object is of interest. Its interpretation varies with dwAspect, as described above.
pvAspect	void *	a pointer through which further parameterization of what is actually drawn can be conveyed. The actual type pointed to is determined solely by dwAspect. None of the currently-defined aspects define such a type; thus, this pointer currently must always be NULL.
ptd	DVTARGETDEVICE *	the target device against which compositional decisions in the rendered picture should be made. May be NULL, in which case the picture should be rendered according to a target device that the object deems an appropriate default. Usually, this is the DISPLAY target device.
hicTargetDev	HDC	an information context on the target device indicated by ptd. May in fact be a device context, but callers cannot rely on that. Will be NULL in the case that ptd is NULL.
hdcDraw	HDC	the device context onto which the drawing should actually be done.
lprcBounds	RECTL *	as in the similarly-named parameter to OLE 1's OleDraw() call. Points to a RECTL structure which indicates the rectangle on hdcDraw on

[58] Notice that it would be reasonable to use this aspect of drawing on things other than OLE objects; thus, we don't require for this aspect that the IViewObject also support IOleObject, as is enforced in DVAPSECT_CONTENT.

[59] Some readers may find the connection between these two interfaces to be regrettable. The author would agree. Such is life in a large engineering project.

		which the object should be drawn. This parameter controls the positioning and stretching of the object.
lprcWBounds	RECTL *	conceptually similar to the like-named parameter to OLE 1's OleDraw() call. NULL unless hdcDraw is a metafile device context. If non-NULL, then lprcWBounds points to a RECTL structure defining the bounding rectangle of the metafile underneath hdcDraw. The rectangle indicated by lprcBounds is nested inside this rectangle; they are in the same coordinate space. This is a regular run-of-the-mill true rectangle, unlike in OLE 1, where this parameter was actually an (offset, extent) pair.
pfnContinue	BOOL (*)(DWORD)	a callback function that the view object should call periodically during a lengthy drawing operation to determine whether the operation should be aborted. A return of false (zero) indicates that the drawing should stop, and that the Draw() call should return E_ABORT. Generally, this should be an exported function.
dwContinue	DWORD	a value that should be passed back as the argument to pfnContinue.
return value	HRESULT	S_OK, DV_E_DVASPECT, DV_E_LINDEX, E_ABORT

6.3.0.2. OleDraw

HRESULT OleDraw(pUnk, dwAspect, hdcDraw, lprcBounds)

OleDraw() is a simple helper function that makes the common case of drawing an object easy. The given pUnk is QueryInterface()'d for IViewObject, the RECT is converted to a RECTL, then

```
pViewObj->Draw(dwAspect, -1, 0, 0, hdcDraw, &rectl, 0, 0, 0)
```

is invoked. In this function, hdcDraw may not be a metafile device context.

Argument	Type	Description
pUnk	IUnknown *	the object which is to be drawn.
dwAspect	DWORD	the aspect of the object that is to be drawn
hdcDraw	HDC	as in IViewObject::Draw(). May not be a metafile device context.
lprcBounds	RECT *	as in IViewObject::Draw().
return value	HRESULT	S_OK, E_NOINTERFACE, errors returned by IViewObject::Draw()

6.3.0.3. IViewObject::GetColorSet

HRESULT IViewObject::GetColorSet(dwAspect, lindex, pvAspect, ptd, hicTargetDev, ppColorSet)

Returns the set of colors which would be used by a call to IViewObject::Draw() with the corresponding parameters. S_FALSE is returned when the set is either empty or doesn't matter to the object.

The implementation of this function conceptually recurses on each of any nested embedded objects and returns a color set which represents the union of all the colors requested. The color sets will eventually percolate up to the top level container (the one who owns the window frame) who, by invoking this on each of its embeddings, will have available to it knowledge of all the colors needed to draw all of its embedded objects. It can use this color set in conjunction with the colors that it itself needs in order to set the overall palette.

The Default Handler implements this function by looking at the data it has on hand to draw the picture. If the drawing format is a DIB, the palette found in the DIB is used. For a regular bitmap, no color information is returned.

If a metafile is the drawing format, then a *special convention* is used: the metafile is enumerated looking for a CreatePalette() metafile record. If one is found, then it is used as the color set. Servers who rely on the Default Handler for drawing and which use metafiles for doing so (that is, most servers) should thus

provide such a CreatePalette() record when they generate their metafiles. If no CreatePalette() metafile record is found, then the implementation in the Default Handler of GetColorSet() returns S_FALSE.

Argument	Type	Description
dwAspect	DWORD	as in IViewObject::Draw().
lindex	LONG	as in IViewObject::Draw().
pvAspect	void *	as in IViewObject::Draw().
ptd	DVTARGETDEVICE *	as in IViewObject::Draw().
hicTargetDev	HDC	as in IViewObject::Draw().
pColorSet	LOGPALETTE **	the place at which to return the set of colors that would be used. NULL is returned along with S_FALSE.
return value	HRESULT	S_OK; S_FALSE, indicating that the set is empty or that the object doesn't care to give this information. OLE_E_BLANK

6.3.0.4. IViewObject::Freeze

HRESULT IViewObject::Freeze(dwAspect, lindex, pvAspect, pdwFreeze)

This function informs the object that it should not change its drawn representation until a subsequent Unfreeze() is called. Until the Unfreeze(), successive calls to Draw() with the same parameters will produce exactly the same picture. The most common use of this function is in banded printing. Being frozen does not persist across unloads and reloads of an object; it is not part of an object's persistent state. An attempt to freeze an already-frozen aspect will return VIEW_S_ALREADYFROZEN, along with the already existing dwFreeze value.

Argument	Type	Description
dwAspect	DWORD	as in Draw().
lindex	LONG	as in Draw().
pvAspect	void *	as in Draw().
pdwFreeze	DWORD *	a place to return a key which is to be passed back in Unfreeze().
return value	HRESULT	S_OK, OLE_E_BLANK, VIEW_S_ALREADYFROZEN

6.3.0.5. IViewObject::Unfreeze

HRESULT IViewObject::Unfreeze(dwFreeze)

Unfreeze a drawing that was previously frozen with IViewObject::Freeze().

Argument	Type	Description
dwFreeze	DWORD	a key previously returned from IViewObject::Freeze().
return value	HRESULT	S_OK

6.3.0.6. IViewObject::SetAdvise

HRESULT IViewObject::SetAdvise(grfAspects, grfAdvf, pAdvSink)

Set up an advisory connection between the view object and an advisory sink through which the sink can be informed when drawings provided by object later change. The caller is asking specifically to be informed about changes to the drawing that would be produced by calling Draw() with any of the one or more aspects that are passed in grfAspects; it is not possible for the caller to request to be independently informed of changes to the drawing as produced on different target devices. This function is semantically very much like IDataObject::Advise(), but for drawings instead of data.

When the indicated drawing aspects change, the object pAdvSink should informed of the change using IAdviseSink::OnViewChange(). At any one time, a given IViewObject instance can support at most one

advisory connection. An existing advisory connection can be torn down by passing a NULL pAdvSink parameter value to this function.

A group of flags are passed in the grfAdvf parameter to control the advisory connection. The constants than can be or'd together and passed in this parameter are found in the enumeration ADVF. These constants were described in IDataObject::Advise().

The arguments to this function have the following meanings.

Argument	Type	Description
grfAspects	DWORD	a group of zero, one, or more values value taken from the enumeration DVASPECT, though zero values is a no-op.
grfAdvf	DWORD	a group of flags taken from the enumeration ADVF.
pAdvSink	IAdviseSink*	the sink which should be informed of changes. Passing a NULL value destroys any existing advisory connection.
return value	HRESULT	S_OK, OLE_E_ADVISENOTSUPPORTED, DV_E_DVASPECT

6.3.0.7. IViewObject::GetAdvise

HRESULT IViewObject::GetAdvise(pgrfAspects, pgrfAdvf, ppAdvSink)

Retrieve the existing advisory connection, if any, on the object. This function simply returns the values passed to the most recent SetAdvise() call.

Argument	Type	Description
pgrfAspects	DWORD *	through here is returned the most recent SetAdvise(grfAspects, ...) parameter. May be NULL, indicating that the caller doesn't want this value returned.
pgrfAdvf	DWORD *	analogous to pgrfAspects, but with respect to the grfAdvf parameter to SetAdvise(). May be NULL.
ppAdvSink	IAdviseSink **	analogous to pgrfAspects, but with respect to the pAdvSink parameter to SetAdvise(). May be NULL.
return value	HRESULT	S_OK.

6.4. IAdviseSink interface

IAdviseSink is an interface by which asynchronous call-backs that result from a change in some advisory connection are sent to the object that originally set up the connection in the first place. A common case of this, for example, is an object changing in such a way as its embedding and linking clients should update their cached presentations of the object: an IDataObject::DAdvise() was done, which eventually results in an IAdviseSink::OnDataChange() callback on the clients.

This interface was also discussed in the previous chapter, where its member functions other than OnDataChange() and OnViewChange() were presented.

```
interface IAdviseSink : IUnknown {
    ...
    virtual    void    OnDataChange(pformatetc, pmedium) = 0;
    virtual    void    OnViewChange(dwAspect, lindex) = 0;
    };
```

6.4.0.1. IAdviseSink::OnDataChange

void IAdviseSink::OnDataChange(pformatetc, pmedium)

Report to the sink that data that it has previously requested to be informed about has now changed. The original request was made using IDataObject::DAdvise(). One OnDataChange() call is made for every Advise() connection currently extant.

The format of the data being passed is in pformatetc, this structure has the same contents as were originally passed to the IDataObject::DAdvise() call. The data itself is passed on the storage medium found in pmedium. If the flag ADVF_NODATA was used, then the medium might be empty instead of actually containing data: pmedium->tymed might be TYMED_NULL. The medium is owned by the caller of OnDataChange(), not by the sink on which the call is made. For each sink, the data it receives in OnDataChange() is only valid for the duration of the call. The sink should not hold on to or free the medium in any way.

Argument	Type	Description
pformatetc	FORMATETC*	the format, etc., on which a DAdvise() was set up.
pmedium	STGMEDIUM*	the medium on which the data is passed.

6.4.0.2. IAdviseSink::OnViewChange

void IAdviseSink::OnViewChange(grfAspects, lindex)

Report to the sink that a view on an object that it has previously requested to be informed about has now changed. The original request was made using IViewObject::SetAdvise().

If grfAspects contains exactly one aspect, then lindex indicates which specific piece of that aspect has changed. See IViewObject::Draw().

Argument	Type	Description
grfAspects	DWORD	a group of values taken from the enumeration DVASPECT.
lindex	LONG	indicates which piece of the view has changed.

6.5. Data & Presentation Transfer & Caching in OLE 2

This section discusses the use in OLE 2 of the interfaces and functions documented earlier in this chapter. It addresses this from several viewpoints: that of a embedding or linking client, that of an object handler, and that of a remote server object. Figure 71 shows the notification connections present when an OLE 2 embedded object is loaded and running (this picture shows only interfaces directly relevant to data and presentation caching; there are other interfaces on these objects, which are not shown in this figure).

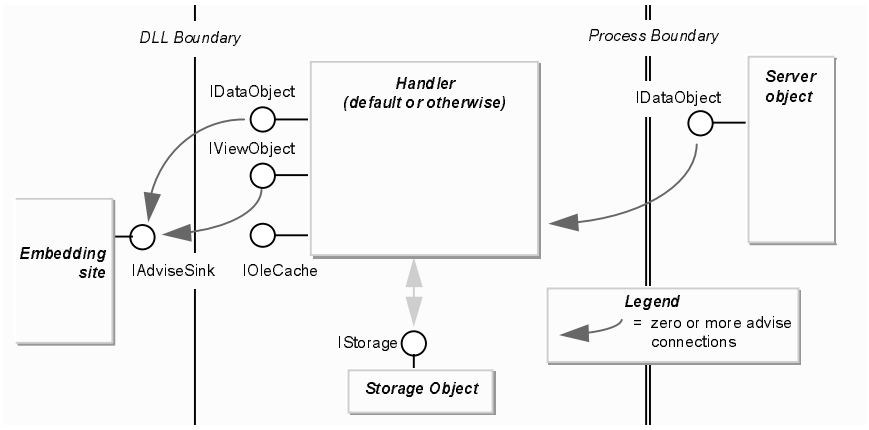

Figure 71. Data and presentation control in an OLE 2 compound document object.

The container of the object engages in a dialog with the Object Handler, asking it to get data in a certain format, to draw itself, etc. In order that this can happen, the handler presents IDataObject, IViewObject, and IOleCache interfaces for use by the client. The container makes GetData()/SetData()/Draw() calls to transfer information to and from the object, makes Advise() calls in order to be informed when things change, and uses IOleCache interface (see below) in order to let the handler know what capabilities it should cache. The IDataObject and IViewObject interface implementations operate out of data cached on the client side. The handler implementation of IViewObject is responsible for determining what *data*

format(s) to cache in order to satisfy client *draw* requests; notice that the *server* object only implements IDataObject, not IViewObject.

An internal object to the Default Handler known as the Data Cache is responsible for getting data in various formats, etc., between memory and the underlying IStorage instance of the embedded object. Custom handlers can access this implementation by aggregating in the Default Handler.

The Server Object supports IDataObject interface. At the time that the Handler connects to the Server in the process of entering the running state, it makes appropriate Advise() connections to the Server's IDataObject interface. These advisory connections are torn down by the client when the object ceases to be in the running state. The IDataObject instance on the Server Object may be connected to linking clients in addition to or instead of its embedding clients, depending on what happens to be running at a given moment.

6.6. IOleCache interface

IOleCache interface is used with OLE 2 compound document objects to provide control of what actually gets cached inside an embedded object and which will therefore be available to the object's container even when the server of the object is either not running or simply not available.

```
interface IOleCache : IUnknown {
    virtual    HRESULT    Cache(pformatetc, grfAdvf, pdwConnection) = 0;
    virtual    HRESULT    Uncache(dwConnection) = 0;
    virtual    HRESULT    EnumCache(ppenumStatData) = 0;
    virtual    HRESULT    InitCache(pDataObj) = 0;
    virtual    HRESULT    SetData(pformatetc, pmedium, fRelease) = 0;
    };
```

6.6.0.1. IOleCache::Cache

HRESULT IOleCache::Cache(pformatetc, grfAdvf, pdwConnection)

Indicate that the format, etc., found in pformatetc should be cached. This function can specify either data or view caching;[60] the latter is indicated by passing a zero clipboard format: `pformatetc->cfFormat == 0`. Through pdwConnection is returned a value with which the specified caching can be turned off with IOleCache::Uncache(). The grfAdf parameter contains a group of flags taken from the enumeration `ADVF`. This enumeration was initially presented earlier, but is repeated here for reference:

```
typedef enum tagADVF {
    // ... other flags not here applicable ...
    ADVF_NODATA = ...,
    ADVF_ONLYONCE = ...,
    ADVF_PRIMEFIRST = ...,
    ADVFCACHE_NOHANDLER = 8,
    ADVFCACHE_FORCEBUILTIN = 16,
    ADVFCACHE_ONSAVE = 32,
    } ADVF;
```

The `ADVF_PRIMEFIRST` flag indicates that the cache is to be filled immediately, not waiting for the first time that the data changes. If the object is not currently running, the cache is in fact filled the first time that the running state is entered. `ADVF_NODATA` is used in caching to indicate that the cache should not in fact be filled by changes in the server object; rather, the container will fill the cache by making explicit `IOleCache::SetData()` calls. See that function for details.

In addition, the `ADVFCACHE_` flags are applicable to IOleCache::Cache(). It is conceivable that a custom object handler might choose not to actually *store* data in a given format, but synthesize it on demand later when it is requested. Of course, this requires that the object handler in fact be present at that later time. These flags permit the container document to anticipate that it might be moved by the user to a location at which the object server application or handler might not be available and take action to ensure that the data produced by the object presentation will available nevertheless.

---

[60] with view caching, the object itself decides what formats to cache in order to handle a future request to draw.

Object handlers authors will want to look at the description of the implementation of this function in the Default Handler, which is found below; in particular, note the behaviour of that implementation when draw caching is asked for.

Value	Description
ADVFCACHE_FORCEBUILTIN	This flag forces data to be cached that only requires code shipped with OLE or the underlying operating system to be present in order to produce it with IDataObject::GetData() or IViewObject::Draw(), as appropriate. By specifying this flag, the container can ensure that the data can be retrieved even if no object server or handler code is available.
ADVFCACHE_NOHANDLER	reserved for future use. Do not specify.
ADVFCACHE_ONSAVE	Do not update the cached representation on every change of the data; rather, wait until the object containing the cache is saved to update the cache. The cache will also be updated when the OLE object transitions from the running state back to the loaded state, since otherwise a subsequent "save" would require a re-running of the object.

The arguments to this function are as follows:

Argument	Type	Description
pformatetc	FORMATETC *	the format, etc., of data that is to be cached.
grfAdvf	DWORD	a group of controlling flags. See above.
pdwConnection	DWORD *	a place to return a value which can be used to turn off the caching. May (rarely) be NULL, in which case the dwConnection can be retrieved by enumeration; see IOleCache::EnumCache().
return value	HRESULT	S_OK

6.6.0.2. IOleCache::Uncache

HRESULT IOleCache::Uncache(dwConnection)

Tear down a cache connection setup previously with IOleCache::Cache(). The dwConnection parameter here is a non-zero value returned through pdwConnection in the call that set up the caching. If this value does not actually indicate a valid connection, then OLE_E_NOCONNECTION is returned.

Argument	Type	Description
dwConnection	DWORD	a non-zero value previously returned in *pdwConnection.
return value	HRESULT	S_OK, OLE_E_NOCONNECTION.

6.6.0.3. IOleCache::EnumCache

HRESULT IOleCache::EnumCache(ppenum)

Enumerate the cache connections that are presently established.

Argument	Type	Description
ppenum	IEnumSTATDATA*	the place at which the new enumerator should be returned. NULL is a legal return value; it indicates that there are presently not any connections.
return value	HRESULT	S_OK, E_OUTOFMEMORY

6.6.0.4. IOleCache::InitCache

HRESULT IOleCache::InitCache(pDataObj)

Fill the cache as needed from data which is offered in this Clipboard Data Object. This is almost exclusively used only in the process of creating an object from the clipboard or through a drag-drop operation in the event that the format CF_EMBEDSOURCE is used to create the object. Its purpose is to fill the data cache in the embedded object from the other data formats, etc., provided on the clipboard or in the drop operation. The function OleCreateFromData() and its relatives do this automatically when appropriate. Sophisticated clients can do this themselves; they will usually want beforehand to use IOleCache::Cache() to set up the cache entries which are then filled by InitCache().

Argument	Type	Description
pDataObj	IDataObject *	the Clipboard Data Object with which the data cache may be initialized.
return value	HRESULT	S_OK

6.6.0.5. IOleCache::SetData

HRESULT IOleCache::SetData(pformatetc, pmedium, fRelease)

Sets data into the cache. This is very much like IOleCache::InitCache(), but only sets a single datum. Other than this, the sole difference between the two is whereas InitCache() will not fill up cache entries tagged with ADVFCACHE_NODATA, SetData() will in fact do so.

A container may find it convenient to use this function in order to maintain the user-set icon aspect of an object. If the container is going set up a cache for icon aspect and stuff it with an icon he provides, then it would create a cache entry by calling IOleCache::Cache() with ADVF_NODATA | ADVF_ONLYONCE | ADVF_PRIMEFIRST. Data stored by subsequent container calls to IOleCache::SetData() will be stored in the cache. It will be used to service IViewObject::Draw() calls that request icon aspect.

Argument	Type	Description
pformatetc	FORMATETC*	the format of the data being set into the cache
pmedium	STGMEDIUM*	the medium containing the data being set into the cache
fRelease	BOOL	true if the IOleCache implementation should release the medium; false if the caller will release it.
return value	HRESULT	S_OK, OLE_E_BLANK,

6.6.0.6. CreateDataCache

STDAPI CreateDataCache(pUnkOuter, rclsid, iid, ppv)

Create and return a new instance of the OLE-provided cache implementation. The returned object supports IOleCache for controlling the cache, IPersistStorage for getting its bits in and out of persistent storage, and IDataObject (sans advises). Typically, this is only used by authors of in-proc server (DLL server) objects.

Argument	Type	Description
pUnkOuter	IUnknown*	the (possibly NULL) controlling unknown of the aggregate in which the cache object is to be instantiated
clsid	REFCLSID	the class id used only to generate default icon labels; most often CLSID_NULL.
iid	REFIID	the interface id with which the caller wishes to communicate with the created cache object. Typically IID_IOleCache.
ppv	void**	the place at which the cache object is returned.
return value	HRESULT	S_OK, E_NOMEMORY

6.7. OLE-provided Implementations of Transfer and Caching Interfaces

6.7.1. Object Handler

In OLE 2, the Object Handler does many things; the discussion here concentrates on how it supports data and presentation caching. The Object Handler is either the Default Handler provided by OLE 2, or is an object-provided handler; object-provided handlers have access to the Default Handler, as so may aggregate it in, delegate to it or otherwise use it as they see fit. We shall at times point out the behaviour of the Default Handler. Object provided handlers may differ at their discretion.

The Default Object Handler presents IDataObject, IViewObject, and IOleCache interfaces. The semantics of these function implementations are as described below.

The Object Handler is responsible for converting the requirement of being able to service an IViewObject-::Draw() call into the appropriate data being cached in the Data Cache. Draw()ing is always done locally, never by remoting the call across to the Server Object. The Object Handler is also responsible for transparently doing appropriate conversion of the data cached for drawing purposes as the object is moved from platform to platform (e.g.: from Windows to the Macintosh). The Default Handler knows how to convert between the presentation formats it knows about (metafiles, bitmaps, DIBs, etc. on Windows, and PICTs on the Macintosh).

The Object Handler also owns the data cache, and ensures it is kept up to date. As the running state is entered, the Handler sets up appropriate advisory connections on the Server Object passing an IAdvise-Sink of itself as the sink. When OnDataChange() is invoked on this sink, the Handler updates the data cache, then as appropriate invokes OnDataChange() and / or OnViewChange() to the sink in its embedding site.

6.7.1.1. IDataObject::GetData

Gets the data out of the Data Cache, if it is there. If it is not there, and if the object is running and so the Server Object is available, then this function delegates to GetData() on the Server Object in order to get the data. In this latter case, the retrieved data is *not* automatically put in the cache; data is put into the cache only if an appropriate IOleCache::Cache() call has been made.

6.7.1.2. IDataObject::SetData

No effect unless the object is running. If the object is running, then this call is delegated to SetData() on the Server Object.

6.7.1.3. IDataObject::EnumFormatEtc

Does the obviously appropriate thing, according to what's in the cache and whether the object is currently running or not.

6.7.1.4. IDataObject::DAdvise

Sets up an advisory connection to the passed in advise sink. A new advisory connection must be established every time the object is loaded; the connections do not persist from session to session.

6.7.1.5. IViewObject::Draw

The object draws the requested picture using the data maintained in the Data Cache.

The Default Handler knows how to draw the `DVASPECT_CONTENT` aspect. The implementation of this basically as it is as in OLE 1, querying the Server Object first to provide a metafile, then a DIB, then a bitmap; see the description of the Default Handler's implementation of IOleCache below. Note, however, that unlike OLE 1 this is now keyed by target device: different target devices are kept separate from each other, and so multiple presentations can be maintained.

6.7.1.6. IOleCache::Cache

This function adds an item to the list of formats, etc., which are kept in the Data Cache.

The cache connections established on this instance of this interface are long lived: they persist through pacifications and reloadings of the object until explicitly broken by the client through IOleCache::-Uncache(). A consequence of this is that the dwConnection value returned from Cache() is logically part of the persistent state of the embedding container, though IOleCache::EnumCache() may be of use to it in this regard.

New cache connections can be established when the object is in *either* the loaded or the running state. When the object is loaded, the filling of the cache is deferred until the object is run. As pictured in Figure 71, when the object is in the running state, the handler uses IDataObject interface on its server in order to provide initial data for the cache when a new advisory connection is established on the handler.

When asked to do draw-caching, at the first time that the object is made running, the cache uses IDataObject::GetData() with a succession of clipboard formats in order to determine what presentation format is actually available from the running object. It tries, in order, the following clipboard formats:

```
CF_METAFILEPICT
CF_DIB
CF_BITMAP
```

Thus, servers who chose to rely on the default handler (that is, do not implement custom handlers) will want to support these formats in GetData(). Internally, the Default Handler converts `CF_BITMAP` data to `CF_DIB` data before storing it persistently.

6.7.2. Server Object

With respect to data and presentation caching, the Server Object implements just IDataObject interface. As its embedding client connects to it as the running state is entered, and as various linking clients connect to it, advisory connections are set up by the clients. These connections are transitory: they are destroyed / torn down when the running state is exited or as the linking clients are closed. The connections are reestablished afresh each time with new DAdvise() calls.

The fact that the Server Object implements IDataObject interface as its support of data and presentation transfer and caching is a consequence of the behaviour of the Default Handler. It is conceivable that some object-supplied handler might choose to communicate with the Server Object using some other interface instead of or in addition to IDataObject.

6.7.3. IDataAdviseHolder interface and implementation

IDataAdviseHolder is an interface used to communicate with an OLE-provided implementation of functionality useful to handlers and servers for remembering the set of IDataObject::DAdvise() calls they have received and sending subsequent change notifications.

```
interface IDataAdviseHolder : IUnknown {
    virtual    HRESULT    Advise(pformatetc, grfAdvf, pAdvSink, pdwConnection) = 0;
    virtual    HRESULT    Unadvise(dwConnection) = 0;
    virtual    HRESULT    EnumAdvise(ppenumAdvise) = 0;
    virtual    HRESULT    SendOnDataChange(pDataObject, dwReserved, advf) = 0;
    };

HRESULT CreateDataAdviseHolder(ppHolder);
```

The holder internally keeps track of the Advise() calls that have been made on it. These can be torn down with Unadvise() and enumerated with EnumAdvise(); see the documentation of similarly named functions in IDataObject for a detailed description of their parameters. It is expected that implementations of these three functions in a server IDataObject implementation will merely delegate calls to the corresponding function on an internal IDataAdviseHolder instance.

The holder also implements the functionality necessary to package up the array of data that needs to be passed to each IAdviseSink::OnDataChange() call along with the functionality of actually invoking On-

DataChange() on each sink. Then, when data changes in a server, it need only pass itself as a parameter to SendOnDataChange() in order to inform all the sinks currently listening on advisory connections.

6.7.3.1. IDataAdviseHolder::SendOnDataChange

HRESULT IDataAdviseHolder::SendOnDataChange(pDataObject, dwReserved, advf)

Send appropriate IAdviseSink::OnDataChange() messages to all sinks currently registered with this data-advise holder. The data that is to be passed in each OnDataChange() is obtained from the passed parameter using pDataObject->GetData() calls.

Argument	Type	Description
pDataObject	IDataObject*	the source of the data to be passed in the OnDataChange() messages; this is the object in which the data change has just occurred.
dwReserved	DWORD	reserved for future use; must be zero.
advf	DWORD	the advise flags for which a change notification in fact is to be sent. Of particular importance here is ADVF_PRIMEFIRST.
return value	HRESULT	S_OK, E_OUTOFMEMORY

6.7.3.2. CreateDataAdviseHandler

HRESULT CreateDataAdviseHolder(ppHolder)

Return a new instance of an OLE-provided implementation of IDataAdviseHolder interface.

Argument	Type	Description
ppHolder	IDataAdviseHolder**	the place at which the new instance is to be returned.
return value	HRESULT	S_OK, E_OUTOFMEMORY

6.7.4. Helper APIs

6.7.4.1. OleDuplicateData

HANDLE OleDuplicateData(hSrc, cfFormat, uiFlags)

This handy helper function merely returns a duplicate of the data found in the given handle. The following clipboard format values:

CF_BITMAP
CF_PALETTE
CF_METAFILEPICT

receive special handling; other clipboard formats are merely duplicated byte-wise.

Argument	Type	Description
hSrc	HANDLE	the data to be duplicated
cfFormat	CLIPFORMAT	the data format of the data in the handle
uiFlags	UINT	the flags with which to allocate the memory for the copied data; passed to GlobalAlloc(). If uiFlags == NULL, then a value of GMEM_MOVEABLE is used as a default.
return value	HANDLE	the duplicated data; NULL indicates error.