Notes Concerning OLE 1

OLE 2 has greatly matured since its first release in April 1993. As a result, its compatibility with OLE 1 is less an issue than it used to be. This edition of *Inside OLE* does not include information about OLE 1 in the printed text. Instead, this document contains any relevant information.

OLE 2 provides an OLE 1 compatibility layer that sits between a container and a local server. Either the server or the container can be written to OLE 1 specifications and the other to OLE 2 specifications without either knowing the difference. Essentially, OLE 2 translates the OLE 1 interfaces on one side into OLE 2 interfaces on the other. Therefore, OLE 2 containers have full access to all OLE 1 servers, and all OLE 2 servers are usable from OLE 1 containers. This allows you to upgrade an OLE 1 application to OLE 2 and know that you will not alienate OLE 1 applications.

The situation is not perfect, however, because of in-process handlers and servers. Because an OLE 2 container talks directly to a handler, it cannot use an OLE 1 handler. (This is not too unfortunate because there are very few OLE 1 handlers. Windows 3.1 Paintbrush is one of the few.) Likewise, an OLE 2 in-process handler server cannot be used by an OLE 1 container. The design differences between version 1 and version 2 simply do not allow it.

There is also a limitation with certain kinds of link sources. OLE 1 allowed links to at most a single item within a file--that is, a File!Item link. If a source provides a moniker any more complex than File or File!Item, linking by an OLE 1 container application will not be possible.

Converting object classes between OLE 2 servers is similar to converting an OLE 1 embedded object to an OLE 2 embedded object. For the most part, the process is the same as converting an OLE 2 embedded object to a different CLSID. The version of the Cosmo sample in Chapters 18 and beyond supports conversion and emulation of the OLE 1 version of Cosmo, which you can find in CHAP18\COSMO1 if you are interested.

This document also looks briefly at some defensive coding for containers such as Patron (Chapters 17 and beyond). This can prevent a few problems with respect to working with a few OLE 1 servers. In addition, this document discusses some support functions that help containers convert old files that contain OLE 1 objects to a compound file that contains OLE 2 objects.

OLE 1 Embedded Object Conversion and Emulation

Converting an OLE 1 embedded object to an OLE 2 embedded object or having an OLE 1 object emulate an OLE 2 object is merely a special case of the conversion and emulation support otherwise present in OLE 2. OLE 2 presents an OLE 1 embedded object to an OLE 2 server in a storage object through *IPersistStorage::Load*, and when the server is asked to save the embedded object in *IPersistStorage::Save*, it simply writes data back to the storage. If the server writes OLE 2 data, it performs conversion; if it writes OLE 1 data, it performs emulation. OLE 2 then takes care of sending that data to whatever container is concerned regardless of version.

To mark an OLE 2 server as capable of converting and emulating an OLE 1 server, you have to make additions to the registry under Conversion, as you do for any other server. But because OLE 1 servers had no concept of how to write clipboard formats to the object's storage, what do you store under the format? The answer is that you store the OLE 1 server's ProgID (its short class name) as the format. These are written in addition to any other formats you support for another OLE 2 server:

```
Conversion
   Readable
       Main = Cosmo1.0,Polyline Figure
   Readwritable
       Main = Cosmo1.0,Polyline Figure
```

All the work happens in *IPersistStorage*. A server must first remember that it read from a storage that contained an OLE 1 object so that it can remember to write that OLE 1 version if emulation is being used. Usually a single Boolean flag such as *m_fReadFromOLE10* is sufficient, which Cosmo stores in its *CPolyline* class. If you use C++, make this member public so that any implementation class for *IPersistStorage* can access it easily because the flag matters only for that interface. This flag is, of course, initially FALSE. It is set to TRUE only when *IPersistStorage::Load* discovers that the storage contains an OLE 1 object instead of the expected OLE 2 object:

```
STDMETHODIMP CImpIPersistStorage::Load(LPSTORAGE pIStorage)
   {
   [Variables and other initialization]

   //Try to open expected OLE 2 storage contents.
   hr=pIStorage->OpenStream("CONTENTS", 0, STGM_DIRECT
       | STGM_READWRITE | STGM_SHARE_EXCLUSIVE, 0, &pIStream);

   /*
```

```
        * Failing that, look for OLE 1 data, which OLE 2 will put
        * in a stream named "\001Ole10Native"
        */
       if (FAILED(hr))
           {
           hr=pIStorage->OpenStream("\001Ole10Native", 0, STGM_DIRECT
               | STGM_READWRITE | STGM_SHARE_EXCLUSIVE, 0, &pIStream);

           if (FAILED(hr))
               return ResultFromScode(STG_E_READFAULT);

           m_pObj->m_pPL->m_fReadFromOLE10=TRUE;
       }

       //Go read the data in whatever format we found it.
       m_pObj->m_pPL->ReadFromStream(pIStream);
       .
       .
       .
       }
```

Here *IPersistStorage::Load* initially tries to open expected OLE 2 data, and failing that, it tries to open a stream named "\001Ole10Native", which will contain the OLE 1 object data. (The \001 is an ASCII 1 preceding "Ole10Native"; it notes a special stream, as described in Chapter 7.) If *Load* finds the OLE 1 stream, it sets *m_fReadFromOLE10* to TRUE so that we can do a few chores later in *IPersistStorage::Save*.

The OLE 1 stream format contains a DWORD count of bytes at the beginning of the stream, which is then followed by the exact native data representation as created by the OLE 1 server. *CPolyline::ReadFromStream* handles this case correctly:

```
LONG CPolyline::ReadFromStream(LPSTREAM pIStream)
   {
   [Variables and other initialization]

   if (m_fReadFromOLE10)
       {
       DWORD dw;

       /*
```

```
            * Skip the DWORD length at the beginning of the
            * Ole10Native stream.
            */
           pIStream->Read((LPVOID)&dw, sizeof(DWORD), &cb);
           }

       //Read version numbers and seek back to file beginning.
       hr=pIStream->Read((LPVOID)&pl, 2*sizeof(WORD), &cb);

       //If we read OLE 1, seek back but skip the size header.
       if (m_fReadFromOLE10)
           LISet32(li, sizeof(DWORD));
       else
           LISet32(li, 0);

       pIStream->Seek(li, STREAM_SEEK_SET, NULL);

       [Read data in appropriate version and set it for editing.]

       .
       .
       .
       }
```

Because Cosmo knows exactly how long data from its earlier version is (96 bytes), we simply skip this value here. (You might, of course, be more interested in it.) In our doing so, the stream's seek pointer is positioned at the beginning of the data, which will always contain version numbers for either version 1 or version 2 data. Cosmo then simply reads those version numbers to determine how much data to read, repositions the seek pointer to the top of the data, reads it, and sets it as the active data ready for editing.

This code looks simple because Cosmo was already set to handle data from its version 1 files. In converting files, we had the version 1 file data in a stream named "CONTENTS", as created by *StgOpenStorage* with STGM_CONVERT. In converting an embedded object, we get the data from the "\001Ole10 Native" stream, which, with the exception of the DWORD header, contains exactly what a version 1 converted file would contain. You might, of course, have to do much more work to

read your version 1 embedded object data.

So now let's wrap up the story by looking at *IPersistStorage::Save* when we're dealing with an OLE 1 embedded object. There are two cases here again: conversion and emulation. If we are converting, we write our OLE 2 object data to the storage, setting the appropriate class, format, and type in the storage. (When your server is initially run to convert the OLE 1 object to your current version, you'll be asked only to load the object and generate a new presentation. Only when the object is next activated, modified, and closed will your *IPersistStorage::Save* be called, at which time you'll write your new data.)

But as described in Chapter 18, as far as conversion is concerned, anytime you write a different data structure to the storage you need to clean up any elements that are now unused. In the conversion case, the "\001Ole10Native" stream will no longer be used because the storage now contains an OLE 2 object. Cosmo takes care of this in *IPersistStorage::Save*:

```
if (m_pObj->m_pPL->m_fReadFromOLE10 && m_fConvert && (lRet >= 0))
   pIStorage->DestroyElement("\001Ole10Native");
```

This code will probably look very similar to the code for your own application. The additional condition (*lRet>=0*) ensures that we destroy only the OLE 1 element in the storage if and only if writing the OLE 2 information worked, which this extra condition indicates.

That leaves us with looking at the final case of emulating an OLE 1 embedded object. This requires a slight change to our *IPersistStorage::Save* implementation:

```
STDMETHODIMP CImpIPersistStorage::Save(LPSTORAGE pIStorage
   , BOOL fSameAsLoad)
   {
   LONG lVer=VERSIONCURRENT;

   .
   .
   .

   if (!m_fConvert && m_pObj->m_pPL->m_fReadFromOLE10)
       lVer=0x00010000;
   .
   .
   .
   [Determine stream in which to write data.]
   lRet=m_pObj->m_pPL->WriteToStream(pIStream, lVer);
   .
   .
   .
   }
```

If the *m_fConvert* flag is FALSE and we read from an OLE 1 object (which we determine by peeking into the public *m_fReadFromOLE10* flag in *CPolyline*), we know we're emulating an OLE 1 embedded object. Therefore, in *IPersistStorage::Save*, we need to write to the "\001Ole10Native" stream instead of our usual "CONTENTS" stream. If *fSameAsLoad* is TRUE within *Save* and we're emulating an OLE 1 object, we can simply write the version 1 data back to the current stream that we hold open from *Load*. Otherwise, we need to create the correct stream with the correct name, and if it is an OLE 1 stream, we must remember to write the DWORD header in the stream to indicate the size of the remaining data:

```
//In IPersistStorage::Save

if (!m_fConvert && m_pObj->m_pPL->m_fReadFromOLE10)
   {
   hr=pIStorage->CreateStream("\001Ole10Native", STGM_DIRECT
       | STGM_CREATE | STGM_WRITE | STGM_SHARE_EXCLUSIVE
       , 0, 0, &pIStream);
   }
else
```

```
{
hr=pIStorage->CreateStream("CONTENTS", STGM_DIRECT
    | STGM_CREATE | STGM_WRITE | STGM_SHARE_EXCLUSIVE
    , 0, 0, &pIStream);
}
```

After creating these streams, Cosmo simply writes its data in the appropriate version to whatever is in *pIStream*. And with this, our modification of Cosmo to handle conversion and emulation is complete.

Notes on OLE 1 Compatibility for Containers

There are a number of considerations for OLE 1 compatibility as far as containers are concerned. OLE 2 provides a compatibility layer so that OLE 2 containers can work with OLE 1 servers (linked or embedded objects) transparently, but the compatibility layer is not perfectly transparent. The first consideration is how to deal with OLE 1 servers that don't quite behave as expected. In my experience, I have come across only two major quirks in servers that can cause a container to behave erratically. The second consideration is important to you if you have an OLE 1 container that you want to convert to an OLE 2 container: how to read and write old files that contain OLE 1 objects. For this situation, we'll briefly look at two OLE 2 API functions that help you do this conversion.

OLE 1 Server Quirks

There are two behavioral oddities that an OLE 2 container might encounter in an OLE 1 server: it might encounter negative extents returned from *IOleObject::GetExtent*, and it might not receive an *IOleClientSite::OnShowWindow(FALSE)* call when the object closes.

First, many programmers (including me) are extremely confused about the use of HIMETRIC units in specifying extents of objects. As we've seen in Chapters 11 and 18, you use the *scaling* of the MM_HIMETRIC mapping mode to express extents, but you do not use the *mapping mode* itself. That is, when you convert a vertical extent to HIMETRIC units for the purpose of working with OLE, you do not negate this number as you would if you were drawing in the MM_HIMETRIC mapping mode. Therefore, all extents returned from a call such as *IOleObject::GetExtent*, and all the extents that you send to *IOleObject::SetExtent*, must be in *absolute units,* expressed in HIMETRIC. In other words, all values are positive. The confusion this caused for OLE 1 programmers resulted in a few servers specifying a negative vertical extent, so a container's call to *IOleObject::GetExtent* might have come back with a negative *y* value. Patron handles this by checking for that case and changing the extent to a positive value, as it should be, after calling *IOleObject::GetExtent*:

```
SIZEL szl;

m_pIOleObject->GetExtent(dwAspect, &szl);

if (0 > szl.cy)
   szl.cy=-szl.cy;
```

This is simply a good, defensive habit for a container to practice.

The second concern is that some OLE 1 servers will not properly generate a call to your *IOleClientSite::OnShowWindow* when the object closes. Remember that we used *OnShowWindow* to add or remove the hatching on a container site depending on the *fShow* parameter to the function. Well, for some OLE 1 servers, *OnShowWindow(FALSE*), which removes the hatching, is never sent. This means that you're stuck with a permanently hatched object. Ugly. To protect yourself, you can

include a redundancy by using *IAdviseSink::OnClose*. Patron makes the same call to *CTenant::ShowAsOpen(FALSE)* in *OnClose* as it does in *OnShowWindow*:

```
//In ICLISITE.CPP
STDMETHODIMP CImpIOleClientSite::OnShowWindow(BOOL fShow)
   {
   m_pTen->ShowAsOpen(fShow);
   return NOERROR;
   }

//Redundancy in IADVSINK.CPP
STDMETHODIMP_(void) CImpIAdviseSink::OnClose(void)
   {
   m_pTen->ShowAsOpen(FALSE);
   return;
   }
```

Again, this is an excellent defensive programming technique.

File Conversion

If you programmed an OLE 1 container and you have now seen how to implement an OLE 2 container, you will have noticed that the storage models in each are quite different. OLE 2 containers treat all compound document objects as if they lived inside a storage object. You can also control where the actual bytes end up by implementing or using an alternative lockbytes object, as described in Chapter 7. OLE 1 containers, on the other hand, treat compound document objects by means of a data structure named OLESTREAM. This structure had one field, OLESTREAMVTBL, in which the container stored the pointer to two functions, *Get* and *Put*, through which the container controlled the placement of bytes within its disk file.

So how do you reconcile the two? Most complete container applications, such as word processors and spreadsheets, will need to be able to read and write files

compatible with previous versions of that application. That means that an OLE 2 container must be able to read objects written using an OLESTREAM and must be able to write OLE 2 objects into an OLESTREAM.

OLE 2 provides two API functions to accommodate these needs: *OleConvertOLESTREAMToIStorage* and *OleConvertIStorageToOLESTREAM*. The first function is used to read an object that was written in an OLESTREAM as if it were in a storage object--that is, to enable your container to load the object simply by calling *OleLoad* as you do for anything else in a storage. The second function is used to take an object that your container knows through a storage and write it to a file through an OLESTREAM. Let's see some hypothetical code to show how this would work. (Patron actually had an OLE 1 version, but the differences between it and the OLE 2 version are simply too significant to try to make conversion work reasonably well.)

The implementation and use of a typical OLESTREAM in an OLE 1 container usually involved some variation of the OLESTREAM structure in which the container put something like a file handle so that the *Get* and *Put* functions of the stream would have access to it. When the container wanted to save or load an object, it would call *OleSaveToStream* or *OleLoadFromStream*, passing a pointer to the OLESTREAM structure. (In this context, these are OLE 1 API functions and *not* the functions of the same name in OLE 2 that take *IStream* pointers.) The OLE 1 libraries would then call *Get* and *Put* as necessary to write the object into the file:

```
//The container-specific OLESTREAM structure
typedef struct
   {
```

```
      LPOLESTREAMVTBL pvt;
      HANDLE hFile;
      } OLE1STREAM, FAR *LPOLE1STREAM;

   //The function that would write a file.
   void FileSave(LPSTR pszFile)
      {
      HANDLE hFile
      OFSTRUCT of;
      LPOLE1STREAM pStream;

      hFile=OpenFile(pszFile, &of, OF_CREATE | OF_WRITE);

      [Other error checking code, and so on]

      /*
       * This is usually some function that allocates and initializes the
       * OLE1STREAM structure and its VTBL.
       */

      pStream=AllocateOLE1STREAM();
      pStream->hFile=hFile;

      [Containers typically wrote other data here.]

      [This would be called for each object saved.]
      OleSaveToStream(pObj, (LPOLESTREAM)pStream);

      .
      .
      .

      [Cleanup]
      }

   //The function that would read a file
   void FileLoad(LPSTR pszFile)
      {
      HANDLE hFile
      OFSTRUCT of;
      LPOLE1STREAM pStream;

      hFile=OpenFile(pszFile, &of, OF_READ);

      [Other error checking code, and so on]

      pStream=AllocateOLE1STREAM();
      pStream->hFile=hFile;

      [Containers typically read other data here.]

      [This would be called for each object loaded.]
      OleLoadFromStream((LPOLESTREAM)pStream, "StdFileEditing", ...);
```

```
      .
      .
      .

      [Cleanup]
      }

   //OLESTREAM methods
   DWORD FAR PASCAL Get(LPOLE1STREAM pStream, LPBYTE pb, DWORD cb)
      {

      if (NULL==pStream->hFile)
      return 0L;

      return _hread(pStream->hFile, (void huge *)pb, cb);
      }

   DWORD FAR PASCAL Put(LPOLE1STREAM pStream, LPBYTE pb, DWORD cb)
      {
      if (NULL==pStream->hFile)
      return 0L;

      return _hwrite(pStream->hFile, (void huge *)pb, cb);
      }
```

Generally, when a container saved a file, it opened the file, wrote whatever header information was necessary, and then wrote headers for objects within that file, which would position the file's seek offset at the place at which the container wanted the object's data saved. It then called the OLE 1 function *OleSaveToStream* for each object in the document, which called *Put*, in which the container wrote the data to the file. In the same manner, the container positioned the file's seek offset at the start of an object and called the OLE 1 function *OleLoadFromStream*, which called *Get* to retrieve that data previously written. All in all, the container controlled where the data was placed in a file by positioning the seek pointer in whatever file handle was in the OLESTREAM structure before calling the OLE 1 *OleSaveToStream*.

To handle such files from an OLE 2 container, we have to modify one of these OLESTREAM instances so that it resembles a storage object. Presumably, you will

have code in your OLE 2 container that knows how to read every other part of a file generated from the OLE 1 version of your application. Eventually that code will come across a record in the file that says, “What follows is an OLE 1 compound document object”--that is, a header on an object in that file. You need to load that object and obtain some OLE 2 interface pointer for it. You can’t use an OLE 1 API function to do this because those functions are ignorant of interfaces and the only function that loads an object and returns an interface pointer is *OleLoad*. *OleLoad* requires an *IStorage* pointer from which to load the object, so we have to call *OleConvertOLESTREAMToIStorage* to get the *IStorage* pointer.

To call *OleConvertOLESTREAMToIStorage*, you have to allocate and initialize an OLESTREAM structure like the one you allocated and initialized in your OLE 1 application. Generally, you can use all the same code you had before. You then pass the OLESTREAM pointer to *OleConvertOLESTREAMToIStorage* to obtain the *IStorage* pointer, which you can then pass to *OleLoad*.

```
void ImportOldFile(LPSTR pszFile)
   {
   HANDLE hFile
   OFSTRUCT of;
   LPOLE1STREAM pStream;
   LPSTORAGE pIStorage;
   LPUNKNOWN pObj;

   hFile=OpenFile(pszFile, &of, OF_READ);

   [Other error checking code, and so on]

   pStream=AllocateOLE1STREAM();
   pStream->hFile=hFile;

   [Containers typically read other data here.]

   .
   .
   .
```

```
    [This would be called in some loop when encountering an OLE 1 object.]
    OleConvertOLESTREAMToIStorage((LPOLESTREAM)pStream, &pIStorage, NULL);
    OleLoad(pIStorage, IID_IUnknown pIOleClientSite, (LPLPVOID)&pObj);
    pIStorage->Release();

    .
    .
    .

    [Cleanup]
    }
```

For the most part, importing an OLE 1 object into an OLE 2 container is a matter of recycling all your OLE 1 code by replacing the call to *OleLoadFromStream* with calls to *OleConvertOLESTREAMToIStorage*, *OleLoad*, and, of course, *IStorage::Release*. (Don't forget the *Release*.) *OleConvertOLESTREAMToIStorage* is a function that returns a new pointer to an interface and therefore has a reference count on that pointer.

The other half of the equation is to write an OLE 2 object--one for which you have an *IStorage* pointer--into an OLE 1 file using your old OLESTREAM implementation. For this, you call *OleConvertIStorageToOLESTREAM*, which calls the OLESTREAM structure's *Put* function to write the object:

```
void ExportOldObjectToFile(HFILE hFile, LPSTORAGE pIStorage)
   {
   LPOLE1STREAM pStream;

   pStream=AllocateOLE1STREAM();
   pStream->hFile=hFile;

   OleConvertIStorageToOLESTREAM(pIStorage, (LPOLESTREAM)pStream);

   .
   .
   .
   }
```

So overall, the majority of the work in reading and writing OLE 1-compatible

files will be in re-creating your own data structures in the file, especially if your OLE 2 container is using compound files. When you hit a spot at which you have to either save or load an OLE 1 object, you need to make only a few OLE 2 calls, as shown in the previous code.