Microsoft

Information at your Fingertips

, Software Design Engineer
Systems Developer Relations

Microsoft Windows Version 3.1

30 March, 1992

Please note that this document and any associated sample code is not officially supported by Microsoft. The information provided here is intended to help software developers programming in the Microsoft Windows environment. Reader are responsible for finding and funding their own support. *Please do not ask the author for technical support as such requests will simply be referred to a support organization such as Microsoft OnLine or Compuserve.*

Updates and error lists to the document and sample code will be posted on both OnLine and Compuserve as necessary.

Your feedback is a very important part in providing documents such as this to the developer community for Microsoft Windows. At the very least, please tell me your impressions. If you can take the time, let me know how you used this document, how you used the sample code, what aspects you found helpful, and what you didn't like. A work like this document is always open to improvement, so please report any problems, errors, or general criticisms you might have. Reach me through mail, fax (dial (206)93MSFAX), or electronic mail at the following addresses:

Internet: kraigb@microsoft.com
Compuserve: 70750,2344

You deserve the best information we can provide. With your help, future documents and samples covering Microsoft Windows technologies will be even better!

Kraig Brockschmidt

Redmond, Washington USA

Object Linking and Embedding Object Streams

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

KEB.SDR.004

Table of Contents

1 What is an OLE Stream?

The Object Linking and Embedding protocol (OLE) defines a somewhat vague method for storing objects contained within a document; an OLE Stream. To OLE client applications, an object is simply a block of any number of bytes. Client applications know nothing about the object's data, but is responsible to store those objects with a document.

While many client applications generally store objects within a document file, the OLE Stream mechanism is actually much more powerful and allows the application to define exactly where it stores objects. When the user manipulates a document, they see objects contained in that document. In no way does it mean that the document *file* also has to contain those same objects.

This article will describes several different methods for using the OLE Stream mechanism in a client application: storing objects in document files, storing objects in separate files, and storing objects in a database. Objects stored in a networked database are very interesting because such shared storage means embedded objects are as efficient to store as linked objects. That is, the document file itself only contains a reference to the object in a database, much like a linked object references an object in another file. Like there is only one link file, there may only be one embedded object for any number of copies of a document.

A separate section of this article deals specifically with the implications of the **OleRelease** and **OleDelete** API within the OLE libraries. These two functions act almost identically for objects stored with a document, but widely differ when objects and documents are stored separately. Finally, the last section describes how to use the OLE Stream mechanism to save object data to memory for use in clipboard operations.

Please note that the code shown in this article does not check for all possible errors, such as failure to open files.

2 Fundamental Notes about Document Files and Objects Names

Commit this to memory: **no matter what method you use to store objects, the file for the container document must *always* contain some reference for each contained object that allows the application to locate the object**. For each object there must be some structure in the document file describing where that object lives. If the application saves objects in the document file, these references may simply be offsets within that file. If the application stores objects in a network database, the references might be a server, table name, and identifier for the object within that table.

Again, the only required function of a reference is to allow the application to locate an object. You can, of course, place additional information in that structure if it's convenient.

One possibility for extra information is an object's persistent name. The name, some text label, should remain static over time, meaning that an object labeled "BlackBox:0016" within a document should always be called "BlackBox:0016" unless explicitly renamed with the **OleRename** function. The name must not change each time the file is saved or loaded, unless you really want to go through the constant hassle of renaming objects. If you use object names as part of a user interface, your users will greatly benefit from persistent names as well, especially if the users provided those names in the first place. A user would much prefer ***"Chart from Sales for August 1991"*** instead of a cryptic MS-DOS filename such as SLSAUG91.CRT.

3 The OLESTREAM Structure: Get and Put Methods

When applications deal with saving or loading objects from some object storage, they deal with the OLESTREAM structure. More specifically, they provide two methods referenced in the stream's OLESTREAMVTBL: Get and Put. These two functions, along with your reference structure, completely define where objects reside.

OLE Client applications invariably define their own replacement for the OLESTREAM structure as they do with other OLE structures, simply because those defined in OLE.H only contain a VTBL pointer as their first field and nothing else. Defining your own replacements allows you to add more fields, as long as the VTBL remains first. When applied to the OLESTREAM structure, your application-defined replacement should contain any data that you want to pass to the Get and Put methods. This data may include a file handle, file name, and the position of the object within the document.

The Get and Put methods are called from the **OleLoadFromStream** and **OleSaveToStream** API functions, respectively. Client applications use these functions for each object in a document. Each method receives an LPOLESTREAM pointer, an LPBYTE pointer to the object data, and a DWORD with the length of the data. The LPOLESTREAM pointer is actually the exact same pointer that the client application passes as the LPOLESTREAM pointer to OleLoadFromStream and OleSaveToStream. When you define your own OLESTREAM structure, you pass a its pointer to these two OLE functions so your Get and Put methods have access to the same pointer.

Note that the Get and Put methods must always be able to handle object data greater than 64K. Windows 3.1 has two new API functions, **_hread** and **_hwrite**, that will read and write greater than 64K data blocks. The code examples in this article make use of these. In addition, there is another new function, **hmemcpy**, that handles memory operations on data greater than 64K. The section of this article that describes using streams for clipboard I/O applies this function.

4 Saving Objects in a Document File

The most common method to save objects contained in a document is to save them within the document file itself. This has the same advantage over other techniques that embedded objects have over linked objects: the entire document, including the objects, is self-contained. If the document only contained embedded object and the objects were saved in the document file, you could copy that single file to a floppy, hand it to someone else, and they would have the complete document too. In cases where you are frequently transporting files between non-networked machines, this is simply the best method to use.

For example, consider an application with the document file structure in **Figure 1**, which is the most common storage method. The application saves application specific data with OLE object references *and* data in-line. Each reference is some sort of code identifying the block as an object and contains a DWORD value equal to the size of the object data that follows it. Each object in the file contains the object's persistent name, its position in the document (however the application defines it), and the object's raw data. **Figure 2** shows a variation on this storage mechanism, where each object reference is a identification code for the object and a DWORD offset pointing to the object at the end of the file. The object data at this point is the same as described above.

µ §

µ §

In both file structures the application uses the **OleSaveToStream** function to store objects in this file and the **OleLoadFromStream** function to retrieve them. The code below shows the structures you might use to define the reference and the object in the file as well as Get and Put methods that might implement these stream mechanisms. Note that the code applies equally to both file structures above–the difference comes from where the application calls **OleSaveToStream** and **OleLoadFromStream**:

```
//A long time ago, in an include file far far away...

/*
 * Object header preceding the object data.  Note that POSITION is an application-
 * specific structure.
 */

typedef struct
   {
   POSITION          pos;          //Position of the object in the document.
   char              szName[40];   //Persistent name of object.
   DWORD             cbObject;     //Size of OLE object data.
   //Object data starts here.
   } FILEOBJECT;

/*
 * OLE Applications define their own OLESTREAM substitute structure to use
 * whenever an OLESTREAM is necessary so they can store and pass extra
 * information to the OLESTREAMVTBL methods Get and Put.
 */

typedef struct
```

```
    {
    LPOLESTREAMVTBL     pvt;          //Standard
    HANDLE              hFile;        //File handle we need in methods.
    DWORD               dwOffset;     //Offset at which we write the object data.
    FILEOBJECT          fo;           //The object header before the data.
    } STREAM;
```

The application's file I/O functions would call OleLoadFromStream to create an object from data contained in the document and OleSaveToStream to get the object data there in the first place. In the OleLoadFromStream case, the client allocates any data structure it uses to store object information while the document is in memory and initializes the STREAM structure before passing it to OleLoadFromStream which calls the Get method:

```
DWORD FAR PASCAL Get(LPSTREAM pStream, LPBYTE pb, DWORD cb)
    {
    DWORD      cbRead;

    /*
     * We have a file handle in the STREAM structure and an offset, so we
     * seek to that location and load the data.  Note that this example
     * does not preserve the file pointer.
     */

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

    _llseek(pStream->hFile, pStream->dwOffset, 0);          //Find the object
    _lread(pStream->hFile, &pStream->fo, sizeof(FILEOBJECT)); //Read the header
    cbRead=_hread(pStream->hFile, (void _huge *)pb, cb);     //Read the object

    return cbRead;
    }
```

Note that the OLECLI library will cause OleLoadFromStream/OleSaveToStream to return an error if the return value of the Get or Put methods does not match the value in the **cb** parameter.

The OleSaveToStream case is similar except the application first initializes the data in the STREAM structure with the appropriate information about the object before passing it to OleSaveToStream. Note that in the second file structure above the application would first write everything in the file except the objects themselves. The first object offset in the file is the length of the application data plus the object reference structure. Once the application saves the first object reference, it calls **OleQuerySize** to determine how large the object data really is. The offset in the file for the next object is then the first object offset plus that object's data size plus the size of the object header (like FILEOBJECT).

```
DWORD FAR PASCAL Put(LPSTREAM pStream, LPBYTE pb, DWORD cb)
    {
    DWORD      cbWritten;

    /*
     * We assume here that the application is calling OleSaveToStream after
     * writing all the application data, and so the file handle currently points
     * to the place where we write the object data.
     */

    if (NULL==pStream->hFile)
```

```
    return 0L;

_lwrite(pStream->hFile, &pStream->fo, sizeof(FILEOBJECT)); //Save the header
cbWritten=_hwrite(pStream->hFile, (void _huge *)pb, cb);   //Save the object

return cbWritten;
}
```

5 Alternate Object Storage Techniques

The example above showed how OLE client applications can save object data with a document. For applications in a non-networked environment, that technique works best simply because the document is entirely contained in one file.

However, consider situations, such as embedded systems or networked workgroups, where document portability to external machines is not an issue. Since you are no longer concerned with copying files to a floppy disk for transport, you can use other files or even a database to store objects. This section describes four different possibilities:

- **One File for a Document's Objects** All objects for a specific document are stored in a separate file associated with the document.
- **Separate Files for Each Object** Each object is stored and maintained in its own file instead of existing as part of a larger document file.
- **One File for all Objects** All objects the application generates are stored in a single and potentially very large file, somewhat like a primitive database. This file could be local or on a network.
- **All Objects in a Database** All objects are stored in a far-off database, where the notion of storage in a specific file disappears completely. The database could be local or on a network.

The discussions below describe each of these techniques with possible reference structures and Get and Put methods where appropriate. The discussions of database type storage show some hypothetical functions in the Get and Put methods since real code would be too complicated and too specific for the purposes of this article. Note that in the same FILEOBJECT header used in the first example above is used in every one of these alternate methods. Those methods involving files save the FILEOBJECT structure as a header before the object data. In the database method, that same information is stored in a database table, instead of inside a specific file. Here's FILEOBJECT again, for your convenience. Note again that POSITION is application-specific:

```
typedef struct
   {
   POSITION          pos;          //Position of the object in the document.
   char              szName[40];   //Persistent name of object.
   DWORD             cbObject;     //Size of OLE object data.
   //Object data starts here.
   } FILEOBJECT;
```

Also, for some interesting extrapolations on the possibilities for shared object storage, see the next section.

6 One File for a Document's Objects

This technique is entirely similar to the first one above that stored all the objects at the end of a document file. Instead of writing objects sequentially to the end of one file, this technique simply opens a new file just for objects and passes that file handle to the **OleLoadFromStream** and **OleSaveToStream** functions. Since we are only moving the file pointer to a specific offset and reading or writing data, the Get and Put methods are exactly the same as the first technique– the only difference is that the file handle in the Get and Put methods references the object file and not the document file and that the offsets will have different values.

7 Separate Files for Each Object

This method requires that each object is given a unique filename that in no way has to match the actual 'object name' stored in the FILEOBJECT structure. Since we only need this filename in Get and Put, the STREAM structure for this technique only needs that filename and the FILEOBJECT:

```
typedef struct
   {
   LPOLESTREAMVTBL    pvt;          //Standard
```

```
    char            szPath[256];    //File name where we store the object.
    FILEOBJECT      fo;             //The object header before the data.
    } STREAM;
```

The Get and Put methods are incredibly simple: open the file, read or write the data, and close the file.

```
DWORD FAR PASCAL Get(LPSTREAM pStream, LPBYTE pb, DWORD cb)
   {
   DWORD      cbRead;
   HANDLE     hFile;

   //You may want to more stringently check for a valid filename.
   if (0==pStream->szPath[0])
      return 0L;

   //Open the file and read the data.
   hFile=_lopen(pStream->szPath, READ | OF_SHARE_DENY_WRITE);
   _lread(hFile, &pStream->fo, sizeof(FILEOBJECT));    //Read the header
   cbRead=_hread(hFile, (void _huge *)pb, cb);         //Read the object
   _lclose(hFile);

   return cbRead;
   }


DWORD FAR PASCAL Put(LPSTREAM pStream, LPBYTE pb, DWORD cb)
   {
   DWORD      cbWritten;
   HANDLE     hFile;

   //You may want to more stringently check for a valid filename.
   if (0==pStream->szPath[0])
      return 0L;

   //Open the file and write the data.
   hFile=_lopen(pStream->szPath, WRITE | OF_SHARE_DENY_WRITE);
   _lwrite(hFile, &pStream->fo, sizeof(FILEOBJECT));    //Save the header
   cbWritten=_hwrite(hFile, (void _huge *)pb, cb);      //Save the object
   _lclose(hFile);

   return cbWritten;
   }
```

8 One File for all Objects

This technique begins to take us into the realm of databases, as a single file which stores all the objects the application has generated over time is essentially a large inefficient database. This may be ideal for an application that does not require a lot of speed and had to be implemented in a hurry.

There are still document files of course, but each object reference in the document simply provides some identifier for the object in the large data file. It could be just an offset into that file, or an index in a table in that file. Let's assume that the data file is indexed with a header of an arbitrary 16384 DWORDs (limiting the number of objects to 16384, obviously, a limitation not related to OLE in any way) where each DWORD is the object's offset from the beginning of the file. The reference in the *document* file is just a WORD index into the header table of the *data* file. Since we only need this WORD to locate the object, that WORD is all we need in the STREAM structure (with the ubiquitous FILEOBJECT, of course):

```
typedef struct
   {
   LPOLESTREAMVTBL    pvt;          //Standard
   WORD               iObject;   ;   //Object index in the data file table.
   FILEOBJECT         fo;            //The object header before the data.
   } STREAM;
```

The Get method simply opens the data file, seeks to the appropriate offset, and reads the data (the code below is specific to the example limiting the number of objects to 16384):

```
DWORD FAR PASCAL Get(LPSTREAM pStream, LPBYTE pb, DWORD cb)
   {
   DWORD      cbRead;
   HANDLE     hFile;
   DWORD      dwOffset;

   //Check if the index is out of range.
   if (16384 <= pStream->iObject)
      return 0L;

   //Open the file and read the offset at the index.
   hFile=_lopen("OBJECTS.DAT", READ | OF_SHARE_DENY_WRITE);
   _llseek(hFile, (pStream->iObject*sizeof(DWORD)), 0);
   _lread(hFile, &dwOffset, sizeof(DWORD));


   //Locate the object in the file.  Assume the offset counts the 64K index table.
   _llseek(hFile, dwOffset, 0);
   _lread(hFile, &pStream->fo, sizeof(FILEOBJECT));    //Read the header
   cbRead=_hread(hFile, (void _huge *)pb, cb);         //Read the object

   _lclose(hFile);
   return cbRead;
   }
```

The Put method is a little more complicated, because you may need to see if there is free space you can use file before expanding it. As we'll see shortly, deleting an object in the document means deleting it from storage, which can open up these blocks in the data file. Specific code to determine where to write an object is not shown in the example below:

```
DWORD FAR PASCAL Put(LPSTREAM pStream, LPBYTE pb, DWORD cb)
   {
   DWORD      cbWritten;
   HANDLE     hFile;
   DWORD      dwOffset;

   hFile=_lopen(pStream->szPath, WRITE | OF_SHARE_DENY_WRITE);
```

```
/*
 * Using whatever free space management you implement for this
 * technique, determine the next available index and the offset
 * where we can fit cb bytes.  We also want to make sure we update
 * the value at that index.
 *
 * Storing the index immediately in pStream->iObject insure that
 * the application knows the index on return from this method.
 */

pStream->iObject=INextIndex(hFIle);                  //Hypothetical
dwOffset=DwNextOffset(hFile, cb);                    //Hypothetical
StoreOffsetAtIndex(hFile, pStream->iObject, dwOffset);  //Hypothetical

//Scan to the right place in the file and write the data.
_llseek(hFile, dwOffset, 0);
_lwrite(hFile, &pStream->fo, sizeof(FILEOBJECT));    //Save the header
cbWritten=_hwrite(hFile, (void _huge *)pb, cb);      //Save the object
_lclose(hFile);

return cbWritten;
}
```

A Put method like this generates the actual index at which the object was stored instead of using any index provided in the STREAM structure. In such as case, the client application should save the object in this data file *before* saving the reference in the document file, since that reference requires the value of the index. This simply means calling **OleSaveToStream** before writing the reference in the document file.

9 All Objects in a Network Database

Along the same lines as storing all objects in a single file, you could store all object in a database table. The idea is identical–the reference stored in the document file is simply an index into a database table. The implementation differs greatly, however, from storing objects in separately managed file. Instead of just opening a file performing file operations, you will probably now use a database language to perform the actions and will probably also need other code in the application to log on to the database server, open and restore connections, and log off.

The examples below assumes that an HDB is a handle to some database connection, like a **dbproc** in Microsoft's SQL Server. Since the application maintains this handle, you will probably need to pass it to the Get and Put methods in your STREAM along with an index to the object:

```
typedef struct
   {
   LPOLESTREAMVTBL     pvt;          //Standard
   HDB                 hDB;          //Open database handle.
   DWORD               iObject;   ;   //Object index in the data file table.
   FILEOBJECT          fo;           //The object header before the data.
   } STREAM;
```

Of course, the reference you save in the document file may contain the database server name (if you allow using different databases) or may only contain the index (if you always use the same database). You might imagine what sort of trouble you can get into if you had a document generated on one machine using objects from one database and you opened that document on another machine connected to a different database with different objects at the same indices. This is another reason for the persistent name that we've been storing in the szName field of the FILEOBJECT structure. In retrieving the object from the database, you could verify that it has the correct object name. This would require you to store that name in the document file as part of the reference as well as with the object itself.

So assuming that you have a database connection and an index, the Get method sends the appropriate commands to the database. The code below assumes that any object index is valid:

```
DWORD FAR PASCAL Get(LPSTREAM pStream, LPBYTE pb, DWORD cb)
   {
   DWORD      cbRead;

   //Check if we have a valid database handle.
   if (NULL==hDB)
      return 0L;

   //Go out and query for the header and the object.  Hypothetical functions.
   FReadAnObjectHeader(hDB, pStream->iObject, &pStream->fo, sizeof(FILEOBJECT));
   cbRead=DwReadAnObject(hDB, pStream->iObject, (void _huge *)pb, cb);

   return cbRead;
   }
```

The simple Put method can let the server take care of any type of garbage collection or free space management:

```
DWORD FAR PASCAL Put(LPSTREAM pStream, LPBYTE pb, DWORD cb)
   {
   DWORD      cbWritten;
   DWORD      iObject;

   //Check if we have a valid database handle.
   if (NULL==hDB)
      return 0L;

   //Go forth and save the object.  Hypothetical functions.
   pStream->iObject=ICreateAnObjectHeader(hDB, &pStream->fo, sizeof(FILEOBJECT));
   cbWritten=DwWriteAnObject(hDC, pStream->iObject, (void _huge *)pb, cb);

   return cbWritten;
   }
```

In the same manner as the previous technique, this Put method generates the index that the application would place in the document file reference, and again, the application would call **OleSaveToStream** first *before* saving the reference in order to get the index.

10 Implications of Shared Object Files or Object Databases

OLE Objects can be linked or embedded. A great advantage to linked objects is that they are only stored once and that everyone who links to it is referring to exactly the same data. If a linked object is shared across a network, then literally hundreds of users could view (and possibly manipulate if they had the right access) that same object.

In a networked environment, the storage techniques above can give all the advantages of linked objects to embedded objects. Instead of the link being contained inside the object itself, the link is the application-defined reference saved in the document file. That link could simply be a network path to an object file or it could be a server.database.table.index structure for a database.

So for all intents and purposes, you can use shared storage to implement links to embedded objects, which may be ideal for a networked system where each workstation has limited storage resources but the server has gigabytes. You could then apply all the link tracking mechanisms for normal OLE linked objects to these shared embedded objects, including updating, changing update options, canceling, and changing the link to a different embedded object.

With some database engines, you could even implement 'hot links' or automatic updates. If multiple workstations have the same document open with the same objects, then they could probably all be connected to the database server as well. If one user changes an object and updates it in the database, the server could easily send notifications to every other workstation that the object changed. In the same way that a client application updates a linked object (when the client's CallBack method receives OLE_CHANGED) it could update this embedded object by asking the *database*, not the OLE libraries, for the new data.

Imagine that you have a large object store available; you could then go as far as implementing an object browser for your system that would allow users to search through the available objects in the shared storage. Instead of having users browse disk directories looking for cryptic MS-DOS 8.3 filenames (such as SWR32792.XLS), you could let them browse structured trees of objects where each object has a full descriptive name, or an icon, or a multimedia presentation to identify the object in its storage. You are then pretty much layering the same OLE structure on top of a collection of OLE objects. Piecing together a report could then be as simple as finding the icon in a browser and dragging it into the document application...

Information at your Fingertips*, today, with OLE 1.0.*

11 Object Store and OleRelease vs. OleDelete

For any storage technique where the object data is not an integral part of a document file, the difference between the **OleRelease** and **OleDelete** API functions are much more pronounced. When objects are stored as part of a document file, an application generally lets the user manipulate the document separately from saving changing. When the document is saved, any objects that were deleted are no longer a part of the document, so overwriting the document file essentially deletes them. Any new objects created are handled automatically in the overwrite as well. So in this case, where document storage and object storage happen together, a call to OleDelete does not necessarily mean deleting the object from the file. As a result, the difference between OleDelete and OleRelease is minimal–neither affect storage.

When you store object disjoint from the document and perhaps have a different model for saving and opening files, a call to OleDelete may literally mean delete it from storage (the delete may only happen after you decide to clear an Undo buffer or remove the ability to restore the file). In this case, the difference between OleRelease and OleDelete is far more pronounced. OleRelease means that the object no longer needs to be in memory on the local machine. Perhaps the object scrolled out of view; perhaps the object was archived and is no longer immediately available as an OLE object. All these cases where you use OleRelease deal with the object's *visibility*, not the object's *availability*.

OleDelete, on the other hand, deals only with the object's *availability*, which of course affects the objects visibility as well. OleDelete means that the object is no longer **anywhere**, in memory or in storage. Deleting the object from a database could mean just marking it for the trash can and physically deleting it later when the document is saved.

The potential hazards of OleDelete might mean a whole system of administration on shared storage, making sure that only a few people that have read-write access to the object store can physically delete objects. Other users may delete them from their document but not from the storage. It's something to be aware of and to spend a good deal of time thinking about if you might implement one of these storage techniques.

12 OLE Streams and Memory

To conclude the topic of OLE streams, there is one other possibility with the Get and Put methods that has nothing to do with permanent storage. The OLE libraries simply provide the Get and Put methods with a pointer to the data (or where to put the data). You do not necessarily have to copy them anywhere, or where you do copy them to is another memory block.

In fact, if your client application can cut and paste data that might *contain* an OLE object in itself, then it essentially implements a stream manipulation technique that deals only with memory in exactly the same fashion as the first storage method described before in this article. In short, the data you manipulate through the clipboard is simply a memory image of your file format or some similar structure. Instead of opening a disk file, you simply allocate a memory block; instead of calling **_hread** or **_hwrite** you call **hmemcpy**.

Pasting data uses the **OleLoadFromStream** function like reading from a file, only the data is in memory instead of a file and you get a pointer to the data with **GlobalLock** instead of something like **OpenFile** or **_lopen**. You're STREAM structure looks only slightly different:

```
typedef struct
   {
   LPOLESTREAMVTBL     pvt;          //Standard
   BYTE _huge *        hpData;       //Pointer to the object's memory.
   MEMOBJECT           mo;           //The object header before the data.
   } STREAM;
```

The MEMOBJECT in this structure is a placeholder for whatever header is in a clipboard data that could contain OLE objects, be it RTF, BIFF, SYLK, etc. These clipboard formats will have standard headers for OLE objects as opposed to your document files that are free to use any header they want.
OleLoadFromStream calls whatever Get method is referenced through the pvt field of the STREAM structure. This Get method is even simpler than any of the file storage methods, since all we need is a simple **hmemcpy**, and there's very little that can go wrong:

```
DWORD FAR PASCAL Get(LPSTREAM pStream, LPBYTE pb, DWORD cb)
   {
   if (NULL==pStream->hpData)
      return 0L;

   /*
   hmemcpy(&pStream->mo, pStream->hData, sizeof(MEMOBJECT));    //Read the header

   //Remember to advance the pointer--this is not automatic like file reads.
   pStream->hpData+=sizeof(MEMOBJECT);
   hmemcpy(pb, pStream->hpData, cb);

   //Point hpData past the OLE object now.
   pStream->hpData+=cb;
   return cb;
   }
```

When copying data to the clipboard, you will always need to call **OleQuerySize** for each object in the data before allocating memory to know how much to allocate. You then copy the application-specific data as necessary, and when you encounter an OLE object, put the memory pointer in your STREAM structure and call OleSaveToStream to write whatever header structure is necessary (like the hypothetical MEMOBJECT structure) and copy the data:

```
DWORD FAR PASCAL Put(LPSTREAM pStream, LPBYTE pb, DWORD cb)
   {
```

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

    hmemcpy(pStream->hpData, &pStream->mo, sizeof(MEMOBJECT));

    //Remember to advance the pointer--this is not automatic like file reads.
    pStream->hpData+=sizeof(MEMOBJECT);
    hmemcpy(pStream->hFile, (void _huge *)pb, cb);

    //Point hpData past the OLE object now.
    pStream->hpData+=cb;
    return cbWritten;
    }
```

Both these examples incremented the hpData pointer in the STREAM past the object's data in order to provide the caller with an updated pointer to the next set of data.