Chapter Five
Structured Storage and Compound Files

Seems like every time you turn around these days there's another set of APIs to read and write files, such as the addition of memory-mapped file I/O under Windows NT. Every mechanism available is concerned only with the functions you can use to create any sort of arbitrary structure within a disk file. In other words, all of the available file I/O techniques are completely ignorant of the file structure as the functions blindly obey their masters and read or write data as they are told. What is missing from the whole picture is any sort of standardization on exactly how data is structured within any particular file regardless of the underlying file system. This standardization is specified in OLE 2.0 as the Structured Storage Model. The implementation of this standard is provides in OLE 2.0 as Compound Files. Both will become incorporated into future versions of the Windows operating system.

Structured Storage describes how to create a 'file system within a file' and how to expose that structure through two types of objects called storages and streams. A storage object within the file acts like a file system directory and holds other storage and stream objects. A stream object within the file acts like a file system file that contains any data you desire. Storage and stream objects greatly simplify storage of complex information within a single file on a physical storage device and inherently support powerful and very desirable features like incremental saves. In addition, Structured Storage specifies standard but flexible data structures called property sets which are used to standardize the data stored inside specific streams within a specific file.

What? Microsoft is asking me to change my file format? Are they drunk or something? Such was a common response I heard when Microsoft first presented the idea of Structured Storage that defines a standardized technique for structuring blocks of data within the confines of a single disk file regardless of the underlying file system. While not trying to change what structures you can actually store in those blocks, Structured Storage *does* change where those blocks actually exist on disk. It also allows you to standardize specific data structures within a few of those blocks, but by no means requires it.

What good is this standardization? It means that given an arbitrary disk file that conforms to the Structured Storage Model, any other piece of code that also knows the model might be able to open that file and examine its contents. If that file contains a standardized property set for something like "Summary Information" that contains title, subject, author, and keywords, then any application familiar with that property set could open the file and determine the title and subject of the document, who wrote it, and possibly search the list of keywords. This is opposed to what we have today where only the application that originally wrote the file can open and browse the contents of that file. Structured storage enables anyone, such as the system shell, to perform the same browsing functions. To the end-user, this eliminates a host of application-specific techniques to find information in files consolidating it into one uniform system-wide tool. Through such a tool an end-user could enter in a query like "Find all the documents I wrote with the word 'vegetarian' in the title" and the system shell would go off and find all files that contained such information. For the applications programmer, this replaces the need to write full browsing yourself with writing files containing a specific property set stream. As we'll see, the latter is considerably less work.

Standardization is one thing, implementation is another–how can we all be assured that we'll all implement the Structured Storage standards in our files the same way? Well, we can't, so Microsoft has provided an implementation of structured storage called Compound Files contained in OLE 2.0's STORAGE.DLL. Compound Files implements storage and stream objects on top of the FAT file system under Windows 3.1 and on top of FAT and NTFS under Windows NT.[1] You can use Compound Files as you would any other set of file I/O APIs, and in fact, how you manipulate a stream object corresponds directly with how we manipulate a file handle today as this chapter will demonstrate. To your application, the data in a stream always appears contiguous, although that data may not actually be stored contiguously within the file itself. This is no different than how file systems like FAT let you look at a file as a contiguous block of bytes through a file handle although the actual bytes are scattered across the disk in disparate sectors.

Ages ago applications had to concern themselves with the absolute sectors in which they stored their data which was painful, to say the least. File systems came along and allowed applications to treat files as a single block, not as separate sectors–what a blessing! Today the structures within files themselves as becoming as painful to maintain as absolute sectors used to be and so OLE 2.0, in Compound Files, is providing the equivalent of a file system within a file to redeem you from the burden of maintaining seek offsets a plenty. The beginning of this chapter will hopefully convince you that problems exist and how structured storage

1What about the Mac? Do we care?

solves those problems, which is then followed by a review of the features and capabilities of OLE 2.0's implementation of storage and stream objects.

In keeping with the theme of this book, you know there are storage and stream objects, so how do you obtain pointers to those objects? What are the interfaces those objects support? What can you do with those objects? You will find your answers in this chapter as we explore how to make simple use of compound files, followed by more complex usage, leading finally to a discussion on how these objects relate to the rest of OLE 2.0 and in particular, compound document containers and objects.

If you choose to use compound files you will, of course, change your absolute file format on the disk, although there is no need for your internal data structures to change at all. For almost all applications, compound files are optional. The only code that must use them are OLE 2.0 compound document objects and containers: a container must give every embedded object must be its own piece of storage from which to load and save itself. However, this does not mean you must use compound files for your container's own native storage: you can create a compound file in memory for an embedded object and then write the contents of that memory to your own files wherever you like. An outline of this process is described in a section of this chapter and is treated again in Chapter 9 in the context of OLE 2.0 containers. OLE objects must be able to load and save embedded object data through compound files, but that does not change any aspect about your application's disk file format.

One of the major programmatic benefits of structured storage is that you can use it as a sharable data transfer medium, that is, an storage object (a piece of a compound file) can be marshaled to another process. This means that instead of always transferring data through global memory you can transfer data through a pointer to data on the disk without having to actually load any data. The only process that needs to actually pull the data into memory is the process that needs to edit and manipulate that data. In other words, only one copy exists in memory. The process that manipulates the data furthermore has incremental access improving performance even more. This capability can be used for data transfer as well see in Chapter 6, and is used for compound document object transfer as well see in Chapters 9-11.

Microsoft was originally motivated to define Structured Storage to improve the performance of compound document scenarios and for the tremendous end-user benefit of shell level document browsing. Other innovations for dealing with traditional files really do little to simplify how you maintain a complex structure within any particular file. Memory-mapped file I/O under Windows NT just changes the expression of that complexity from a file handle into a pointer with some benefit, of course, but about the same amount of paint. Structured Storage is the one technology to truly solve the problems of file structure just like the file system solved the problems of disk structure.

Motivation

A man and his 16-year old daughter were traveling to Washington, D.C. from Seattle via Chicago. Their plane arrived late in Chicago causing them to miss their connecting flight, and so they were awarded First Class seats on the next flight. The father was excited–it had been a long time since he last rode in the posh forward cabin and looked forward to warming his innards with the smoothness of fine drink. He exercised his early-boarding privilege and, as protocol demands, the steward asked him if he would like a drink before takeoff. No problem, gin and tonic it is. The man, drink in hand, leaned back and began enjoying the start of what would be a wonderfully relaxing flight. That is, it would have been if his daughter had not asked a simple but probing question, one that would change pleasure to guilt, one for which he had no answer. She asked with a menacing stare, "Just what exactly does that *do* for you?"

So what does structured storage, and OLE 2.0's implementation of Compound Files, *do* for you? In a manner of speaking, compound files give you the warmth of alcohol without the hangover. But in no way are you required to drink: this section will, however, encourage you to "start the habit" so-to-speak as we explore how we might add file I/O capabilities to the Patron sample, which to this point has no provision for such operations.

Patron Files with a Hangover

Patron is designed to have a document made of pages where each page serves as the inn for any number of bitmaps, metafiles, and compound document objects, that is, tenants. We need a file layout for each document that describes those contents. If we were to implement Patron using traditional file I/O, we would create a layout with three primary structures:

1. File header structure contains how many pages are in the document, the printer configuration (a serialized DEVMODE structure), and an offset to the first page.
2. Page structure contains a header specifying how many tenants (metafiles, bitmaps, OLE

objects) live on this page, an offset of the next page, and a variable length list of offsets to the tenants. All page structures are stored sequentially in the file before writing any tenants.

3. Tenant structures contain header information such as the length of the record in the file and the type of tenant followed by a serialization of the tenant's actual data.

Such structures would result in a layout shown in Figure 5-1.

Figure 5-1: A possible Patron file with traditional file I/O.

Certainly this sort of layout is manageable, albeit tedious. When we write a file in this format we would first write the file header, then all the page headers, then all the tenants. Writing the file header is simple since we know its size and can easily calculate the offset of the first page to store in this header. Before writing the page headers, we need to build the entire list in memory to determine the total size of the page list and store the appropriate tenant offsets in each page structure. Once we have this list, we write it all out to disk at once, then writing each tenant in turn. Besides a little tedium in calculating all the offsets, this code would be simple enough and performance would be good since all the pages are at the front of the file and we only have to do potentially large seeks when accessing a particular tenant.

Let's say now that the user deletes a page. We have two choices the next time we save. One is to rewrite the entire file, which is typically the easiest choice. The other is to mark the deleted page in the file as 'unused' and mark all the tenant spaces that were deleted as 'unused' as well. This would not reduce the file size but would allow a very quick save.

Now the user adds a page in the middle of the document and adds a few tenants to that page. If this new page structure can fit in an unused page block from one previously deleted, we could attempt to incrementally write the new document, storing the new page structure over the deleted one and modifying the stored seek offset of the previous page to point to this new one. If there are no open spaces for this new page, we can either append the page structure to the end of the file, store it in free space from a deleted tenant, or choose to rewrite the entire file. Choosing either of the first two options defeats the purpose of originally storing page structures in sequence before storing any tenants. Choosing the last option is potentially slow since Patron files, housing bitmaps and metafiles and OLE objects, can become quite huge.

At this point we do what engineers call "compromise" and weigh the possible options: we can have incremental saves with file fragmentation or we can have efficient files for reading with slower-than-molasses saves. In a performance-driven market, we generally choose incremental saves so the save timings look great in the trade rag reviews. Get a few developers to put in some overtime and you'll have a wonderfully elaborate scheme of managing free space in your files as best you can as you would handle free space in any memory manager. You would allow 'fast saves' or 'full saves,' the latter of which would full defragment the file by performing a full rewrite. Cool. We just turned a problem into a couple of 'features' by writing a great deal of file management code. That's all well and done, but the effort to write such code may have required you to sacrifice other important features of your application. Next version, I guess.

Realize that investing an enormous effort providing for file defragmentation does not really buy you a whole lot until the user defragments their hard disk. Even though your applications sees the file as defragmented, the actual physical location of the bytes on the disk is generally random since the file system isolates you from physical sector locations. All that effort you made to defragment your file structures doesn't really improve performance significantly; when the disk itself is fragmented, a 128 byte seek in your file may equate to a 128MB seek on the disk and a 10MB seek in your file may actually only seek 10K on the disk.

File defragmentation is only a way to compress the file or reduce the file size down to minimum, an important feature for users that might want to copy the file to a floppy or upload it at a painful 2400 baud to a BBS. Implementing your own defragmentation serves no other purpose.

Yep, that code felt good, but now comes the hangover: all that code does not do everything you wanted.

The Non-Alcoholic Alternative

If we could afford ourselves the luxury, we would save an enormous amount of time if Patron could just write its files in a directory tree instead of one huge file. This would have serious problems for end users since there would not be one file to just copy off the system–it would have to be a more involved backup of many directories. But again, if we could, using a directory tree would be much, much simpler to implement:

1. The file becomes the 'root' directory for the storage, not necessarily the root directory on a disk as this root may be a subdirectory somewhere on the disk, but as far as Patron is concerned, its the root of the file. In this directory we store a *file* that contains the number of pages in this

document, the printer configuration, and the *name* of the first page. This is the same type of information that we would store in a file header.

2. Each page itself is a subdirectory off the root with a name like PAGEnnnn.nnn which would allow up to 10 million pages per document. Within each page subdirectory is a file containing the page information which could contain the *names* of tenants, like a single file format would store offsets.

3. Each tenant would have an entire file to itself, or better yet, an entire subdirectory off the page in which we could write as many files as necessary to save that tenant.

This scheme would create the layout shown in Figure 5-2.

Figure 5-2: A possible Patron storage scheme using directories.

This implementation has promise (again at the sacrifice of file portability) When we add a page, we only need to create another directory off the root and write a file containing page information. We never have to worry about rewriting the entire storage; there is simply no need because we let the file system worry about actual placement of the data. When we delete a page, we delete the directory and all its contents, returning the space to the file system which manages reuse of that space already. Therefore we have no need to provide for defragmentation, instead allowing the end user to choose their own tool.

In addition, adding a tenant to a page is as simple as creating a subdirectory off the page and writing all the tenants information into however many files is convenient in that new directory. The page itself doesn't even need to keep track of the tenants itself since it can just ask the file system for a list of directories in the form, for example, TEN*.*.

All elements, the file header, the pages, and each tenant, individually benefit from incremental saves. Changing the size of any particular file in this storage model or changing the number of subdirectories of any other directory does not affect any other aspect of the entire storage. There is no need for ever rewriting the entire storage. Applications that store bitmaps and metafiles can appreciate this, since those data tend to become very large.

Of course, the FAT file system can slow to a crawl with too many files or subdirectories in a single directory since it uses a sequential search through the directory sectors. More advanced file systems like NTFS use binary search algorithms to improve performance, so depending on your needs you would choose the appropriate file system on which to base your storage.

Alcohol without the Hangovers: Compound Files

Up to now you have two choices: drink and get a hangover or drink non-alcoholic beverages and risk alienation from your social group. What we really want is an alcoholic drink with no side effects (synthahol), in other terms, we want the efficiency and benefits of using a directory structure like the file system but we want to keep that within a single file in order to allow document portability. We want a file system as efficient as NTFS implemented within a file. We want OLE 2.0's Structured Storage Model.

Structured storage in itself is not implementation but rather a specification of how storage is exposed to the system and applications. The OLE 2.0-provided implementation of the model is called Compound Files which sits on top of the actual file system and takes full advantage of that system. Compound files provides all the free-space management for you just like any other file system, exposing your files to your application as two objects: streams and storages. The former implements an interface called IStream, the latter an interface called IStorage. Streams are the logical equivalent of a disk file while storages are like directories.

Using streams and storages we'll implement Patron's file I/O with a structure shown in Figure 5-3. Directories have become storages, files have become streams. Best of all, instead of this entire structure living separated on the file system, it lives within a single file that an end user may copy as any other a single file.

Figure 5-3: Patron's storage scheme using Compound Files.

Features of Compound Files

The design of structured storage, and specifically the Compound File implementation, provides a number of

significant features which applications can exploit to their benefit. You need to be familiar with these features before we can apply them in practice:

1. Streams, Storages, and LockBytes Objects: Units of data, directories, and byte arrays on the physical device.
2. Element Naming: Storages and streams can have names up to 31 characters.
3. Access Modes: Storages and streams can be opened in transacted mode such that changes are cached until committed (flushed).
4. Incremental Access: Modifications to any element does not require a complete file rewrite and elements can be read as little as necessary.
5. Sharable Elements: Storage and stream objects can be passed to other processes.

Again, use of compound files is completely optional for all but OLE 2.0 container applications, and even those can skirt the issue (but with a little cost to overall performance). However, many applications can greatly simplify their storage management by building on top of compound files, implementing an incremental save feature with little code or a feature to revert changes to a document without having to manage any previous state yourself. That code now exists in the storage implementation.

Stream, Storage, and LockBytes Objects

Structured storage is defined in terms of three objects: a stream contain data like a file, a storage contain streams and storages like a directory, and a LockBytes presents some physical device as a generic byte array. These objects combine to for the structured storage model as shown in Figure 5-4.

Figure 5-4: LockBytes sit on a device, a root storage build on the LockBytes, streams and storages live inside the any other storage.

All three objects Windows Objects just as described in Chapter 3, that is, they each implement one or more interfaces and provide separate function tables for each.[1] A LockBytes object supports the ILockBytes interface and to obtain a pointer you can either call an OLE 2.0 API for standard implementations or implement your own, in which case you already have the pointer. A storage object implements the IStorage interface to which you obtain a pointer by calling one of a number of other APIs or by calling a IStorage member function in an existing storage object . Some of the APIs create a storage object on the default file system whereas others allow you to create a storage object on top of a specific LockBytes. Those storage object that are attached to a real file system entity also support an interface called IRootStorage which is used primarily in low-memory save situation. In any case, a stream object implements the IStream interface, and such a pointer is generally obtained by calling a member function in the IStorage interface.

The section "Compound File Objects and Interfaces" below describes the exact APIs through which you obtain interface pointers, describes the ILockBytes, IStorage/IRootStorage, and IStream interfaces, and highlights the differences between the specification of Structured Storage (that is, the interfaces) and the Compound File implementation in OLE 2.0. Note also that all storage-related interfaces are defined in STORAGE.H and derive from IUnknown; complete details on their parameters and return values are described in the OLE 2.0 Programmer's Reference.

Element Naming

Storage and streams are identified by a name that can be up to 32 characters long with the exception of a root storage associated with a disk file that may have a name as long as the file system allows. The name of a root storage must obey the restrictions of the file system, otherwise the name may contain any characters above ASCII 32 with the exception of ".", "\", "/", ":", and "!". Characters below ASCII 32 are reserved for system use with the exception of ASCII 3 which is for exclusive use by a compound document container application for marking special elements, which won't be a topic in this chapter. Compound files store the name as provided by the caller with no conversion to upper or lower case, but all comparisons made on the names under Windows 3.1 are case-insensitive.

The actual names of elements in compound files are generally not intended to be shown directly to an end user and therefore need not be localized. When it becomes necessary in a future release of Windows there will be standard place to store a localized user-readable name.

Access Modes

Streams and storage objects support access modes as any traditional file, indicated through STGM_* flags,

1Really, I'm not trying to pound this into your head...well, maybe.

many of which translate directly into OF_* flags that OLE 2.0's compound file implementation passes directly down to the Windows OpenFile function:

Structured Storage Flag	Definition using OpenFile Flags
STGM_READ (default)	OF_READ
STGM_WRITE	OF_WRITE
STGM_READWRITE	OF_READWRITE
STGM_SHARE_DENYNONE	OF_SHARE_DENY_NONE
STGM_SHARE_DENYREAD	OF_SHARE_DENY_READ
STGM_SHARE_DENYWRITE	OF_SHARE_DENY_WRITE
STGM_SHARE_EXCLUSIVE	OF_SHARE_EXCLUSIVE
STGM_CREATE	OF_CREATE

As with OpenFile, any of these flags can be OR'd together providing the same effects on the compound file as they would have with any other traditional file.

OLE 2.0's compound files support a few other flags with special functions above traditional file I/O:

Structured Storage Flag	Function
STGM_DIRECT (default)	Opens the element for direct access.
STGM_TRANSACTED	Opens the element in transacted mode; changes are buffered and not saved until the element is committed. OLE 2.0 only supports transactioning on storages.
STGM_FAILIFTHERE (default)	Prevents overwrites.
STGM_CONVERT (storages only)	Allows an application to convert any existing file (or stream or LockBytes) into a storage that contains a single stream named "CONTENTS" where the stream contains the exact data in the original file. This allows an application to use compound file APIs to open any file, only concerning itself with the differences once the file is open as a storage. If the file is opened with STGM_DIRECT, the old file is immediately converted on disk, and therefore STGM_CONVERT always required STGM_WRITE.
STGM_DELETEONRELEASE	Deletes the file from the disk when ::Release on the storage object managing that file reduces the reference count to zero. Highly useful for temporary files.
STGM_PRIORITY	Allows an application to reduce the cost of opening a storage in transacted mode by pre-reading specific streams and excluding them from being buffered. This rarely used feature is described in further detail in the OLE 2.0 Programmer's Reference.

Transacted Storages

The most interesting, and certainly the most powerful, of these modes is STGM_TRANSACTED. When you open a storage in transacted mode, the compound file implementation does not make any changes to the actual disk file (or LockBytes) until you commit those changes (see IStorage::Commit). Instead, any changes are recorded in memory inside a copy of the storage's structure. To reduce overall memory use, a snapshot copy of the element in the disk file is not made until a change is made to that element, that is, very little memory is used if you open a file in transacted mode and never make any changes. Much more memory is used, of course, if you make a one-byte modification to every stream since that requires a snapshot of each stream to exist in memory.

When a transacted storage is committed, the snapshot copies of the modified elements are written to disk. If the transacted storage is released (IStorage::Release) or reverted (IStorage::Revert) the modified copies are discarded and the storage is put back into a state as if it were just opened.

Any storage object may be opened in transacted mode, regardless of the mode of its parent storage, that is, if you open a root storage with STGM_DIRECT you may open a sub-storage with STGM_TRANSACTED and only that sub-storage is transacted. In other words, only when the sub-storage is committed will there be

any change in the outer file opened in direct mode.

When multiple levels of storages are opened in transacted mode, committing changes notifies the immediate parent of the committed storage. This in turn becomes an transacted change in that parent, that is, the changes percolate upwards. Only when the highest transacted storage is committed are all those changes actually saved permanently.

Transacted changes reflect any operations you might perform on a storage through the IStorage interface. For example, newly created storages or streams are created in memory and not reflected to disk until the outermost commit occurs. Likewise moving, renaming, or destroying elements has no permanent effect on the physical device until that final commit. Another interesting feature of a transacted storage is that you may open a storage as read-only transacted and manipulate that storage as if it were read-write: the only operation you are barred from is a commit.

Use of transacted storage enables easier file sharing. Two or more different applications, or different users on a network, may open the same transacted storage with read-write permissions but not with exclusive access. When the application commits that storage (that is, when the user saves) the application tells the storage to only commit whatever is current, that is, do not commit changes to those parts of the storage that have been modified by another user since the file was originally opened.

Transacted mode enables a new technique in application design. Since most changes made to storages and streams are recorded in memory, writing directly to a transacted storage or stream is only slightly slower than writing directly to memory. For any block of bytes that you would normally access using a memory pointer, you could use a stream. By doing so you gain a number of benefits:

- Writing past the end of the stream automatically expands the stream instead of causing a UAE.

- Saving the data requires a simple commit on the storage in which the stream lives instead of copying the data from your own memory structures into the stream before you do the commit.

- Undo is a simple matter of reverting changes. You can implement multiple levels of Undo by duplicating the streams at appropriate intervals, letting the compound file implementation worry about memory allocation and copying the data.

There are cases, of course, where a memory structure contains fields like pointers that make no sense to save persistently, so you would still need a translation from the memory structure to the file structure at commit time. However, for even small structures that are persistent representations this technique can save you from keeping the same data in a memory structure–when you need it, load from a stream, when you change it, write to the stream.

Incremental Access

An incremental save feature is one of the end user's favorite time-saver–instead of rewriting the entire file every time the user says File/Save, you only save the bits that actually changed. For example, adding one tenant to a page in Patron should only save a modification of the page header and the information for the new tenant instead of rewriting the entire page or the entire file. As we explored earlier in this chapter, providing such a feature can mean a lot of design and coding work on your part if you stuck with traditional file I/O for your storage.

Making simple changes to a few values in a structure has always been a cheap incremental operation where just that structure needs modification. However, when you shrink, enlarge, move, add, or delete elements, things can get extremely tedious if you are implementing this on your own. Since compound files isolate you from the details about the allocating or freeing space in the file, your modifications to any element do no affect any other element.

When a compound file needs to expand a stream to accommodate more data, it finds the required amount of space somewhere in the file (either in previously freed regions or at the end of the file) and reserves it as part of the original stream. The data and actual location of the original stream remain the same as the stream expands into the newly allocated region. Since no other stream or storage is affected by this expansion there is no need to make any further changes to the physical file. In the same manner, if you add a new stream or storage to an existing storage, the space for that new element as well as any space needed to update the storage's directory is either recycled from free space or allocated at the end of the file. No other elements need to move or change in any way to accommodate the new addition. Deleting any element is a simple matter of marking that space as free, allowing it to be overwritten at a later time by any other element.

Compound files make incremental saves the norm and full, compact saves the exception. In normal

operation, a compound file will eventually become fragmented internally which can make the file larger than necessary. A full save operation copies all the data from the existing file into a new file which defragments all the streams and storages, greatly reducing the wasted space in the file. As an example of how this is done we'll implement a File Manager extension called Smasher in the "Compound File Defragmentation" section at the end of this chapter.

Compound files not only provide for incremental saves, but for the more generic incremental access. This is most important for compound document objects such that when activated for editing they are given a storage object from which they only need to read as little data as they need to perform their editing tasks. When asked to save they only need to write as much data has changed, nothing more. In OLE 1.0, **all** of the object's data had to be loaded into global memory and passed back and forth via DDE: when the container loaded the object all the data went into memory; when the object needed to save, it put all the data back into memory again. Such transfers were slow and memory-hungry. Compound files, on the other hand, take very little memory to actually transfer since the element are sharable across process boundaries.

Sharable Elements

A most important feature of structure storage that benefits data transfer and compound document implementation is that storages and streams can be marshaled across process boundaries. This means that you can use a storage or stream object to transfer data between applications instead of being continually forced into global memory. With a storage or stream object, the data can live on the disk until it becomes necessary by whoever consumes that data; at that time, the consumer can load the data directly from disk through the object.

We will explore the implications of this benefit in later chapters. Chapter 6, for example, shows how OLE 2.0's Uniform Data Transfer mechanisms can all use compound file elements to transfer literally anything. Something like a large bitmap that always lives on disk is best transferred such that it remains on disk; since data transfer can use compound files as a transfer medium in addition to many others, the source of the data can choose the best medium for transport.

Chapter 9 and 10 show how a storage object is used to transfer embedded object data between container and server. Structured storage was originally created to solve the specific problem of OLE 1.0 where all embedded object data had to be loaded into global memory, exchanged via DDE messages, and possibly copied on the receiving end. This resulting in multiple copies of the data existing at once which brought any system to its knees with large bitmaps. Embedded object in the OLE 1.0 model always had to read and write their data to global memory and pass it to the container who was responsible for placing it on disk. When the container reloaded the object, it had to read the entire object data from disk into memory and pass it to the object. All of this was very slow and inefficient.

With structured storage, OLE 2.0 containers create an IStorage instance for each embedded object. Creating any new storage uses little memory, and if the storage is direct and lives on disk, then very little memory will ever be used. In any case a container hands that storage to the embedded object who becomes solely responsible for loading and saving itself to that storage. This effectively transfers data directly between the embedded object and the container's document file, bypassing container code completely. Since the embedded object has the entire storage to itself, it benefits from incremental access–it only needs to load what is necessary to display or edit the data, and benefits from incremental saves like any other storage user.

Compound Files Objects and Interfaces

As mentioned before, compound files are built on three objects: storages, streams, and LockBytes, that support the IStorage (possibly IRootStorage), IStream, and ILockBytes interfaces respectively. Each object handles specific functions within the compound file implementation. The interfaces actually specify more functionality than is implemented in OLE 2.0 so the sections below will point out what features are not available. Also note that when these sections list various storage-related APIs, those prefixed with "Stg" are found in STORAGE.DLL and prototyped in STORAGE.H; otherwise they exist in OLE2.DLL and are prototypes in OLE2.H.

Storage Objects and the IStorage Interface

A storage object is like a directory which may contain any number of storages (subdirectories) and any number of streams (files), but in itself does not hold any data. Each storage object supports the IStorage interface described in Table 3-1. Since any sub-storage it a storage in itself, like a directory is always a directory, they may themselves contain more storages and more streams ad nauseam until you deplete your available disk space. Each storage has access rights (read, write, share, etc.), a feature lacking in MS-DOS

directories. A storage object can enumerate its elements or copy, move, rename, delete, and change times on an element. A storage gives you a programmatic equivalent of COMMAND.COM functions which is generally lacking in traditional file I/O libraries.

Table 3-1: The IStorage Interface

IStorage Member	Equivalent	Description
Release	none	Closes the storage. For a root storage, closes the compound file it represents. If the storage is opened in transacted mode, Release also implies a Revert.
CreateStream	OpenFile	Creates and opens a stream within the storage
OpenStream	OpenFile	Opens an existing stream.
CreateStorage	mkdir, chdir	Creates and opens a new sub-storage.
OpenStorage	chdir	Opens an existing storage.
CopyTo	copy	Copies the entire contents from the storage into another storage. The layout of the destination storage may differ.
Commit	none	Insures that all changes made to the storage open in transacted mode are reflected on the device.
Revert	none	Discards any changes made to the storage opened in transacted mode since the last Commit.
EnumElements	dir	Returns an IEnumSTATSTG object that enumerates the sub-storages and streams directly contained in the storage.
MoveElementTo	copy, delete	Moves a sub-storage or a stream from the storage into another storage. May move or copy the element.
DestroyElement	delete, deltree[1]	Removes a specified sub-storage or stream from within the storage. If a sub-storage is destroyed then all elements contained within it are also destroyed.
RenameElement	rename	Changes the name of a stream or sub-storage.
SetElementTimes	none	Sets the modification, last access, and creation date and time of a sub-storage or stream, subject to file system support.
SetClass	none	Associates a CLSID with the storage which can be retrieved with Stat. Allows anyone to know who might be able to manipulate the contents of the storage.
SetStateBits	none	Marks the storage with various flags defining how other agents might be able to treat this storage.
Stat	varies	Retrieves statistics for the storage such as name, create, modify, access times, etc.

Compound File Implementation of storages: OLE 2.0 implements complete storages that support transactioning. All member functions of IStorage work as specified with the exception of ::SetStateBits which has no behavior. The data for a storage may not be contiguous inside the file itself.

How you obtain an IStorage pointer on a storage object depends on whether you want a root storage object or a sub-storage below another storage object. For the latter you call IStorage::CreateStorage. For root storages OLE 2.0 offers four API, two of which create a new compound file and two that open an existing file:

StgCreateDocfile[2]	Opens a new compound file given a filename in the form of a root storage on the default file system LockBytes. Will generate a temporary file if no filename is provided. If the file already exists, this function may either fail or overwrite the existing file, depending on flags you pass.
StgCreateDocfileOnILockBytes	Opens a new compound file on a given LockBytes object but otherwise acts like StgCreateDocfile.
StgOpenStorage	Opens an existing compound file given a filename but will not create a new file as will StgCreateDocfile.
StgOpenStorageOnILockBytes	Opens an existing compound file that exists in the given LockBytes but otherwise acts like StgOpenStorage.

There are three additional APIs in STORAGE.DLL that are frequently used with the APIs listed above:

1MS-DOS version 6.0.

2Note that the name 'Docfile' is an archaic term for a compound file which has been preserved in these function names for compatibility with early releases of OLE 2.0.

StgIsStorageFile	Tests if a given file contains a compound file.
StgIsStorageLockBytes	Tests if a given LockBytes contains a compound file.
StgSetTimes	Provides the ::SetElementTimes equivalent for a root storage without having to open the storage.

A root storage obtained with one of the Stg* APIs above will also support an interface called IRootStorage in addition to IStorage. You can obtain a pointer to this interface by calling IStorage::QueryInterface with IID_IRootStorage. IRootStorage contains one member function (besides IUnknown, of course) called SwitchToFile that is described below in "Low Memory Save Operations."

Once you have a pointer to an IStorage interface you generally use that storage object by calling IStorage member functions to create other sub-storages or streams and to manage elements within the storage. In general the ::Create/OpenStream and ::Create/OpenStorage members are the most optimized whereas other members are not. For example, you will see much better performance by storing a table of your storage's elements in a stream rather than relying on ::EnumElements. You should also make judicious use of ::MoveElementTo, ::RenameElement, and ::DestroyElement, making sure that you make few calls to these functions during performance-critical operations.

A number of the Stg* APIs and interface functions have some extra parameters to deal with transaction optimizations when a storage object (root or otherwise) is opened with STGM_TRANSACTED. The optimizations allow an application to exclude specific elements of a storage from being transacted although the entire storage is opened transacted. By excluding such elements, you reduce the overall amount of memory necessary to record changes to your data. These exclusions operate along with the STGM_PRIORITY flag for which you should refer to the OLE 2.0 Programmer's Reference. The topic of optimizations and exclusions is not treated in this chapter.

You can also pass an IStorage pointer to a few other functions in OLE2.DLL that help applications and objects storing their data in compound files to mark those files in such that an external agent could look at that storage and get an idea who might be able to load or edit the contents. We'll see where to call some of these APIs later when we make use of compound files:

WriteClassStg	Serialize a CLSID into an OLE-controlled stream that identifies the application or object writing other data.
ReadClassStg	Load the CLSID previously written with WriteClassStg.
WriteFmtUserType	Serialize a clipboard format and a user-readable name describing the format of the contents of the storage. This could be used by another application to see what sort of format might be contained in your streams and perhaps load them.
ReadFmtUserType	Reads the clipboard format and the string previously written with WriteFmtUserType.

Stream Objects and the IStream Interface

A stream is the equivalent of standard files but exposed through the IStream interface shown in Table 3-2. An IStream pointer to a stream in a compound file is always obtained through IStorage::CreateStream or IStorage::OpenStream although there are a few OLE 2.0 APIs for obtaining a stream object outside of a compound file as well:

CreateStreamOnHGlobal	Builds a stream object on a piece of global memory.
GetHGlobalFromStream	Returns the global memory handle used in a stream created with CreateStreamOnHGlobal.
CreateStreamOnFile	Builds a stream object on any existing file handle.

These functions in OLE2.DLL are generally used to serialize some data to a stream that we can pass to another process. In those situations the stream is being used as a generic transfer medium and not for file-related storage.

Many stream functions equate directly to existing file functions so most code that uses functions like _lread and _lwrite are easily rewritten to handle a stream. Each stream in itself has access rights (read, write, share, etc.) and a single seek pointer just like files. However, since a stream object does not use a file handle, you can open a stream and leave it open with no penalty to the underlying file system. If you maintain an open read-only stream, you don't exclude other parts of your application from opening that stream with write permissions on a temporary basis, although you will prevent other code from opening the stream for exclusive access.

Table 3-2: The IStream Interface

IStream Member	File Equivalent[1]	Description
Release	_lclose	Closes the stream for the user of the IStream pointer through which it's called.
Read	_lread	Reads a given number of bytes from the current seek pointer into memory.
Write	_lwrite	Writes a number of bytes from memory to the stream starting at the current seek pointer.
Seek	_llseek	Moves the seek pointer to a new offset from the beginning of the stream end of the stream, or the current position.
SetSize	_chsize	Pre-allocates space for the stream but does not preclude writing outside that stream (see below).
CopyTo	_fmemcpy	Copies the a number of bytes from the current seek pointer in the stream to the current seek pointer in another stream.
Commit	none	Insures that all changes made to the stream open in transacted mode are reflected on the device (not supported in OLE 2.0).
Revert	none	Discards any changes made to the stream opened in transacted mode since the last Commit (not supported in OLE 2.0).
LockRegion	_locking	Restrict access to a byte range in the stream instead of the stream as a whole (not supported in OLE 2.0).
UnlockRegion	_locking	Frees restrictions set with LockRegion (not supported in OLE 2.0).
Stat	_stat	Retrieves statistics for the stream such as name, create, modify, access times, etc.
Clone	_dup	Creates a new stream object with an independent seek pointer that references the same actual bytes.

The ::Read, ::Write, and ::Seek members of IStream are the most optimized for performance in the entire compound file implementation. These will show speeds lacking traditional MS-DOS operations by only a few percent. Other operations like ::CopyTo and ::SetSize are potentially expensive and should be used with such caution.

Inside the compound file the data contained in a stream is not necessarily contiguous, just like the physical location of the contents of an MS-DOS file on a hard disk may be in widely separated sectors. From the view of the streams user, the stream is contiguous–you let the stream implementation worry about exact placement.

1Both Windows API and C run-time functions are shown here.

Compound File Implementation of streams: OLE 2.0 does not implement stream transactioning nor region locking, that is, ::Commit, ::Revert, ::LockRegion, and ::UnlockRegion are no-ops. In addition, the IStream interface allows streams to be 2^{64} bytes, that is, a single read or write could transfer 2^{64} bytes at a time and the seek pointers is a 64-bit value. OLE 2.0's implementation is limited to 2^{32} byte transfers and uses a 32-bit seek pointer.

LockBytes Objects and the ILockBytes Interface

All root storage objects (and only root storage objects) are built on top of some byte array represented by a LockBytes object that implements the ILockBytes interface shown in Table 5-3. LockBytes isolate the root storage from any concern about how the bytes actually get to their final destination on whatever storage device the LockBytes accesses. For example, a LockBytes built on the file system writes bytes to a file; a LockBytes on global memory writes data to some memory block.

Table 5-3: The ILockBytes Interface

ILockBytes Member	Description
ReadAt	Reads a number of bytes from a given location in the byte array. If there are not enough bytes on the device to satisfy the request, Read returns what can be read.
WriteAt	Writes a number of bytes to a given location in the bytes array, expanding the allocations on the device to accommodate the request.
Flush	Ensures that any internal buffers in the LockBytes are written to the physical device.
SetSize	Pre-allocates a specific amount of space on the device.
LockRegion	Locks a range of bytes on the device for write access or exclusive access.
UnlockRegion	Reverses a LockRegion call.
Stat	Fills a STATSTG structure with information about the LockBytes, which reflects information about the device.

Compound File Implementation of LockBytes: OLE 2.0's LockBytes implementation supports the entire interface, region locking included.

A LockBytes object that writes to the default file system is used inherently when you create a compound file with StgCreateDocfile or StgOpenStorage. The specific implementation of this LockBytes is not available for use outside this context. OLE 2.0 also provides a standard LockBytes built on global memory that you manage through two APIs in OLE2.DLL:

CreateILockBytesOnHGlobal	Creates LockBytes object on a piece of global memory that either this function or the caller may allocate.
GetHGlobalFromILockBytes	Returns the global memory handle in use by a LockBytes from CreateILockBytesOnHGlobal.

If you need to create a compound file on some other device than the file system or global memory, you may choose to implement your own LockBytes. For example, you may want to send the data across a network to a database without ever having to bother the storage object about the details. In this case you implement your LockBytes however you want (as long as you can provide an ILockBytes function table) and call StgCreateDocfileOnILockBytes or StgOpenStorageOnILockBytes to obtain an IStorage pointer. This IStorage is indistinguishable from any IStorage built on a different LockBytes.

While the read and write mechanisms in a LockBytes are similar to those in a stream, a LockBytes maintains no seek pointer and is instead always told from where to read or where to write. In addition, the actual physical location of the bytes may not be contiguous, that is, they may span multiple physical files, multiple global memory allocations, multiple database fields, etc. The purpose of the LockBytes object is to isolate any storage and stream objects from the physical aspects of the byte device.

Container Applications and Compound Files in Memory

CreateILockBytesOnHGlobal and GetHGlobalFromILockBytes allow an application to create a compound file image in memory and copy that memory anywhere else. The most common case

where this technique is useful is for OLE 2.0 container applications that do not wish to use compound files for their own documents since those applications are still required to provide some storage object to every embedded object. In this case, the container can first call CreateILockBytesOnHGlobal then call StgCreateDocfileOnILockBytes to get storage object built on memory. Once the embedded object is saved in that storage, you can GetHGlobalFromILockBytes, GlobalLock the handle, and write the contents of that memory anywhere you desire, into your own file format if desired. When reloading the embedded object you would allocate memory, load the contents of the file record into that memory, CreateILockBytesOnHGlobal, and call StgOpenStorageOnILockBytes to get a storage object for that embedding.

The ::Stat Member Function and STATSTG

Each interface contains a member function called ::Stat which is essentially identical to the standard ANSI C run-time _stat function. ::State returns its information in a STATSTG structure: the name of the element, creation and modification times, the type of object (storage, stream, etc.), the access mode under which the object is open, and whether or not the object supports region locking as described through various interfaces:

```
typedef struct FARSTRUCT tagSTATSTG
  {
  char FAR       *pwcsName;          //Name of the element
  DWORD          type;              //Type of element
  ULARGE_INTEGER   cbSize;          //Size of element
  FILETIME       mtime;             //Last modification date/time
  FILETIME       ctime;             //Creation date/time
  FILETIME       atime;             //Last access date/time
  DWORD          grfMode;           //Mode element is opened in
  DWORD          grfLocksSupported; //Support region locking?
  CLSID          clsid;             //CLSID of the element.
  DWORD          grfStateBits;      //Current state
  DWORD          reserved;
  } STATSTG;
```

This structure is also used to enumerate elements within a storage as shown in the next section through an interface called IEnumSTATSTG. The OLE 2.0 Programmer's Reference has complete details on the STATSTG structure. However, the one important point to remember about using this structure is that the storage DLL allocates the string pointed to by pwcsName is allocated using the task allocator from CoGetMalloc. You as the user of the STATSTG structure are responsible to free that string using the task allocator yourself:

```
[Code to get a STATSTG structure in the st variable]
LPMALLOC    pIMalloc;

CoGetMalloc(MEMCTX_TASK, &pIMalloc);
pIMalloc->Free((LPVOID)st.pwcsName);
pIMalloc->Release();
```

Note that if your call to CoInitialize worked on startup, then calls to CoGetMalloc will not fail if you always pass in a valid MEMCTX_ flag and a valid pointer to your LPMALLOC variable. Therefore there is no need in this code fragment to check the return value of CoGetMalloc.

Compound Files in Practice

Now that we've thoroughly beat into the deep earth all the interfaces and APIs related to compound files, we can look at how to actually apply it all to implementing file functions in an application. For this chapter I have modified the Chapter 2 version of Schmoo to write its data into a compound file, demonstrating the simplest use of compound files: open, read or write, then close. This version of Schmoo also retains compatibility with old versions of its files using the conversion feature of compound files allowing it to treat old files as storages.

I have also added compound file support into Patron that has a much more complicated storage scheme, since we implement part of the storage model shown before in Figure 5-3. Since Patron files exercise transactioning and incremental saves, they will, over time, become fragmented. A program called Smasher, which is really a File Manager Extension DLL, is presented at the end of this chapter to demonstrate how to defragment a compound file.

This chapter also shows a modification of the object DLL version of Schmoo's Polyline to deal with storage; the changes made here apply directly to an implementation of Polyline as a compound document object in Chapter 10. This implementation will show exactly what is required of a compound document

object, be it in a server DLL or server EXE, as far as storage is concerned.

Some applications will want to use Compound Files as a stand-alone technology, powerful as it is, without doing anything related to other OLE 2.0 technologies like component objects, data transfer, or compound documents. At a minimum, compound file usage requires COMPOBJ.DLL and STORAGE.DLL. The latter's necessity is obvious, but the former is a little more obscure. COMPOBJ.DLL is required since functions like IStorage::Stat will internally call CoGetMalloc (see "The ::Stat Member Function and STATSTG" above). Therefore an application that uses compound files must call CoBuildVersion and CoInitialize on startup and CoUninitialize on shutdown as explained in "The New Application for Windows Objects" in Chapter 4. However, since storage objects always live in DLLs, no marshaling will occur so a SetMessageQueue(96) call is not necessary.

Simple Storage: Schmoo

Let's start off with small modifications to Schmoo as shown in Listing 5-1. The changes are simple and only affect two functions: instead of opening a regular file with which to read or write data we open a root storage. Instead of using file I/O functions like _lread and _lwrite we obtain a stream pointer and use IStream member functions. The simplest uses of compound files can be reduced down to a few calls to APIs and interface members. Note that Schmoo is now also a component object user since storage and stream objects are considered component objects; therefore Schmoo must CoInitialize, etc.

DOCUMENT.CPP

```
[Other code unaffected]

/*
 * CSchmooDoc::ULoad
 *
 * Purpose:
 *  Loads a given document without any user interface overwriting the
 *  previous contents of the Polyline window.  We do this by opening
 *  the file and telling the Polyline to load itself from that file.
 *
 * Parameters:
 *  fChangeFile     BOOL indicating if we're to update the window title
 *                  and the filename from using this file.
```

Listing 5-1: Modifications to the Schmoo program for simple use of Compound Files.

```
 *  pszFile         LPSTR to the filename to load, NULL if the file is
 *                  new and untitled.
 *
 * Return Value:
 *  UINT            An error value from DOCERR_*
 */

UINT CSchmooDoc::ULoad(BOOL fChangeFile, LPSTR pszFile)
   {
   HRESULT        hr;
   LPSTORAGE      pIStorage;

   if (NULL==pszFile)
      {
```

```
    //For a new untitled document, just rename ourselved.
    Rename(NULL);
    m_lVer=VERSIONCURRENT;
    return DOCERR_NONE;
    }

/*
 * If this is not a Compound File, open the file using STGM_CONVERT
 * in TRANSACTED mode to effectively see old files as a storage with
 * one stream called "CONTENTS" (which is conveniently the name we use
 * in the new files).  We must use STGM_TRANSACTED here or else
 * the old file will be immediately converted on disk:  we only want
 * a converted image in memory from which to read.  In addition,
 * note that we need STGM_READWRITE as well since conversion is
 * inherently a write operation.
 */

pIStorage=NULL;

if (FAILED(StgIsStorageFile(pszFile)))
    {
    hr=StgCreateDocfile(pszFile, STGM_TRANSACTED | STGM_READ  //WRITE
        | STGM_CONVERT | STGM_SHARE_EXCLUSIVE, 0, &pIStorage);

    if (FAILED(hr))
        {
        //If we were denied write access, try to load the old way
        if (STG_E_ACCESSDENIED==GetScode(hr))
            m_lVer=m_pPL->ReadFromFile(pszFile);
        else
            return DOCERR_COULDNOTOPEN;
        }
    }
else
    {
    hr=StgOpenStorage(pszFile, NULL, STGM_DIRECT | STGM_READ
        | STGM_SHARE_EXCLUSIVE, NULL, 0, &pIStorage);

    if (FAILED(hr))
        return DOCERR_COULDNOTOPEN;
    }

if (NULL!=pIStorage)
    {
    m_lVer=m_pPL->ReadFromStorage(pIStorage);
    pIStorage->Release();
```

```
      }

   if (POLYLINE_E_READFAILURE==m_lVer)
      return DOCERR_READFAILURE;

   if (POLYLINE_E_UNSUPPORTEDVERSION==m_lVer)
      return DOCERR_UNSUPPORTEDVERSION;

   if (fChangeFile)
      Rename(pszFile);

   //Importing a file makes things dirty
   FDirtySet(!fChangeFile);

   return DOCERR_NONE;
   }



/*
 * CSchmooDoc::USave
 *
 * Purpose:
 *  Writes the file to a known filename, requiring that the user has
 *  previously used FileOpen or FileSaveAs in order to have a filename.
 *
 * Parameters:
 *  uType          UINT indicating the type of file the user requested
 *                 to save in the File Save As dialog.
 *  pszFile        LPSTR under which to save.  If NULL, use the current name.
 *
 * Return Value:
 *  UINT           An error value from DOCERR_*
 */

UINT CSchmooDoc::USave(UINT uType, LPSTR pszFile)
   {
   LONG        lVer, lRet;
   UINT        uTemp;
   BOOL        fRename=TRUE;
   HRESULT     hr;
   LPSTORAGE   pIStorage;

   if (NULL==pszFile)
      {
      fRename=FALSE;
```

```
    pszFile=m_szFile;
    }

  /*
   * Type 1 is the current version, type 2 is version 1.0 of the Polyline
   * so we use this to send the right version to CPolyline::WriteToFile.
   */

  switch (uType)
    {
    case 0:        //From Save, use loaded version.
      lVer=m_lVer;
      break;

    case 1:
      lVer=VERSIONCURRENT;
      break;

    case 2:
      lVer=MAKELONG(0, 1);    //1.0
      break;

    default:
      return DOCERR_UNSUPPORTEDVERSION;
    }

  /*
   * If the version the user wants to save is different than the
   * version that we loaded, and m_lVer is not zero (new document),
   * then inform the user of the version change and verify.
   */
  if (0!=m_lVer && m_lVer!=lVer)
    {
    char       szMsg[128];

    wsprintf(szMsg, PSZ(IDS_VERSIONCHANGE), (UINT)HIWORD(m_lVer)
      , (UINT)LOWORD(m_lVer), (UINT)HIWORD(lVer), (UINT)LOWORD(lVer));

    uTemp=MessageBox(m_hWnd, szMsg, PSZ(IDS_DOCUMENTCAPTION),
MB_YESNOCANCEL);

    if (IDCANCEL==uTemp)
      return DOCERR_CANCELLED;

    //If the user won't upgrade versions, revert to loaded version.
    if (IDNO==uTemp)
```

```
        lVer=m_lVer;
    }

/*
 * For 1.0 files, still use the old code.  For new files, use
 * storages instead
 */
if (lVer==MAKELONG(0, 1))
    lRet=m_pPL->WriteToFile(pszFile, lVer);
else
    {
    hr=StgCreateDocfile(pszFile, STGM_DIRECT | STGM_READWRITE
        | STGM_CREATE | STGM_SHARE_EXCLUSIVE, 0, &pIStorage);

    if (FAILED(hr))
        return DOCERR_COULDNOTOPEN;

    //Mark this as one of our class
    WriteClassStg(pIStorage, CLSID_SchmooFigure);

    //Write user-readable class information
    WriteFmtUserTypeStg(pIStorage, m_cf, PSZ(IDS_CLIPBOARDFORMAT));

    lRet=m_pPL->WriteToStorage(pIStorage, lVer);
    pIStorage->Release();
    }

if (POLYLINE_E_NONE!=lRet)
    return DOCERR_WRITEFAILURE;

//Saving makes us clean
FDirtySet(FALSE);

//Update the known version of this document.
m_lVer=lVer;

if (fRename)
    Rename(pszFile);

return DOCERR_NONE;
}
```

To write a simple file with a single stream, Schmoo performs the following steps where steps 1, 2 and 6 occur in CSchmooDoc::ULoad (DOCUMENT.CPP) and steps 3, 4, and 5 occur in CPolyline::WriteToStorage (POLYLINE.CPP), a new function added to CPolyline to handle storage objects in addition to WriteToFile that deals in file handles:

1. **StgCreateDocfile** using STGM_DIRECT | STGM_CREATE creates a new compound file, overwriting any that already exists. This returns an IStorage pointer for this new file. Since we use STGM_DIRECT there is no need to later call IStorage::Commit.
2. **WriteClassStg** and **WriteFmtUserTypeStg** sets various flags on the storage and saves standard class information.
3. **IStorage::CreateStream** using the name "CONTENTS" which returns an IStream pointer.
4. **IStream::Write** saves the data, passing a pointer to and size of the data to go into the stream.
5. **IStream::Release** closes the stream, matching IStorage::CreateStream.
6. **IStorage::Release** closes the storage, matching StgCreateDocfile.

In a fashion similar Schmoo makes the following calls to open and read the previously saved data during a File Open operation. Steps 1, 2, 3, and 7 happen in CSchmooDoc::ULoad and step 4, 5, and 6 happen in CPolyline::ReadFromStorage which handles storage objects as opposed to CPolyline::ReadFromFile:

1. **StgIsStorageFile** determines if the filename refers to a compound file created by the OLE 2.0 (or compatible) implementation of structured storage. This function looks for a signature at the beginning of the disk file to determine whether or not it can be read as a compound file.
2. If the file is a compound file, **StgOpenStorage** opens the storage for reading returning an IStorage pointer. Otherwise **StgCreateDocfile** using STGM_TRANSACTED | STGM_CONVERT opens a non-compound file as a storage object returning an IStorage pointer.
3. At the application's option, **ReadClassStg** loads the CLSID previously saved from WriteClassStg and IsEqualCLSID compares the expected class with the one in the file. If the two don't match, you didn't write this file.
4. **IStorage::OpenStream** on the name "CONTENTS" returns the IStream pointer for the data.
5. **IStream::Read** loads the data from the file into our memory structures.
6. **IStream::Release** closes the stream, matching the IStorage::OpenStream call.
7. **IStorage::Release** closes the storage, matching the StgOpenStorage or StgCreateDocfile calls.

The most interesting aspects of this code is the use of the STGM_CONVERT flag when dealing with an old file format and the correspondence between stream operations and traditional file operations, which are the topics of the next two sections. Note also that Schmoo, while it writes a CLSID to the storage, does not check this during File Open using ReadClassStg because I want Schmoo and Component Schmoo (CoSchmoo, modified for storage objects a little later) to retain file compatibility. Since the two applications write different CLSIDs into their storages we can just skip the ReadClassStg step. Using ReadClassStg is just an extra check you can perform to really validate a file before loading potentially large amounts of data.

Pulling Rabbits from the Hat with STGM_CONVERT

Schmoo is capable of reading and writing two different versions of its files; in Chapter 2 both formats were typical MS-DOS files. For this chapter and those beyond, Schmoo will maintain its version 2.0 file format in a compound file instead, but still remain compatible with old files (both version 1.0 files and the version 2.0 files generated with the Chapter 2 version). Schmoo still retains the capability to write an old file format, simply by virtue of preserving the old code. However, reading files of either format has changed significantly.

We could approach the problem of reading multiple formats in two ways. The first would be to test the file using StgIsStorageFile and failing that test, open and read that file using traditional file I/O functions. Doubtless you already have code that handles such an operation and I encourage you to keep it if works well. The second approach which I've used in Schmoo to demonstrate the technique, is to use the STGM_CONVERT flag.

When Schmoo sees a non-compound file when loading it calls StgCreateDocfile passing STGM_CONVERT instead of STGM_CREATE. The STGM_CONVERT flag causes OLE to open the file as if it were a storage object containing a single stream named "CONTENTS".

```
LPSTORAGE   pIStorage;
HRESULT     hr;

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

if (FAILED(hr))
```

```
{
if (STG_E_ACCESSDENIED==GetScode(hr))
   [Try loading the file using traditional file I/O]
}
```

If this operation succeeds, the HRESULT returned from StgCreateDocfile will contain STG_S_CONVERTED which is not equal to NOERROR but which does not mean an error occurred. Therefore we use the FAILED macro to actually test success.

NOTE: When using StgCreateDocfile for conversion I purposely passed STGM_TRANSACTED | STGM_READWRITE, but never bother to commit anything. The semantics of STGM_CONVERT in StgCreateDocfile mean "convert the file now." If you use STGM_DIRECT in this case the old file will be immediately overwritten with a compound file. By specifying STGM_TRANSACTED you create the conversion *in memory only* leaving the disk image unaffected. If you called IStorage::Commit on such a transacted storage you would then change the actual file on the disk. Since conversion is a potential write operation, you must specify at least STGM_WRITE along with STGM_CONVERT. If the file is marked as read-only, however, StgCreateDocfile will fail with STG_E_ACCESSDENIED. In such a situation you can default to loading the file with old code that only requires read-only access.

Once we have opened this old file as a compound file, we have a storage object (an IStorage pointer) through which we can treat the file as if it were a compound file. As shown in Listing 5-1, Schmoo passes an IStorage from either StgCreateDocfile or StgOpenStorage to the same code in the Polyline to read its data from the stream. It is not a consequence that Schmoo's version 2.0 file format is a storage containing a single stream named "CONTENTS"–that naming allows the Polyline to read the data from the stream regardless of how it actually lives on the file system.

Streams vs. Files

The "Stream Objects and IStream Interface" section above attempted to show that there is a strong parallel between traditional file I/O functions (in both the Windows API and the C run-time library) and the member functions of the IStream interface. In the Polyline implementation we can see these similarities by comparing the old ::ReadFromFile function used in the Chapter 2 implementation of Schmoo to the ::ReadFromStorage function used exclusively in the new version. (Note that ::ReadFromFile is still used in the new version of Schmoo when the file is marked read-only, restricting our use of STGM_CONVERT.) The two functions ::ReadFromFile and ::ReadFromStorage are shown below side-by-side to illustrate the utter similarities between the two implementations:

```
LONG CPolyline::ReadFromFile
   (LPSTR pszFile)
   {
   OFSTRUCT       of;
   HFILE          hFile;
   POLYLINEDATA   pl;
   UINT           cb=-1;
   UINT           cbExpect=0;



   if (NULL==pszFile)
      return POLYLINE_E_READFAILURE;

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




   if (HFILE_ERROR==hFile)
      return POLYLINE_E_READFAILURE;

   cb=_lread(hFile, (LPSTR)&pl
      , 2*sizeof(WORD));

   if (2*sizeof(WORD)!=cb)
      {
      _lclose(hFile);
      return POLYLINE_E_READFAILURE;
      }


   _llseek(hFile, 0L, 0);

   [Code here to calculate cbExpect
    based on the version number]
```

```
LONG CPolyline::ReadFromStorage
   (LPSTORAGE pIStorage)
   {
   HRESULT        hr;
   LPSTREAM       pIStream;
   POLYLINEDATA   pl;
   ULONG          cb=-1;
   ULONG          cbExpect=0;
   LARGE_INTEGER  li;



   if (NULL==pIStorage)
      return POLYLINE_E_READFAILURE;

   hr=pIStorage->OpenStream("CONTENTS", 0
      , STGM_DIRECT | STGM_READ
      | STGM_SHARE_EXCLUSIVE, 0
      , &pIStream);

   if (FAILED(hr))
      return POLYLINE_E_READFAILURE;

   hr=pIStream->Read((LPVOID)&pl
      , 2*sizeof(WORD), &cb);

   if (FAILED(hr) || 2*sizeof(WORD)!=cb)
      {
      pIStream->Release();
      return POLYLINE_E_READFAILURE;
      }

   LISet32(li, 0);
   pIStream->Seek(li, STREAM_SEEK_SET, NULL);

   [Code here to calculate cbExpect
    based on the version number]
```

```
cb=_lread(hFile, (LPSTR)&pl
   , cbExpect);
_lclose(hFile);

if (cbExpect!=cb)
   return POLYLINE_E_READFAILURE;

DataSet(&pl, TRUE, TRUE);
return MAKELONG(pl.wVerMin
   , pl.wVerMaj);
}
```

```
hr=pIStream->Read((LPVOID)&pl
   , cbExpect, &cb);
pIStream->Release();

if (cbExpect!=cb)
   return POLYLINE_E_READFAILURE;

DataSet(&pl, TRUE, TRUE);
return MAKELONG(pl.wVerMin
   , pl.wVerMaj);
}
```

The first difference is that a call to OpenFile has been replaced with a call to OpenStream; we now treat the storage in which the data lives just like we treated the file system before. All of the old file I/O functions called with the file handle are then replaced by calls through the IStream pointer instead. A second general difference is the calling convention imposed by the use of IStream members. Remember that interface members generally return an HRESULT thereby requiring you to pass pointers to variable in which those functions return additional information. So instead of a function like _lread that returns the number of bytes read, we have IStream::Read that returns an HRESULT and fills another variable with the number of bytes read. Other than that, these two functions are semantically equivalent.

The code above also shows a difference in seeking within a file as opposed to a stream. The structured storage definition of IStream allows storages and streams to contain up to 2^{64} addressable bytes of data. Since a stream can be that large, you have to use the LARGE_INTEGER type to pass the seek offset which is that unfamiliar code in ::ReadFromStorage above:

```
LARGE_INTEGER   li;

LISet32(li, 0);
pIStream->Seek(li, STREAM_SEEK_SET, NULL);
```

A LARGE_INTEGER has two fields: a DWORD LowPart field and a LONG (signed) HighPart field. The LISet32 macro sets LowPart to the value specified and performs sign-extension into HighPart. There is also a ULARGE_INTEGER which is composed of two DWORD parts with an associated ULISet32 macro. The third parameter to IStream::Seek above could be a ULARGE_INTEGER which receives the seek offset in the stream before the call. Passing a NULL simply means you're not interested.

The ::CreateStream, ::OpenStream, ::Read, ::Write, and ::Seek members of IStorage and IStream are the most commonly used, and the most performance optimized functions in the entire compound file implementation of OLE 2.0. For simple storage uses such as Schmoo, these with only a few others, like IStream::Seek, may be all you need. Other members are used for more complicated storage models.

Complex Compound Files: Patron

The study of English grammar defines a number a sentence structures. A simple sentence express one idea, like "The rabbit sat in his form." Compound sentences expresses more than one independent idea, such as "The rabbit sat in his form and the photographer set up his camera." In such a sentence there is only a vague notion of concurrency, but no hard evidence. A complex sentence defines such a relationship as in "The rabbit sat in his form *while* the photographer set up his camera." A complex-compound sentence is more on the order of "Although the rabbit had trepidation about most humans, it calmly sat in its form and the photographer continued to set up his camera."

If we can relate ideas to elements in a OLE 2.0's structured storage model, we can see how these descriptions of sentences apply to compound files. The simplest use of the model is writing a single stream into a file using CreateStreamOnFile. When you use a root storage that contains one stream, you've made a compound file (sentence) where the two elements are related mostly by virtue of them living in the same place at the same time. When we start adding more streams in the root storage, we make things more complex–the meaning of the data in one stream may be defined partially or completely by the context of the data in another stream. As we get even more complex, we start adding sub-storages alongside these streams that generally don't need any dependency on the streams but do occupy space in the same file. This is the notion of complex-compound files.

For Patron we'll implement exactly that storage model shown in Figure 5-5 which is the same as that shown in Figure 5-3 but without the page headers streams or tenants storages since we don't yet have the

Figure 5-5: Exact layout of Patron's compound files in this chapter.

capability to create tenants. Each Patron file is a root storage underneath which lives a stream containing the device configuration (printer parameters) and a stream containing the list of pages in the file by a DWORD identifier where the list of IDs stored in the page list defines which ID is page 1, page 2, etc.. Each page is then stored as a sub-storage itself below the root storage where the name of the page storage is "Page " appended with the ID. At this time these storages themselves will not contain any other streams or storages, but will provide the structure in which we can store tenants, either bitmaps and metafiles as we'll see in Chapter 7 or compound document objects as we'll see starting in Chapter 9.

To accommodate file I/O, Patron has undergone some considerable modifications and additions, the more important of which are shown in Listing 5-2. For the most part, Patron follows the sequence of steps described for Schmoo in the previous section with the exception that not everything happens at the same time or in the same place. Patron is also now a component object user, like Schmoo.

Changes made to DOCUMENT.CPP handle opening and saving the compound file, that is, implementing File New, File Open, File Save, and File Save As from the main window's point of view (CPatronDoc::ULoad and CPatronDoc::USave). The document-level code only manages the root storage itself and thus passes that storage to the CPages object that maintains the actual page list. All stream and sub-storage creation happens on the pages level. Still, if you follow the code, you will see that Patron generally creates a root storage, writes streams into it (as well as sub-storages), calls functions like WriteClassStg to identify the file, and calls IStorage::Release to finally close the file.

DOCUMENT.CPP

```
[Other code unaffected]


CPatronDoc::~CPatronDoc(void)
   {
   if (NULL!=m_pPG)
      delete m_pPG;

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

   return;
   }
```

Listing 5-2: The Patron program using Compound Files for its storage.

```
/*
 * CPatronDoc::ULoad
 *
 * Purpose:
 *  Loads a given document without any user interface overwriting the
 *  previous contents of the editor.
 *
 * Parameters:
```

```
 *  fChangeFile     BOOL indicating if we're to update the window title
 *                  and the filename from using this file.
 *  pszFile         LPSTR to the filename to load.  Could be NULL for
 *                  an untitled document.
 *
 * Return Value:
 *  UINT            An error value from DOCERR_*
 */


UINT CPatronDoc::ULoad(BOOL fChangeFile, LPSTR pszFile)
   {
   RECT       rc;
   HRESULT    hr;
   LPSTORAGE  pIStorage;
   CLSID      clsID;
   DWORD      dwMode=STGM_TRANSACTED | STGM_READWRITE |
STGM_SHARE_EXCLUSIVE;

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

      //Mark this as one of our class since we check with ReadClassStg below.
      if (SUCCEEDED(hr))
         WriteClassStg(pIStorage, CLSID_PatronPages);
      }
   else
      {
      hr=StgOpenStorage(pszFile, NULL, dwMode, NULL, 0, &pIStorage);
      }

   if (FAILED(hr))
      return DOCERR_COULDNOTOPEN;

   //Check if this is our type of file and exit if not.
   hr=ReadClassStg(pIStorage, &clsID);

   if (FAILED(hr) || !IsEqualCLSID(clsID, CLSID_PatronPages))
      {
      pIStorage->Release();
      return DOCERR_READFAILURE;
      }
```

```
   //Attempt to create our contained Pages window.
   m_pPG=new CPages(m_hInst);
   GetClientRect(m_hWnd, &rc);

   if (!m_pPG->FInit(m_hWnd, &rc, WS_CHILD | WS_VISIBLE, ID_PAGES, NULL))
      {
      pIStorage->Release();
      return DOCERR_NOFILE;
      }

   if (!m_pPG->FIStorageSet(pIStorage, FALSE, (BOOL)(NULL==pszFile)))
      {
      pIStorage->Release();
      return DOCERR_READFAILURE;
      }

   Rename(pszFile);

   //Do initial setup if this is a new file, otherwise Pages handles things.
   if (NULL==pszFile)
      {
      //Go initialize the Pages for the default printer.
      if (!PrinterSetup(NULL, TRUE))
         return DOCERR_COULDNOTOPEN;

      //Go create an initial page.
      m_pPG->PageInsert(0);
      }

   m_pIStorage=pIStorage;

   FDirtySet(FALSE);
   return DOCERR_NONE;
   }



/*
 * CPatronDoc::USave
 *
 * Purpose:
 *  Writes the file to a known filename, requiring that the user has
 *  previously used FileOpen or FileSaveAs in order to have a filename.
 *
 * Parameters:
```

```
*  uType          UINT indicating the type of file the user requested
*                 to save in the File Save As dialog.
*  pszFile        LPSTR under which to save.  If NULL, use the current name.
*
* Return Value:
*  UINT           An error value from DOCERR_*
*/

UINT CPatronDoc::USave(UINT uType, LPSTR pszFile)
   {
   HRESULT     hr;
   LPSTORAGE   pIStorage;

   //Save or Save As with the same file is just a commit.
   if (NULL==pszFile || (NULL!=pszFile && 0==lstrcmpi(pszFile, m_szFile)))
      {
      WriteFmtUserTypeStg(m_pIStorage, m_cf, PSZ(IDS_CLIPBOARDFORMAT));

      //Insure pages are up to date.
      m_pPG->FIStorageUpdate(FALSE);

      //Commit everyting
      m_pIStorage->Commit(STGC_ONLYIFCURRENT);

      FDirtySet(FALSE);
      return DOCERR_NONE;
      }

   /*
    * When we're given a name, open the storage, creating it new if
    * it does not exist or overwriting the old one.  Then ::CopyTo from
    * the current to the new, ::Commit the new, then ::Release the old.
    */
   hr=StgCreateDocfile(pszFile, STGM_TRANSACTED | STGM_READWRITE
      | STGM_CREATE | STGM_SHARE_EXCLUSIVE, 0, &pIStorage);

   if (FAILED(hr))
      return DOCERR_COULDNOTOPEN;

   WriteClassStg(pIStorage, CLSID_PatronPages);
   WriteFmtUserTypeStg(pIStorage, m_cf, PSZ(IDS_CLIPBOARDFORMAT));

   //Insure all pages are up-to-date.
   m_pPG->FIStorageUpdate(TRUE);

   //This also copies the CLSID we stuff in here on file creation.
```

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

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

   pIStorage->Commit(STGC_ONLYIFCURRENT);

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

   //Make this new storage current
   m_pIStorage=pIStorage;
   m_pPG->FIStorageSet(pIStorage, TRUE, FALSE);

   FDirtySet(FALSE);
   Rename(pszFile);    //Update caption bar.

   return DOCERR_NONE;
   }
```

PAGES.H

```
class __far CPage
   {
   private:
      DWORD       m_dwID;            //Persistent DWORD identifier
      LPSTORAGE   m_pIStorage;       //Sub-storage for this page.

   public:
      CPage(DWORD);
      ~CPage(void);

      DWORD   GetID(void);
      BOOL    FOpen(LPSTORAGE);
      void    Close(BOOL);
      void    Update(void);
      void    Destroy(LPSTORAGE);
   };
```

```
typedef CPage FAR * LPPAGE;



/*
 * Structures to save with the document describing the device
 * configuration and pages that we have.  This is followed by
 * a list of DWORD IDs for the individual pages.
 */

typedef struct __far tagDEVICECONFIG
   {
   DEVMODE     dm;
   char        szDriver[CCHDEVICENAME];
   char        szDevice[CCHDEVICENAME];
   } DEVICECONFIG, FAR * LPDEVICECONFIG;

typedef struct __far tagPAGELIST
   {
   UINT        cPages;
   UINT        iPageCur;
   DWORD       dwIDNext;
   } PAGELIST, FAR *LPPAGELIST;


class __far CPages : public CWindow
   {
   [Existing members omitted from this listing]

      LPSTORAGE   m_pIStorage;          //Root storage o
      //m_hDevMode, m_szDriver, m_szDevice removed

   public:
   [Existing members omitted from this listing]

      BOOL      FIStorageSet(LPSTORAGE, BOOL, BOOL); //Was ::New previously
      BOOL      FIStorageUpdate(BOOL);

   };

typedef CPages FAR * LPCPages;

//Fixed names of streams in the Pages IStorage
#define SZSTREAMPAGELIST        "Page List"
#define SZSTREAMDEVICECONFIG    "Device Configuration"
```

PAGES.CPP

```
CPages::CPages(HINSTANCE hInst)
    : CWindow(hInst)
    {
   [Other code omitted from listing]

    m_pIStorage=NULL;
    return;
    }


CPages::~CPages(void)
    {
    //Insure memory is cleaned up in list, and do final IStorage::Release
    FIStorageSet(NULL, FALSE, FALSE);

   [Other code omitted from listing]

    return;
    }



/*
 * CPages::FIStrorageSet
 *
 * Purpose:
 *  Provides the document's IStorage to the pages for its own uses.
 *  If this is a new storage, then we initalize it with streams we
 *  want to always exists.  If this is an open, then we create
 *  our page list from the PageList string we wrote before.
 *
 * Parameters:
 *  pIStorage       LPSTORAGE to the new or opened storage.  If this is
 *                  NULL then we just clean up and exit.
 *  fChange         BOOL indicating is this was a Save As operation
 *                  meaning that we have the structure already, we
 *                  just need to change our value of m_pIStorage.
 *  fInitNew        BOOL indicating if this is a new storage or one
 *                  opened from a previous save.
 */
```

```
BOOL CPages::FIStorageSet(LPSTORAGE pIStorage, BOOL fChange, BOOL fInitNew)
   {
   DWORD          dwMode=STGM_DIRECT | STGM_READWRITE |
STGM_SHARE_EXCLUSIVE;
   HRESULT         hr;
   LPPAGE          pPage;
   BOOL           fRet=FALSE;
   ULONG           cbRead;
   PAGELIST        pgList;
   LPSTREAM        pIStream;
   LPMALLOC        pIMalloc;
   LPDWORD         pdwID;
   UINT           i;

   //If we're just changing saved roots, just open the current page again
   if (fChange)
      {
      if (NULL==pIStorage)
         return FALSE;

      m_pIStorage->Release();
      m_pIStorage=pIStorage;
      m_pIStorage->AddRef();

      FPageGet(m_iPageCur, &pPage, TRUE);
      return TRUE;
      }

   //On new or open, clean out whatever it is we have.
   for (i=0; i < m_cPages; i++)
      {
      if (FPageGet(i, &pPage, FALSE))
         delete pPage;
      }

   SendMessage(m_hWndPageList, LB_RESETCONTENT, 0, 0L);

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

   m_pIStorage=NULL;

   //If we're just cleaning up, then we're done.
   if (NULL==pIStorage)
      return TRUE;
```

```
   m_pIStorage=pIStorage;
   m_pIStorage->AddRef();

   //If this is a new storage, create the streams we require
   if (fInitNew)
      {
      //Page list header.
      hr=m_pIStorage->CreateStream(SZSTREAMPAGELIST, dwMode | STGM_CREATE
         , 0, 0, &pIStream);

      if (FAILED(hr))
         return FALSE;

      pIStream->Release();

      //Device Configuration
      hr=m_pIStorage->CreateStream(SZSTREAMDEVICECONFIG, dwMode |
STGM_CREATE
         , 0, 0, &pIStream);

      if (FAILED(hr))
         return FALSE;

      pIStream->Release();
      return TRUE;
      }

   /*
    * We're opening an existing file:
    *  1)  Configure for the device we're on
    *  2)  Read the Page List and create page entries for each.
    */

   ConfigureForDevice();

   //Read the page list.
   hr=m_pIStorage->OpenStream(SZSTREAMPAGELIST, NULL, dwMode, 0,
&pIStream);

   if (FAILED(hr))
      return FALSE;

   if (SUCCEEDED(CoGetMalloc(MEMCTX_SHARED, &pIMalloc)))
      {
      pIStream->Read((LPVOID)&pgList, sizeof(PAGELIST), &cbRead);
      m_cPages  =pgList.cPages;
```

```
    m_iPageCur=pgList.iPageCur;
    m_dwIDNext=pgList.dwIDNext;

    fRet=TRUE;
    cbRead=pgList.cPages*sizeof(DWORD);

    if (0!=cbRead)
       {
       pdwID=(LPDWORD)pIMalloc->Alloc(cbRead);

       if (NULL!=pdwID)
          {
          pIStream->Read((LPVOID)pdwID, cbRead, &cbRead);

          for (i=0; i < m_cPages; i++)
             fRet &=FPageAdd(-1, *(pdwID+i), FALSE); //-1==end of list

          pIMalloc->Free((LPVOID)pdwID);
          }
       }

    pIMalloc->Release();
    }

 pIStream->Release();

 if (!fRet)
    return FALSE;

 FPageGet(m_iPageCur, &pPage, TRUE);

 InvalidateRect(m_hWnd, NULL, FALSE);
 UpdateWindow(m_hWnd);

 return TRUE;
 }



BOOL CPages::FIStorageUpdate(BOOL fCloseAll)
 {
 LPPAGE         pPage;
 LPSTREAM       pIStream;
 LPMALLOC       pIMalloc;
 LPDWORD        pdwID;
 ULONG          cb;
```

```
HRESULT        hr;
PAGELIST       pgList;
BOOL           fRet=FALSE;
UINT           i;

//We only need to possibly close the current page--nothing else is open.
if (FPageGet(m_iPageCur, &pPage, FALSE))
   {
   pPage->Update();

   if (fCloseAll)
      pPage->Close(FALSE);
   }

//We don't hold anything else open, so we can just write the page list.
hr=m_pIStorage->OpenStream(SZSTREAMPAGELIST, NULL, STGM_DIRECT
   | STGM_READWRITE | STGM_SHARE_EXCLUSIVE, 0, &pIStream);

if (FAILED(hr))
   return FALSE;

if (SUCCEEDED(CoGetMalloc(MEMCTX_SHARED, &pIMalloc)))
   {
   pgList.cPages=m_cPages;
   pgList.iPageCur=m_iPageCur;
   pgList.dwIDNext=m_dwIDNext;

   pIStream->Write((LPVOID)&pgList, sizeof(PAGELIST), &cb);

   cb=m_cPages*sizeof(DWORD);
   pdwID=(LPDWORD)pIMalloc->Alloc(cb);

   if (NULL!=pdwID)
      {
      for (i=0; i < m_cPages; i++)
         {
         FPageGet(i, &pPage, FALSE);
         *(pdwID+i)=pPage->GetID();
         }

      pIStream->Write((LPVOID)pdwID, cb, &cb);
      pIMalloc->Free((LPVOID)pdwID);
      fRet=TRUE;
      }
   pIMalloc->Release();
   }
```

```
   pIStream->Release();

   return fRet;
   }



UINT CPages::PageInsert(UINT uReserved)
   {
   LPPAGE      pPage;

   if (0!=m_cPages)
      {
      //Close the current page, committing changes.
      if (!FPageGet(m_iPageCur, &pPage, FALSE))
         return 0;

      pPage->Close(TRUE);
      }

   [Other code omitted from listing]
   }

UINT CPages::PageDelete(UINT uReserved)
   {
   LPPAGE      pPage;

   if (!FPageGet(m_iPageCur, &pPage, FALSE))
      return -1;

   //Delete the page in both the storage and in memory.
   SendMessage(m_hWndPageList, LB_DELETESTRING, m_iPageCur, 0L);

   pPage->Destroy(m_pIStorage);

   [Other code omitted from listing]
   }


UINT CPages::CurPageSet(UINT iPage)
   {
   UINT    iPagePrev=m_iPageCur;
   LPPAGE  pPage;

   //Close the old page committing changes.
```

```
   if (!FPageGet(iPagePrev, &pPage, FALSE))
      return -1;

   pPage->Close(TRUE);

   [Code to adjust page number omitted from listing]

   //Open the new page.
   FPageGet(m_iPageCur, &pPage, TRUE);

   InvalidateRect(m_hWnd, NULL, FALSE);
   UpdateWindow(m_hWnd);
   return iPagePrev;
   }



BOOL CPages::DevModeSet(HGLOBAL hDevMode, HGLOBAL hDevNames)
   {
   LPDEVNAMES      pdn;
   LPSTR           psz;
   DEVICECONFIG    dc;
   LPDEVMODE       pdm;
   LPSTREAM        pIStream;
   HRESULT         hr;
   ULONG           cbWrite;
   BOOL            fRet=FALSE;

   if (NULL==hDevMode || NULL==hDevNames)
      return FALSE;

   hr=m_pIStorage->OpenStream(SZSTREAMDEVICECONFIG, 0, STGM_DIRECT
      | STGM_WRITE | STGM_SHARE_EXCLUSIVE, 0, &pIStream);

   if (FAILED(hr))
      return FALSE;

   pdm=(LPDEVMODE)GlobalLock(hDevMode);

   if (NULL!=pdm)
      {
      dc.dm=*pdm;
      GlobalUnlock(hDevMode);
      psz=(LPSTR)GlobalLock(hDevNames);
```

```
      if (NULL!=psz)
         {
         pdn=(LPDEVNAMES)psz;
         lstrcpy(dc.szDriver, psz+pdn->wDriverOffset);
         lstrcpy(dc.szDevice, psz+pdn->wDeviceOffset);

         pIStream->Write((LPVOID)&dc, sizeof(DEVICECONFIG), &cbWrite);
         GlobalUnlock(hDevNames);
         fRet=TRUE;
         }
      }

   if (!fRet)
      return FALSE;

   GlobalFree(hDevNames);
   GlobalFree(hDevMode);

   return ConfigureForDevice();
   }


HGLOBAL CPages::DevModeGet(void)
   {
   HGLOBAL     hMem;
   LPDEVMODE   pdm;
   ULONG       cbRead;
   LPSTREAM    pIStream;
   HRESULT     hr;

   hr=m_pIStorage->OpenStream(SZSTREAMDEVICECONFIG, 0, STGM_DIRECT
      | STGM_READ | STGM_SHARE_EXCLUSIVE, 0, &pIStream);

   if (FAILED(hr))
      return FALSE;

   hMem=GlobalAlloc(GHND, sizeof(DEVMODE));

   if (NULL!=hMem)
      {
      pdm=(LPDEVMODE)GlobalLock(hMem);
      pIStream->Read((LPVOID)pdm, sizeof(DEVMODE), &cbRead);
      GlobalUnlock(hMem);
      }

   pIStream->Release();
```

```
   return hMem;
   }

BOOL CPages::ConfigureForDevice(void)
   {
   POINT          ptOffset, ptPaper;
   RECT           rc;
   HDC            hDC;
   DEVICECONFIG    dc;
   HRESULT         hr;
   LPSTREAM        pIStream;
   ULONG          cbRead;

   //Read the DEVMODE and driver names from our header stream.
   hr=m_pIStorage->OpenStream(SZSTREAMDEVICECONFIG, 0, STGM_DIRECT
      | STGM_READ | STGM_SHARE_EXCLUSIVE, 0, &pIStream);

   if (FAILED(hr))
      return FALSE;

   pIStream->Read((LPVOID)&dc, sizeof(DEVICECONFIG), &cbRead);
   pIStream->Release();

   hDC=CreateIC(dc.szDriver, dc.szDevice, NULL, &dc.dm);

   [Other code omitted from listing]
   }


BOOL CPages::FPageGet(UINT iPage, LPPAGE FAR *ppPage, BOOL fOpen)
   {
   if (NULL==ppPage)
      return FALSE;

   if (sizeof(LPPAGE)==SendMessage(m_hWndPageList, LB_GETTEXT, iPage
      , (LONG)(LPVOID)ppPage))
      {
      if (fOpen)
         (*ppPage)->FOpen(m_pIStorage);

      return TRUE;
      }

   return FALSE;
   }
```

```
BOOL CPages::FPageAdd(UINT iPage, DWORD dwID, BOOL fOpenStorage)
   {
   LPPAGE      pPage;
   LRESULT     lr;

   pPage=new CPage(dwID);

   if (NULL==pPage)
      return FALSE;

   if (fOpenStorage)
      pPage->FOpen(m_pIStorage);

   if (0xffff==iPage)
      iPage--;

   //Now try to add to the listbox.
   lr=SendMessage(m_hWndPageList, LB_INSERTSTRING, iPage+1, (LONG)pPage);

   if (LB_ERRSPACE==lr)
      {
      if (fOpenStorage)
         pPage->Close(FALSE);

      delete pPage;
      return FALSE;
      }

   return TRUE;
   }
```

PAGE.CPP

```
/*
 * PAGE.CPP
 * Modifications for Chapter 5
 *
 * Implementation of the CPage class which is a simple structure.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */
```

```
#include "patron.h"


CPage::CPage(DWORD dwID)
   {
   m_dwID     =dwID;
   m_pIStorage=NULL;
   return;
   }

CPage::~CPage(void)
   {
   Close(FALSE);
   return;
   }



DWORD CPage::GetID(void)
   {
   return m_dwID;
   }



BOOL CPage::FOpen(LPSTORAGE pIStorage)
   {
   BOOL        fNULL=FALSE;
   HRESULT     hr=NOERROR;
   DWORD       dwMode=STGM_TRANSACTED | STGM_READWRITE |
STGM_SHARE_EXCLUSIVE;
   char        szTemp[32];

   if (NULL==m_pIStorage)
      {
      fNULL=TRUE;

      if (NULL==pIStorage)
         return FALSE;

      /*
       * Attempt to open the storage under this ID.  If there is none, then
       * create it.  In either case we end up with an IStorage that we
       * either save in pPage or release.
       */
```

```
        wsprintf(szTemp, "Page %lu", m_dwID);

        hr=pIStorage->OpenStorage(szTemp, NULL, dwMode, NULL, 0, &m_pIStorage);

        if (FAILED(hr))
            hr=pIStorage->CreateStorage(szTemp, dwMode, 0, 0, &m_pIStorage);
        }
    else
        m_pIStorage->AddRef();

    if (FAILED(hr))
        {
        if (fNULL)
            m_pIStorage=NULL;

        return FALSE;
        }

    return TRUE;
    }



void CPage::Close(BOOL fCommit)
    {
    if (NULL==m_pIStorage)
        return;

    if (fCommit)
        Update();

    if (0==m_pIStorage->Release())
        m_pIStorage=NULL;

    return;
    }



void CPage::Update(void)
    {
    if (NULL!=m_pIStorage)
        m_pIStorage->Commit(STGC_ONLYIFCURRENT);

    return;
    }
```

```
void CPage::Destroy(LPSTORAGE pIStorage)
   {
   char       szTemp[32];

   if (NULL!=pIStorage)
      {
      wsprintf(szTemp, "Page %lu", m_dwID);
      pIStorage->DestroyElement(szTemp);
      }

   return;
   }
```

The Root Storage and Temporary Files

Patron always keeps the root storage open: if the file is untitled (from a File New command), Patron uses a temporary compound file created by passing a NULL to StgCreateDocfile. Note you should use the STGM_DELETEONRELEASE flag with temporary files:

```
hr=StgCreateDocfile(NULL, STGM_TRANSACTED | STGM_READWRITE | STGM_CREATE
   | STGM_SHARE|EXCLUSIVE | STGM_DELETEONRELEASE, 0, &m_pIStorage);
```

If the file was already exists on the disk (opened with the File Open command), Patron opens it with StgOpenStorage and keeps that storage open. These two functions are called from CPatronDoc::ULoad in DOCUMENT.CPP depending on whether or not the user chose File New or File Open.

Keeping a storage open in this manner is as expensive as keeping an open file using traditional file I/O: the Windows 3.1 implementation of compound files uses a file handle to each open root storage, although all sub-storages and streams require no additional MS-DOS resources. If you can tolerate the cost of keeping the root storage open, you have the benefit of keeping anything else open with the only cost being that of memory.

This temporary storage will be created with a pseudo-random name in the directory of your TEMP environment variable. If you specify STGM_DELETEONRELEASE then the OLE 2.0 libraries will keep the files cleaned out. If, however, your application crashes before releasing a temporary file, or you fail to call IStorage::Release the final time, then you will end up with a number of these orphaned temp files.

Temporary files are also interesting in File Save As cases, described below in "File Save As Operations."

Managing Sub-Storages

As mentioned above, Patron manages a sub-storage for each page in the overall document. This, of course, means somewhat more code complexity for the cases of creating and destroying pages.

Whenever Patron creates a new page in the document, it calls IStorage::CreateStorage using a name of "Page xx" where xx is the ASCII for a DWORD page index. The code shown here is taken from PAGE.CPP in the function CPage::FOpen, with either creates a new sub-storage for an new page or opens the sub-storage for an existing page using IStorage::OpenStorage:

```
BOOL CPage::FOpen(LPSTORAGE pIStorage)
   {
   HRESULT hr=NOERROR;
   DWORD   dwMode=STGM_TRANSACTED | STGM_READWRITE | STGM_SHARE_EXCLUSIVE;
   char    szTemp[32];

   //m_dwID is the page ID (not the page number) stored persistently.
   wsprintf(szTemp, "Page %lu", m_dwID);

   hr=pIStorage->OpenStorage(szTemp, NULL, dwMode, NULL, 0, &m_pIStorage);

   if (FAILED(hr))
      hr=pIStorage->CreateStorage(szTemp, dwMode, 0, 0, &m_pIStorage);
```

The page identifier, m_dwID is assigned when creating a new page. Each Patron document creates the first page with an ID of zero and increments the ID for each page thereafter. The next usable ID us stored persistently in the file, so the IDs continue to increment throughout the life of the document. The ordering of the pages is written into another stream as a sequence of these IDs that reflect the positions in the document where they were created. IDs are not recycled when a page is destroyed, but the DWORD counter would only overflow if you sat here and created one page every second until the Gregorian year 2129. I desperately hope this software is obsolete by then!

Speaking of destroying a page, this operation required a call to IStorage::DestroyElement to counter the IStorage::CreateStorage in the code above. The function CPage::Destroy takes care of this in Patron:

```
//pIStorage is the document's root storage.
void CPage::Destroy(LPSTORAGE pIStorage)
  {
  char      szTemp[32];

  if (NULL!=pIStorage)
    {
    wsprintf(szTemp, "Page %lu", m_dwID);
    pIStorage->DestroyElement(szTemp);
    }

  return;
  }
```

The CPage class is a simple structure used to manage the open storage and the identifier for a page. Since it maintains it's ID, we keep the code to generate a page number from that ID down in the CPage implementation.

Multi-Level Commits

Patron opens its files in transacted mode, so it must call IStorage::Commit on its open storage before closing the document. Until that time, all changes, including new and deleted pages, are, what a ***HACK***[1] would term "fleeting." In other words, all these changes are only stored in memory such that turning off your machine would only commit them to the Great-Big-Bit-Bucket in the sky.

Of course, we would like to commit those changes to the actual disk file instead during a File Save operation. A File Save As operation is a little different and is treated in the next section. To save changes to a file we opened previously we only have to call IStorage::Commit on the root to send all changes to the disk. The catch is, however, that we have to make sure that every sub-storage opened in transacted mode has also been committed.

Whenever a change is made to a sub-storage (either any modification to a STGM_DIRECT storage or a ::Commit on an STGM_TRANSACTED storage) those changes are only published to the immediate storage in which it's contained. That is, if I have storages A, B, and C in a compound file:

and I change storage C, only storage B is cognizant of those changes:

If storage B is direct then it immediately publishes the change in C to storage A. If storage A is direct, then those changes are immediately written to disk. If storage B is transacted, however, changes in C are not published to storage A until storage B is committed, just as changes to storage B (including commits) are not published to the actual disk until storage A is committed.

So the whole trick with multiple-levels of transacted storages is to walk through the whole chain making sure everything that needs committing gets it, then commit the root storage to actually save all the changes permanently. Patron handles this by telling the current page to commit, writing any streams that might require modification, and the committing the root storage. Patron does not worry about committing any other page because as the user switches between pages, Patron calls IStorage::Commit and IStorage::Release on the current page before opening the next page. This means that when we want to commit the outermost storage, we only have to commit the current page instead of walking all pages to commit each in turn.

A commit can happen in a few different ways based on one of four flags passed to IStorage::Commit:

1A slave who would hold laurels wreath over the head of a victorious general head during a triumph in imperial Rome. While the victor would bask in applause, the slave would whisper in his ear, "All glory is fleeting." ***HACK: I think I could watch Patton again where he mentions this at the end of the movie. Otherwise there's a good ol' Star Trek where Mr. Spock uses the word :)***

STGC_DEFAULT	No special semantics; just commit changes.
STGC_ONLYIFCURRENT	In a file-sharing scenario, this flag prevents one process from overwriting changes made in another process since the first process opened the storage. If our changes are not current, then IStorage::Commit returns the code STG_E_NOTCURRENT on which you can attempt to merge changes or inform the user to take appropriate action.
STGC_OVERWRITE	Attempts to overwrite the entire existing file resulting in smaller file sizes. However, this is somewhat prone to failure due to memory constraints, and if such a failure occurs, the actual disk file will generally be in Limbo: neither old nor new versions. This flag is not recommended for general use.
STGC_DANGEROUSLYCOMMITMERELYTODISKCACHE	OLE 2.0 designers never said they couldn't be verbose. This flag to ::Commit allows compound files to write the changes to an existing disk cache, such as Microsoft Windows' SmartDrv 4.0 instead of writing to the cache then forcing a flush of that cache (Int 21h Fn68h). Default behavior, in order to get very robust saves, is to flush the cache immediately to avoid risky disk buffering. If you want better performance for saves, you can risk using this flag, which will not be any worse that using traditional file I/O as it stands today.

A sibling function to IStorage::Commit is IStorage::Revert which dumps all the changes kept in memory for a transacted storage made since the last ::Commit. This affects all open sub-storages and streams as well, . In addition, if you ::Release and IStorage without committing it first you imply ::Revert, that is, you discard changes. Therefore it is not necessary to call ::Revert on an operation like File Close.

File Save As Operations

The use of temporary files have the interesting problem of getting all the data from that temporary file into another file with a user-specified name, such as when executing a File Save As command. Applications typically do this by creating the new file and copying the data from the temporary file into this new file, deleting the temporary file at the end.

Compound files are no different. As we've seen already you can use StgCreateDocfile to create a temporary compound file for you, as Patron does. Then question is how to copy the data from the temporary file into the final destination file. One very painful way would b to somehow load all the streams and storages into memory and write them out to the new storage. Ouch. I'm not sure about you, but I would not enjoy taking a few years to write this code.

Anticipating this much change, the OLE 2.0 architects kindly included IStorage::CopyTo the structured storage model which takes whatever data is in one storage object, regardless of whether it's a root or sub-storage, and copies that data into another storage. Patron uses this when saving any file under a new name, which encompasses renaming temp files as well as doing a Save As on an already known file.

The first step is to open the new destination file using StgCreateDocfile and the STGM_CREATE flag; use of this flag means create the file if it's not already there, and if it is there, overwrite it completely. Most applications use this method in Save As operations. Once you have this new storage opened, call the ::CopyTo in your currently open storage, passing the new destination storage as the fourth parameter:

```
m_pIStorage->CopyTo(NULL, NULL, NULL, pIStorage);
```

The other three parameters have to do again, with stream exclusion and STGM_PRIORITY which is not covered in this book. Remember also that if you opened the new storage with STGM_TRANSACTED you must call ::Commit to actually save the changes to the new disk file.

When you have changed the file in which your current data lives, you generally want to keep that new file open as the active document. For this reason you should close all sub-storages and streams in the original storage before ::CopyTo and reopen them later in the context of the new root storage. Otherwise you might end up talking to the wrong file, or a file that no longer exists! Not good. Patron handles this through it's policy of only keeping the current page open (which, as we have seen, cleans up the commit procedure somewhat) so that when it does copy from one storage to another it only has to close the current page, do

the ::CopyTo, then switch root storages and reopen the page. When designing your use of compound files, keep this in mind.

Low-Memory Save As Operations

When memory is low a typical Save As operation as described above might fail. So you are left with a bunch of uncommitted changes to the file you originally opened.

In such a situation you want to be able to take all uncommitted changes that live in memory and all unchanged part of the storage that still live on disk and write them to a new storage without taking up any more memory. In other words, we want to make a copy of the original disk file to the new file, then commit the changes into that new file.

A special interface called IRootStorage supports just this functionality. You can obtain a pointer to this interface by calling IStorage::QueryInterface with IID_IRootStorage on a storage object from StgOpenStorage or StgCreateDocfile. IRootStorage has one member function: SwitchToFile(LPSTR pszNewFile) where pszNewFile is the name of the new file to associate with the storage object. This effectively makes a disk copy of the original file and internally associates your IStorage object with that new file. You may now call IStorage::Commit to save changes to that new file. ***Preview Note: Patron does not use this, at least not yet.***

Streams as Memory Structures

A great use for a stream object is to use it for run-time management of certain structures that eventually have to end up in your disk file. These structures are those that can be persistently saved, that is, contain no pointers or other references to values only determined at run-time. Since you will need to write some of these structures to disk anyway, it makes sense to keep them in the file in the form of a stream instead of duplicating that structure in memory. Instead of having to worry about saving that structure to disk when the time comes you can just commit since that structure is already in a stream inside your storages.

The best candidates for this sort of treatment are structures that define a configuration for your application that is not likely to change often, since reading and writing a stream is considerably slower than performing quick pointer dereferences in memory. Some examples are a LOGFONT structure that describes the current font you are using or a DEVMODE structure that defines the printer setup for the document.

Patron saves the latter structure, DEVMODE, in a stream called "Device Configuration" that lives off the document's root storage. When a new document is created this stream is filled with the configuration of the default printer. When the user later chooses Printer Setup, Patron first reads the contents of this stream to recreate a DEVMODE structure to pass to the common dialog function PrintDlg. When PrintDlg returns, Patron writes the new configuration to the stream and reinitializes the display for the new parameters. Since the data lives in a stream, Patron doesn't need to perform any other steps during a File Save operation.

Streams are a very powerful for managing structures, especially those that change size frequently. A common problem with memory structures is that you have to continually reallocate them as your data grows, and reallocation is always a little more code that we'd like. A stream, on the other hand, will expand itself to accommodate a write beyond its current boundaries. In other words, the reallocation code we all hate to write is hidden down in the stream implementation.

Standard Summary Information Property Set

Preview Note: Contents of this section pending code for it and decision on how to fit property sets into this chapter. Since property sets are a potentially big topic, I don't know if one section here may suffice. What optimally will happen is that there is some other release of code we can take advantage of here, either by showing how to use it or maybe providing a dialog box that collects the information and writes it to a storage.

Other OLE 2.0 Technologies and Structured Storage

OLE 2.0 makes much more use of structured storage and compound files than we have so far exercised in this chapter. As we'll see in Chapter 6, structured storage can be used as a data transfer medium as much as global memory is today. The later chapters that deal with compound documents use structured storage to allow compound document objects to write themselves directly into a storage object provided by the container. If the object is given a storage that exists already within the container's disk file, then the object is saving itself *directly* to that file and has full incremental access to that structure since the container gives the object a storage object for that object's exclusive use.

The more general question that various C++ class libraries (al á MFC) attempt to address is how to serialize any arbitrary object so as to save it in a file. The lowest-level MFC base object class called CObject has, among others, two members functions called ::IsSerializable and ::Serialize. ::IsSerializable tells the caller if a call to ::Serialize will actually do anything. If the answer is positive, then the user can call ::Serialize to save the object's persistent data to a file (a object called an 'archive' in MFC). As we had mentioned when discussing QueryInterface, this mechanism does not tightly couple the questions of whether the object can serialize (CObject::IsSerializable) with the actual function of serialization (CObject::Serialize). The object cannot prevent its user from calling ::Serialize unexpectedly.

OLE 2.0 objects, be they component objects, data objects, compound document object, what have you, have more control. They to will answer the question "can you persistently save yourself" and if the answer is positive they will provide the means to accomplish such an operation. The means are expressed in three interfaces called IPersistStorage, IPersistStream, and IPersistFile, providing the functions through which an object's user can tell the object to save to an storage object, a stream object, or to a file given the filename.

To ask the object if it can serialize to one of these types of elements you must QueryInterface that object for IID_IPersistStorage, IID_IPersistStream, or IID_IPersistFile, which says "Hey object! Can you serialize yourself to an *xx*?" If the object says no, it returns no interface and an error. If it answers positive, then it returns the interface pointer through which you can ask it to perform the function of serialization. Only when the object supports this functionality are you allowed to even thing about calling it.

The DLL implementation of Polyline discussed in Chapter 4 had two functions in its custom IPolyline interface: ::ReadFromFile and ::WriteToFile, and the Component Schmoo program used these functions to essentially serialize the Polyline object. For this chapter I have eliminated the two file-oriented functions in IPolyline and instead implemented the IPersistStorage interface on the Polyline object itself–so now this object supports multiple interfaces. The implementation is described in sections below. I chose IPersistStorage for the object because Component Schmoo for this chapter used compound files for its file I/O and when we turn Polyline into a compound document object in Chapter 10 we will need an implementation of IPersistStorage. The IPersistFile interface is used for servicing linked object, so while I'll mention it here we won't discuss any implementation until Chapter 13. The IPersistStream is not used very often in OLE 2.0 applications, so this book doesn't ever have a need to implement it.

The remainder of this section is split into three parts. The first describes the IPersistStorage, IPersistStream, and IPersistFile interfaces. The second details how an application like Component Schmoo *uses* the IPersistStorage interface, which applies to understanding OLE 2.0 container applications. The last section discusses the implementation of the IPersistStorage on the Polyline object which will provide a good foundation for compound document work in later chapters. What is shown here are the basics of how to use and implement objects that know how to save themselves, but there will be changes when compound documents are involved.

NOTE: The IPersist, IPersistStorage, and IPersistFile interfaces are defined in OLE2.H. Component Schmoo still uses CoInitialize and CoUninitialize instead of the Ole* variants since use of compound files and the IPersist* interfaces need only the Co* variants.

The IPersistStorage, -Stream, and -File Interfaces

When an object answers the questions "can you save yourself to some element?" it responds with a pointer to one of three IPersist* interfaces. All three interfaces derive from the IPersist interface we discussed and implemented in the Koala object of Chapter 4. IPersist contributes only one member function, GetClassID, to the three interfaces introduced here. All the interfaces then provide information related to serialization as well as the actual capabilities to do so. They should be used if you want to provide serialization capabilities from your own object since they are standard and published interfaces. Note that the member functions shown in the table below apply only to transferring data to an from and storage medium: they imply no user interface and have no direct relationship to File menu commands. Therefore members like ::Load mean "load the data" but do not imply "execute File Open."

IPersistStorage	IPersistStream	IPersistFile	Description
GetClassID	GetClassID	GetClassID	Returns the CLSID of the object. A user can call this function to determine if an object identified by the interface might be able to load a storage, stream, or file marked with another CLSID (such as with WriteClassStg).
IsDirty	IsDirty	IsDirty	Replies whether the object should be saved in its present state, returning the SCODE S_OK if the object *is* dirty and S_FALSE if not.
Load	Load	Load	Instructs the object to load itself from a storage object, stream object, or from a file identified by a filename. Objects implementing IPersistStorage or IPersistStream may AddRef the storage or stream object and hold on to it for incremental access. IPersistStorage objects will only see one ::Load call in its lifetime and excludes the use of ::InitNew.
Save	Save	Save	Instructs the object to save itself to the element passed to this function. For IPersistStorage, this function is also told if the storage object is the same as previously passed to ::Load, in which case the object can perform an incremental save. For IPersistStream the object is told whether or not to reset its dirty flag. For IPersistFile, the object is told if it should consider this saved file the current file or if it should ignore the name and continue to use the file passed to ::Load.
SaveCompleted		SaveCompleted	(Not in IPersistStream) Instructs an object that a call to ::Save is finished. See "A Heavy Dose of Protocol with IPersistStorage" below for more details.
InitNew			IPersistStorage Only: When a brand-new object is initially created the user is contractually obligated to provide an IStorage in which the object can write incremental changes. The object can ignore this call or hold on to the IStorage with an ::AddRef for incremental access. The object can only receive one ::InitNew call in its lifetime and use of this function precludes use of ::Load, that is ::InitNew and ::Load are mutually exclusive.
HandsOffStorage			IPersistStorage Only: Instructs the object to release any kind of reference count it is maintaining on its storage object. The user is contractually obligated to call this only immediately after a ::Save and before

	a ::SaveCompleted.
GetSizeMax	IPersistStream Only: Asks the object to return how large the stream would be if ::Save was called immediately.
GetCurFile	IPersistFile Only: Provides the caller with the current file known to the object.

The only methods to obtain a pointer to any one of these interfaces is to request it in a call to IClassFactory::CreateInstance (CoCreateInstance also) or by asking for it via QueryInterface on some other interface pointer. All three interfaces have built-in marshaling support and are generally used in compound document scenarios: IPersistStorage for embedded objects, IPersistStream for monikers, IPersistFile for linked objects. Choosing to implement one of them or choosing to use an object through any interface carries some responsibility with the benefit of being able to treat most object through the same interfaces. This is, in fact, one of the principles of compound documents, that a container can ask any embedded object to save through an IPersistStorage. For an application like a container, the cost is a little more work but the benefit is the ability to use any compound document object without any reliance on custom interfaces.

A Heavy Dose of Protocol with IPersistStorage

By nature structured storage supports not only the capability for incremental saves but supports full incremental access. This carries with it a few problems since many different agents may, at any given time, have various storages and streams open and uncommitted when the agent controlling the root storage wishes to do a complete save, such as to a new file.

As a basis for our discussion let's assume we have an application in control of a root storage in which lives a sub-storage for an object. The object supports the IPersistStorage interface such that the application can communicate information about storage:

When the application creates a new file it will create a temporary file for the root storage and create the sub-storage for the object. In this case the application is required to call the object's IPersistStorage::InitNew passing the sub-storages IStorage pointer the new . Inside ::InitNew the object may retain the IStorage pointer by calling IStorage::AddRef and saving the pointer in memory. Similarly, when the application opens an existing file it will reopen the sub-storage (with IStorage::OpenStorage on the root) and pass that sub-storage to the object's IPersistStorage::Load in which the object again may retain the pointer. In either case, ::InitNew or ::Load, the object is given an IStorage pointer that it can access incrementally as much as desired throughout the object's lifetime:

Some very simple objects will have no need to retain any IStorage because their data is so small (the Polyline object, for example, has a 106-byte data structure and so maintains it all in memory). Most objects will, however, want to retain the pointer such that they can load as little data as is necessary to operate. Let's call this free-access state normal mode. Once an object is in normal mode, the application is not allowed to again call ::InitNew or ::Load.

Two things may now happen to the object, that is, the application may call *either* ::Save or ::HandsOffStorage. ::Save instructs the object to save its changes (either to its currently held IStorage or to a new one) and enter zombie state (also termed 'no-scribble'). While the object is zombified it may not perform any incremental writes to the storage although it may still read from the storage; most editing operations on this object will, in general, fail. One does not converse well with zombies. When the application will again allow the object to perform incremental writes, it calls the object's ::SaveCompleted

function which allows the object to return to normal state.

When the application wishes to perform a full save it requires that the object is not holding on to an open IStorage, that is, the application cannot rename or delete its root file when an object is holding on to a piece of that file. In such situations the application will call ::HandsOffStorage. If the application calls ::HandsOffStorage without first calling ::Save, the object must shrug its shoulders, heave a heavy sigh, and blindly ::Release its held IStorage. If the storage was opened transacted, this will, of course, discard changes. The application is in control since it determines the access mode of the storage anyway; the object has to trust that the application doesn't want the object saved.

When the application calls ::HandsOffStorage the object enters hands-off state where it cannot read *or* write to a storage by simple fact that it has to ::Release any storage on which it would even attempt such operations. Since the object has no hold on the storage, the application is free to party all over its root storage. When the application has finished partying, it must call ::SaveCompleted on the object which brings it from hands-off mode to normal mode.

The application must always pass an IStorage in the ::SaveCompleted call, regardless of the object's current state. This IStorage must always contains the structure expected by the object since the object now has the right to retain a pointer to this new IStorage. The IStorage may or may not be the same as passed to ::InitNew, ::Load, or ::Save–if the object still has an IStorage after ::Save it should ::Release that pointer and reopen what it needs in the new one passed through ::SaveCompleted.

All of this protocol can be reduced down to simple checklists for both the user of the object and the object itself, which are provided in the following two sections. Note that IPersistFile also has ::SaveCompleted by not ::HandsOffStorage, meaning that you must treat IPersistFile as you would IPersistStorage, ignoring the ::HandsOffStorage implications which obviously do not apply.

Of Component Users and IPersistStorage: Component Schmoo

The last section outlined some of the responsibilities of an object user when dealing with an object through IPersistStorage. The Component Schmoo application is a user of the Polyline object, so since we're replacing custom file I/O members in Polyline with IPersistStorage, CoSchmoo must follow the protocol. CoSchmoo is gradually becoming a container exclusively for Polyline objects and most of the discussion in this section is pertinent to OLE 2.0 container applications. The important code changes made to CoSchmoo are shown in the code fragments below and so full code listing are not provided here.

For convenience, CoSchmoo always retains a pointer to the Polyline's IPersistStorage, first obtaining the pointer via QueryInterface after creation and releasing that pointer when freeing the object as a whole. This saves CoSchmoo from having to QueryInterface for IPersistStorage in the middle of a load or save operation. In the code below, m_pIPersistStorage is of type LPPERSISTSTORAGE:

```
BOOL CSchmooDoc::FInit(...)
  {
  ...

  hr=CoCreateInstance(CLSID_Polyline5, NULL, CLSCTX_INPROC_SERVER
    , IID_IPolyline5, (LPVOID FAR *)&m_pPL);

  if (FAILED(hr))
    return FALSE;

  hr=m_pPL->QueryInterface(IID_IPersistStorage, &m_pIPersistStorage);

  if (FAILED(hr))
    return FALSE;

  ...
  }

CSchmooDoc::~CSchmooDoc(void)
  {
  ...
```

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

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

    return;
    }
```

When loading a file or creating a new one, CoSchmoo follows its responsibilities and provides an IStorage to the Polyline's IPersistStorage::Load or ::InitNew. For a new file, CoSchmoo creates a new temp file using StgCreateDocfile, passes it to the Polyline, then releases it. If the Polyline did not bother to save this storage we don't care–our release accounts for our reference count. When CoSchmoo loads an existing file, it opens it using StgOpenStorage and passes that storage object to the Polyline. Again, we release the storage here ourselves trusting that Polyline will AddRef the storage object before holding on to it:

```
    //For new files
    hr=StgCreateDocfile(NULL, STGM_DIRECT | STGM_READWRITE | STGM_CREATE
       | STGM_DELETEONRELEASE | STGM_SHARE_EXCLUSIVE, 0, &pIStorage);

    if (FAILED(hr))
       return DOCERR_COULDNOTOPEN;

    m_pIPersistStorage->InitNew(pIStorage);
    pIStorage->Release();

    ...

    //For existing files.
    hr=StgOpenStorage(pszFile, NULL, STGM_DIRECT | STGM_READ
       | STGM_SHARE_EXCLUSIVE, NULL, 0, &pIStorage);

    if (FAILED(hr))
       return DOCERR_COULDNOTOPEN;

    hr=m_pIPersistStorage->Load(pIStorage);
    pIStorage->Release();
```

For File Save CoSchmoo always creates a new root storage and passes it to the Polyline being sure to call ::SaveCompleted as protocol demands. The FALSE flag to ::Save indicates that the storage passes in ::Save is not the same as passed to ::Load or ::InitNew:

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

    if (FAILED(hr))
       return DOCERR_COULDNOTOPEN;

    pIStorage->SetStateBits(STGSTATE_DOC, STGSTATE_DOC);

    m_pIPersistStorage->Save(pIStorage, FALSE);
    m_pIPersistStorage->SaveCompleted(pIStorage);

    pIStorage->Release();
```

OLE 2.0 Container Applications and IPersistStorage

OLE 2.0 containers never use IPersistStorage directly; instead, they use functions like OleCreate*, OleLoad, and OleSave which internally call various IPersistStorage members. All of three of these functions (along with the other OleCreate* variants) require you to pass an IStorage pointer for the object. The OleCreate calls, after creating the object using CoCreateInstance, QueryInterface for IPersistStorage on the new object, then pass your IStorage to the object through IPersistStorage::InitNew. OleLoad will first read the CLSID from private streams that OLE places in

the storage (in OleSave), calls CoCreateInstance to get the object, calls QueryInterface for IPersistStorage, then passes your IStorage to IPersistStorage::Load. Since you cannot call OleLoad and OleCreate* for the same object, you cannot call ::Load and ::InitNew on the same IPersistStorage. Finally, to OleSave you pass some interface pointer to the object to save and the IStorage. OLE generates calls to QueryInterface for IPersistStorage, then IPersistStorage::Save passing your IStorage (and you also tell OleSave if it's the same as passed to ::Load or ::InitNew), then IPersistStorage::SaveCompleted. All these Ole* calls will also call IPersistStorage::Release when they're complete.

The only occasion where a container may need to use IPersistStorage itself is when it required use of ::HandsOffStorage: the OLE 2.0 APIs don't make a provision for such an action. But as mentioned in an earlier chapter, most APIs are wrappers for *common* sequences of operations. ::HandsOffStorage is not considered common enough to warrant wrapping in an API.

Of Component Objects and IPersistStorage: Polyline

Our good friend Polyline is on the road to becoming a full compound document object in a DLL, and part of the implementation of such an object is to support IPersistStorage. A number of changes had to occur to Polyline. The most major change significant to our discussion is the addition of IPERSTOR.CPP, shown in Listing 5-3. The other changes, mostly minor, that occur in other files handle the fact that Polyline now has two interfaces, IPolyline and IPersistStorage, where the implementation of the latter is contained in the CImpIPersistStorage class implemented in Listing 5-3. Note that the IPOLY5.H file in the INC directory is a modification if IPolyline from which I have removed the file-related members ::ReadFromFile and ::WriteToFile as their semantics are replaced with IPersistStorage.[1]

IPERSTOR.CPP

```
/*
 * IPERSTOR.CPP
 *
 * Implementation of the IPersistStorage interface that we expose on the
 * Polyline object.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */


#include "polyline.h"


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

Listing 5-3: The IPersistStorage interface implementation for the Polyline object.

1IPersistFile was not used because Polyline will eventually become a compound document object where IPersistStorage is required. We also want to demonstrate compound files in this chapter which IPersistStorage uses, but not IPersistFile.

```
CImpIPersistStorage::~CImpIPersistStorage(void)
   {
   return;
   }



STDMETHODIMP CImpIPersistStorage::QueryInterface(REFIID riid, LPVOID FAR *ppv)
   {
   return m_punkOuter->QueryInterface(riid, ppv);
   }

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

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



STDMETHODIMP CImpIPersistStorage::GetClassID(LPCLSID pClsID)
   {
   LPCPolyline     pObj=(LPCPolyline)m_pObj;

   *pClsID=pObj->m_clsID;
   return NOERROR;
   }



STDMETHODIMP CImpIPersistStorage::IsDirty(void)
   {
   LPCPolyline     pObj=(LPCPolyline)m_pObj;

   return ResultFromScode(pObj->m_fDirty ? S_OK : S_FALSE);
   }



STDMETHODIMP CImpIPersistStorage::InitNew(LPSTORAGE pIStorage)
```

```
    {
    //Nothing to do.  We don't need a storage outside ::Load and ::Save.
    return NOERROR;
    }



STDMETHODIMP CImpIPersistStorage::Load(LPSTORAGE pIStorage)
    {
    LPCPolyline     pObj=(LPCPolyline)m_pObj;
    POLYLINEDATA    pl;
    ULONG           cb;
    LPSTREAM        pIStream;
    HRESULT         hr;

    if (NULL==pIStorage)
        return ResultFromScode(STG_E_INVALIDPOINTER);

    //We don't check ClassStg to remain compatible with other chatpers.

    //Open the CONTENTS stream
    hr=pIStorage->OpenStream("CONTENTS", 0, STGM_DIRECT | STGM_READ
        | STGM_SHARE_EXCLUSIVE, 0, &pIStream);

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

    //Read all the data into the POLYLINEDATA structure.
    hr=pIStream->Read((LPVOID)&pl, CBPOLYLINEDATA, &cb);
    pIStream->Release();

    if (CBPOLYLINEDATA!=cb)
        return ResultFromScode(STG_E_READFAULT);

    pObj->m_pIPolyline->DataSet(&pl, TRUE, TRUE);
    return NOERROR;
    }



STDMETHODIMP CImpIPersistStorage::Save(LPSTORAGE pIStorage, BOOL
fSameAsLoad)
    {
    LPCPolyline     pObj=(LPCPolyline)m_pObj;
    POLYLINEDATA    pl;
    ULONG           cb;
```

```
   LPSTREAM       pIStream;
   HRESULT        hr;

   if (NULL==pIStorage)
      return ResultFromScode(STG_E_INVALIDPOINTER);

   /*
    * fSameAsLoad is not important to us since we always rewrite
    * an entire stream as well as the identification tags for this
    * object.  Note that we don't bother to check the ClassStg
    * above in ::Load to remain compatible with other revisions
    * of Polyline in other chapters.
    */

   WriteClassStg(pIStorage, pObj->m_clsID);
   WriteFmtUserTypeStg(pIStorage, pObj->m_cf, (*pObj->m_pST)[IDS_USERTYPE]);

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

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

   pObj->m_pIPolyline->DataGet(&pl);
   hr=pIStream->Write((LPVOID)&pl, CBPOLYLINEDATA, &cb);
   pIStream->Release();

   return (SUCCEEDED(hr) && CBPOLYLINEDATA==cb) ?
      NOERROR : ResultFromScode(STG_E_WRITEFAULT);
   }


STDMETHODIMP CImpIPersistStorage::SaveCompleted(LPSTORAGE pIStorage)
   {
   /*
    * We have nothing to do here since we do everything in ::Load and
    * ::Save.  For most objects than handle saves this way, they need
    * no code here.  Other objects must release their current storage
    * here and begin using the new one in pIStorage.
    */

   return NOERROR;
   }


STDMETHODIMP CImpIPersistStorage::HandsOffStorage(void)
```

```
{
//Nothing for us to do
return NOERROR;
}
```

You can see that most of the implementation of this interface (::GetClassID, ::IsDirty, ::InitNew, ::SaveCompleted, and ::HandsOffStorage) is trivial since the Polyline requires no incremental access to its eventual storage. ::GetClassID and ::IsDirty only require one-line implementations to return an already known value or status. That leaves only ::Load and ::Save as the interesting functions, but even they are brainless.

Polyline's IPersistStorage::Load simply opens the "CONTENTS" stream, reads the data, releases the stream, and makes the data current. We continue to use "CONTENTS" as the stream name here to remain compatible with the version 2.0 files from the Schmoo (not CoSchmoo) application although we no longer worry about reading 1.0 formats or about reading the newer format we wrote in Chapter 4. If you want to convert other formats use the Schmoo program in this chapter instead. Other than these simplifications, ::Load is implemented like any other compound file code.

The implementation of IPersistStorage::Save does pretty much the standard things as well, simplified by the fact that Polyline doesn't hold on to any storage. Therefore ::Save just creates a new stream in the provides storage and writes to it. During this function as well, we also write identification streams into the storage as done in the other samples:

```
WriteClassStg(pIStorage, pObj->m_clsID);
WriteFmtUserTypeStg(pIStorage, pObj->m_cf, (*pObj->m_pST)[IDS_USERTYPE]);
```

The resulting storages created with Component Schmoo/Polyline here are identical to those generated by the version of Schmoo shown in this chapter. The only difference is that we write a differently CLSID, format, and user type into the storage than Schmoo. You can use the DFVIEW.EXE tool in the OLE 2.0 toolkit to peek into these files and verify my claims.

OLE 2.0 Embedded Objects and IPersistStorage

All OLE 2.0 embedded objects must implement IPersistStorage as one of the three fundamental compound document object interfaces. Polyline will eventually become such an object so it makes sense at this time to implement this portion of compound document requirements now. An embedded object should not expect that the IStorage it receives through IPersistStorage is actually on disk, nor can it assume anything about how the storage was opened (although it can find out using IStorage::Stat). Any embedded object must follow the same protocol as described here irrespective of the context in which it's being used.

Compound File Defragmentation

Since compound files inherently provide incremental saves, the physical size of a compound file on disk will typically be greater than necessary. This is because the size of the file is determined by the amount of space between the first and last sectors used by that file. This is like calculating free space on your hard disk by the location of the first and last files on it instead of by the amount of actual unused sectors: you could have two 1K files on a 1GB disk but since they are located at opposite ends of the drive the disk is considered full.

While this does not actually happen on hard disks, it can happen within the confines of a compound file: there may be plenty of unused space inside the file itself, but the size of that file as reported by the operating system is defined by the first and last used sectors regardless of the internal allocation. While free space is recycled when you write data to the compound file, there is always this possibility of internal fragmentation and larger-than-necessary files as shown in Figure 5-6.

Figure 5-6: A fragmented compound file that takes up more room than necessary on the file system.

A number of tools are available to defragment your hard drive. The Smasher utility in Listing 5-4 is such a tool for a compound file. Smasher is implemented as a File Manager Extension DLL that is compatible both with the File Managers of both Windows 3.1 and Windows for Workgroups 3.1. In the latter system

Smasher also contributes a toolbar button which is an additional feature of the Windows for Workgroups File Manager. Note that Smasher is written in C++ although most of it looks like straight C: the interface member calls use pInterface->MemberFunction(...) instead of pInterface->lpVtbl->MemberFunction(pInterface, ...). Other than that there are no C++ specifics in this code.

SMASHER.CPP

```
/*
 * SMASHER.CPP
 *
 * Functions to demonstrate a File Manager extension DLL that implements
 * a toolbar button to defragment selected compound files.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved.
 */


#include <windows.h>
#include <ole2.h>
#include "wfext.h"              //Windows for Workgroups version.
#include "smasher.h"


HINSTANCE   g_hInst;
BOOL        fInitialized;


//Toolbar to place on Windows for Workgroups File Manager.
EXT_BUTTON btns[1]={{IDM_SMASH, IDS_SMASHHELP+1, 0}};


HANDLE FAR PASCAL LibMain(HINSTANCE hInstance, WORD wDataSeg
    , WORD cbHeapSize, LPSTR lpCmdLine)
    {
    //Remember our instance.
    g_hInst=hInstance;

    if (0!=cbHeapSize)
        UnlockData(0);

    return hInstance;
    }



void FAR PASCAL WEP(int bSystemExit)
    {
```

```
   return;
   }



/*
 * FMExtensionProc
 *
 * Purpose:
 *  File Manager Extension callback function, receives messages from
 *  file manager when extension toolbar buttons and commands are
 *  invoked.
 *
 * Parameters:
 *  hWnd          HWND of File Manager.
 *  iMsg          UINT message identifier
 *  lParam        LONG extra information.
 */
```

Listing 5-4: The Smasher extension for File Manager that defragments a Compound File.

```
HMENU FAR PASCAL FMExtensionProc(HWND hWnd, UINT iMsg, LONG lParam)
   {
   HMENU             hMenu=NULL;
   HRESULT           hr;
   LPMALLOC          pIMalloc;
   LPFMS_LOAD        pLoad;
   LPFMS_TOOLBARLOAD   pTool;
   LPFMS_HELPSTRING    pHelp;

   switch (iMsg)
      {
      case FMEVENT_LOAD:
         pLoad=(LPFMS_LOAD)lParam;
         pLoad->dwSize=sizeof(FMS_LOAD);

         /*
          * Check if our host did CoInitialize by trying CoGetMalloc.
          * If it doesn't work, then we'll CoInitialize ourselves.
          */
         hr=CoGetMalloc(MEMCTX_TASK, &pIMalloc);

         if (SUCCEEDED(hr))
            pIMalloc->Release();
         else
            {
```

```
        hr=CoInitialize(NULL);

        if (FAILED(hr))
            return NULL;

        fInitialized=TRUE;
        }

    //Assign the popup menu name for extension
    LoadString(g_hInst, IDS_SMASH, pLoad->szMenuName
        , sizeof(pLoad->szMenuName));

    //Load the popup menu
    pLoad->hMenu=LoadMenu(g_hInst, MAKEINTRESOURCE(IDR_MENU));

    return pLoad->hMenu;


case FMEVENT_UNLOAD:
    if (fInitialized)
        CoUninitialize();
    break;


case FMEVENT_TOOLBARLOAD:
    /*
     * File Manager has loaded our toolbar extension, so fill
     * the TOOLBARLOAD structure with information about our
     * buttons.  This is only for Windows for Workgroups.
     */

    pTool=(LPFMS_TOOLBARLOAD)lParam;
    pTool->lpButtons= (LPEXT_BUTTON)&btns;
    pTool->cButtons = 1;
    pTool->cBitmaps = 1;
    pTool->idBitmap = IDR_BITMAP;
    break;


case FMEVENT_HELPSTRING:
    //File Manager is requesting a status-line help string.
    pHelp=(LPFMS_HELPSTRING)lParam;

    LoadString(g_hInst, IDS_SMASHHELP+pHelp->idCommand
            , pHelp->szHelp, sizeof(pHelp->szHelp));
```

```
        break;



    case IDM_SMASH:
        SmashSelectedFiles(hWnd);
        break;
    }

  return hMenu;
  }



BOOL SmashSelectedFiles(HWND hWnd)
  {
  FMS_GETFILESEL  fms;
  UINT          cFiles;
  UINT          i;
  LPSTR          pszErr;
  HRESULT        hr;
  STATSTG        st;
  OFSTRUCT       of;
  LPMALLOC       pIMalloc;
  LPSTORAGE      pIStorageOld;
  LPSTORAGE      pIStorageNew;

  /*
   * Retrieve information from File Manager about the selected
   * files and allocate memory for the paths and filenames.
   */

  //Get the number of selected items.
  cFiles=(UINT)SendMessage(hWnd, FM_GETSELCOUNT, 0, 0L);

  //Nothing to do, so quit.
  if (0==cFiles)
    return TRUE;

  //Get error string memory
  hr=CoGetMalloc(MEMCTX_TASK, &pIMalloc);

  if (FAILED(hr))
    return FALSE;

  pszErr=(LPSTR)pIMalloc->Alloc(1024);
```

```
   /*
    * Enumerate the selected files and directories using the FM_GETFILESEL
    * message to the File Manager window.  For each file, check if its
    * a Compound File (StgIsStorageFile) and if not, skip it.
    *
    * If it is a compound file, then create a temp file and ::CopyTo
    * from old to new.  If this works, then we reopen the old file
    * in overwrite mode and ::CopyTo back into it.
    */

   for (i = 0; i < cFiles; i++)
      {
      SendMessage(hWnd, FM_GETFILESEL, i, (LONG)(LPSTR)&fms);

      //Skip non-storages.
      hr=StgIsStorageFile(fms.szName);

      if (FAILED(hr))
         {
         wsprintf(pszErr, SZERRNOTACOMPOUNDFILE, (LPSTR)fms.szName);
         MessageBox(hWnd, pszErr, SZSMASHER, MB_OK | MB_ICONHAND);
         continue;
         }

      /*
       * Create a temporary Compound File.  We don't use DELETEONRELEASE
       * in case we have to save it when coying over the old file fails.
       */
      hr=StgCreateDocfile(NULL, STGM_CREATE | STGM_DIRECT |
STGM_READWRITE
         | STGM_SHARE_EXCLUSIVE, 0, &pIStorageNew);

      if (FAILED(hr))
         {
         MessageBox(hWnd, SZERRTEMPFILE, SZSMASHER, MB_OK |
MB_ICONHAND);
         continue;
         }

      //Open the existing file as read-only
      hr=StgOpenStorage(fms.szName, NULL, STGM_DIRECT | STGM_READ
         | STGM_SHARE_DENY_WRITE, NULL, 0, &pIStorageOld);

      if (FAILED(hr))
         {
```

```
            pIStorageNew->Release();
            wsprintf(pszErr, SZERROPENFAILED, (LPSTR)fms.szName);
            MessageBox(hWnd, pszErr, SZSMASHER, MB_OK | MB_ICONHAND);
            continue;
            }

        /*
         * Compress with ::CopyTo.  Since the temp is opened in
         * direct mode, changes are immediate.
         */
        hr=pIStorageOld->CopyTo(NULL, NULL, NULL, pIStorageNew);
        pIStorageOld->Release();

        if (FAILED(hr))
            {
            pIStorageNew->Release();
            MessageBox(hWnd, SZERRTEMPFILECOPY, SZSMASHER, MB_OK |
MB_ICONHAND);
            continue;
            }

        //Temp file contains the defragmented copy now, try copying back.
        hr=StgOpenStorage(fms.szName, NULL, STGM_DIRECT | STGM_CREATE
            | STGM_WRITE | STGM_SHARE_EXCLUSIVE, NULL, 0, &pIStorageOld);

        if (FAILED(hr))
            {
            pIStorageNew->Stat(&st, 0);
            pIStorageNew->Release();

            wsprintf(pszErr, SZERRTEMPHASFILE, (LPSTR)st.pwcsName);
            pIMalloc->Free((LPVOID)st.pwcsName);

            MessageBox(hWnd, pszErr, SZSMASHER, MB_OK | MB_ICONHAND);
            continue;
            }

        //Copy over the old one.
        pIStorageNew->CopyTo(NULL, NULL, NULL, pIStorageOld);
        pIStorageOld->Release();

        //Delete the temporary file.
        pIStorageNew->Stat(&st, 0);
        pIStorageNew->Release();

        OpenFile(st.pwcsName, &of, OF_DELETE);
```

```
    pIMalloc->Free((LPVOID)st.pwcsName);
    }

  pIMalloc->Free((LPVOID)pszErr);
  pIMalloc->Release();

  return TRUE;
  }
```

SMASHER.H

```
/*
 * SMASHER.H
 *
 * Definitions and function prototypes for the file manager extesion
 * SMASHER.DLL.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved.
 */


//Resource identifiers
#define IDR_BITMAP          1
#define IDR_MENU            1

//Menu constants
#define IDS_SMASH           1
#define IDS_SMASHHELP       100
#define IDM_SMASH           1


//Cheap strings
#define SZSMASHER               "Compound File Smasher"
#define SZERRNOTACOMPOUNDFILE   "Smasher cannot defragment %s.\n\rNot a
compound file."
#define SZERRTEMPFILE           "Could not create an intermediate file.\n\rCheck disk
space and your TEMP environment variable."
#define SZERROPENFAILED         "Could not access %s for defragmentation.\n\rFile
could be locked."
#define SZERRTEMPFILECOPY       "Could not write to intermediate file.\n\rCould be out
of disk space or memory."
#define SZERRTEMPHASFILE        "Failure to overwrite file.  Defragemented version can
be found in\r\n%s."
```

```
//SMASHER.CPP
BOOL SmashSelectedFiles(HWND);

extern "C"
   {
   HMENU FAR PASCAL FMExtensionProc(HWND hWnd, UINT iMsg, LONG lParam);
   }
```

SMASHER.RC

```
/*
 * SMASHER.RC
 *
 * Menu and string resources for SMASHER.DLL.  The menu is passed
 * as a popup menu to File Manager, the bitmap is used to define
 * the toolbar button image, and the string table is used to provide
 * File Manager with menu commands and strings.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved.
 */

#include <windows.h>
#include "smasher.h"

IDR_BITMAP BITMAP smasher.bmp


IDR_MENU MENU
  BEGIN
   MENUITEM "&Smash Compound File...",     IDM_SMASH
  END


STRINGTABLE
  BEGIN
   IDS_SMASH              "&Smash Compound Files"
   IDS_SMASHHELP,         "Commands for Compound Files"
   IDS_SMASHHELP+1,       "Defragments the selected Compound Files"
  END
```

SMASHER.BMP

This image is used for a toolbar button in the Windows for Workgroups 3.1 File Manager.

Smasher's implementation is simple. First it calls StgIsStorageFile to check that the file is actually a compound file. Next it creates a temporary file for the defragmented copy after which it opens the file to defragment. It then calls ::CopyTo from the original file to the temporary file, which performs the defragmentation as shown in Figure 5-7.

Figure 5-7: IStorage::CopyTo inherently defragments all storages and streams in the process of copying one compound file to another. Unused space is returned to the file system.

Smasher then reopens the original file with write permissions and uses ::CopyTo to write the defragmented data from the temporary file into the new file under the original name. When this is done Smasher closes the files and deletes the temporary one.

There are two other important points about Smasher. First of all, in order to delete the temporary file it has to have a filename to pass to OpenFile(..., OF_DELETE). This can be obtained by calling ::Stat on the temporary file which fills a STATSTG structure pointing to the filename in pwcsName. As mentioned earlier, we are responsible for this string which we must pass to the task allocator's ::Free when we are done. We also use this task allocator to get 1K of scratch memory in which to generate error messages, so the function SmashSelectedFiles calls CoGetMalloc early on to obtain the allocator matching that call with IMalloc::Release at the end.

Now remember again that CoGetMalloc returns the task allocator from CoInitialize. Well, File Manager is not an OLE 2.0 application and therefore has not called CoInitialize; that does not stop us, however, from calling it ourselves in this DLL which happens in FMEVENT_LOAD case of FMExtensionProc. In this case we first *try* CoGetMalloc to test whether or not File Manager has already called CoInitialize, as it will in the future. If it has not, then we can go ahead and call CoInitailize(NULL), making sure to match that call in FMEVENT_UNLOAD with CoUninitilize.

If you would like more information on File Manager Extensions, please refer to your Windows 3.1 Software Development Kit.

Summary

Structured storage is a model designed to sit on top of an existing file system that provides sharable storage elements which can greatly improve performance of many large data transfers as well as simplifying the implementation of features such as incremental save. The model describes a 'file system within a file' with storage objects that act like directories and stream objects that act like files. Applications benefit from structuring data into a directory and file model (thereby reducing many uses of seek offsets) but still maintain the data within a single entity on the actual file system. The actual storage device is hidden from storage and stream objects by a LockBytes object.

The OLE 2.0 provided implementation of this model, Compound Files, not only can apply anywhere you would normally use traditional file I/O but also open new possibilities of managing your application's data structures. Compound files also offer support for transactioning to further reinforce the strength of this storage model over traditional file usage. Compound files are an important part of OLE 2.0 as they are used in data transfer as well as compound document implementations. Three standard interfaces define functions for objects that wish to support serialization to a storage object, a stream object, or a file.

One drawback to using compound files is a potential for larger disk files that can also become internally fragmented, but compound files provides its own method for defragmentation. It is therefore trivial to write a defragmentation tool.