Asynchronous Storage

OLE Team, May 1, 1996

Asynchronous Storage turns the existing Compound Files technology into a flexible solution for managing complex data over slow-link, high-latency networks. A sophisticated layout tool provides fine-grained control over the order of content streams (and even within contents streams) at publish-time. An asynchronous download and access mechanism allows data to be processed as soon as it becomes available at run-time. A single request to the server triggers the download of arbitrarily nested objects contained within a page, eliminating costly requests for embedded images and objects. The exact order in which elements of a page become available is predefined at publish-time and not dependent on random factors of network topology and server availability.

1. Introduction
2. Scenarios
3. Asynchronous Compound Files (ACF)
3.1 Architecture of Asynchronous Compound Files
3.2 Asynchronous Byte Array Wrapper Object
3.2.1 Interfaces
3.2.1.1 ILockBytes
3.2.1.2 IFillLockBytes
3.3 Progress Notification: IProgressNotify
3.4 Asyncronous Docfile APIs
3.4.1 Open Asynchronous Storage on Wrapper
3.4.2 Create wrapper object on ILockBytes
3.4.3 Create wrapper object on File
3.4.4 Examples
4. Optimization for Compound Files
4.1 ILayoutStorage
4.1.1 Scripted layout: ILayoutStorage::LayoutScript
4.1.1.1 StorageLayout
4.1.1.2 Example
4.1.2 Monitoring: ILayoutStorage::BeginMonitor/EndMonitor
4.1.2.1 Example
4.1.3 Optimizing the file: ILayoutStorage::ReLayoutDocfile
4.1.4 Combining Scripting and Monitoring
5. Index

NOTE: THIS DOCUMENT IS AN EARLY RELEASE OF THE FINAL SPECIFICATION. IT IS MEANT TO SPECIFY AND ACCOMPANY SOFTWARE THAT IS STILL IN DEVELOPMENT. SOME OF THE INFORMATION IN THIS DOCUMENTATION MAY BE INACCURATE OR MAY NOT BE AN ACCURATE REPRESENTATION OF THE FUNCTIONALITY OF THE FINAL SPECIFICATION OR SOFTWARE. MICROSOFT ASSUMES NO RESPONSIBILITY FOR ANY DAMAGES THAT MIGHT OCCUR EITHER DIRECTLY OR INDIRECTLY FROM THESE INACCURACIES. MICROSOFT MAY HAVE TRADEMARKS, COPYRIGHTS, PATENTS OR PENDING PATENT APPLICATIONS, OR OTHER INTELLECTUAL PROPERTY RIGHTS

 © Microsoft Corporation 1996. All Rights Reserved.

COVERING SUBJECT MATTER IN THIS DOCUMENT. THE FURNISHING OF THIS DOCUMENT DOES NOT GIVE YOU A LICENSE TO THESE TRADEMARKS, COPYRIGHTS, PATENTS, OR OTHER INTELLECTUAL PROPERTY RIGHTS.

 © Microsoft Corporation 1996. All Rights Reserved.

1Introduction

The Internet requires new approaches to application design, primarily due to its low-speed, extremely high latency network access.

Asynchronous Monikers address these issues by providing an asynchronous programming model for object instantiation. The instantiation of persisted objects involves a call to IMoniker::BindToStorage which returns a storage object. The most common storage objects expose IStream and IStorage interfaces.

This document focuses on the asynchronous programming model involving storage objects.

Of specific interest are objects exposing IStorage that represent Compound Files. These are used in many of today's applications and currently suffer the following limitations:

- the complete file has to be downloaded before it can be safely accessed through the existing APIs.
- applications currently have no control over the layout of the Compound File they create. Information required first for progressive rendering could be physically located at the end of the file.

This document describes solutions to both problems and is aimed at enabling existing applications to efficiently render their content when accessed over the existing protocols on the Internet.

2Scenarios

The common use of Asynchronous Compound Files is in the context of a URL Moniker.

The URL Moniker returns an asynchronous IStream or IStorage implementation from IMoniker::BindToStorage. Clients indicate the storage mode of the asynchronous storage: The storage can block when the data being read from it is not available yet or it can return a new error code E_PENDING (ABINDF_ASYNCSTORAGE).

Non-asynchronous aware clients will obtain a blocking asynchronous storage from a bind to an asynchronous moniker. This enables them to do progressive rendering and will even allow some existing applications to render before having access to the complete file.

Asynchronous aware clients may want to obtain a non-blocking asynchronous storage. They will perform read operations from within the IBindStatusCallback::OnDataAvailable notifications.

Please refer to the Asynchronous Moniker Specification for detailed scenarios.

3Asynchronous Compound Files (ACF)

Compound Files are implemented on top of an abstraction of a file: a Byte Array Object. This object exposes it's functionality through the interface **ILockBytes**.

There are currently two implementation of Byte Array Objects:

- File: reads and writes data to a file (not currently exposed but used internally).
- Memory: reads and writes data to memory

An application can provide a private implementation of a Byte Array Object and reuse COM's implementation of Compound Files by calling StgCreateDocfileOnILockBytes or StgOpenStorageOnILockBytes.

If this Byte Array Object exposes non-blocking asynchronous behavior (by returning E_PENDING), the existing compound file implementation will propagate this error to the original caller.

Thus by providing an asynchronous Byte Array implementation, the existing Compound File implementation can be leveraged to provide Asynchronous Compound Files.

Existing applications are not prepared to handle these new error codes and will assume that there is an unrecoverable error. ACF thus have to provide two basic modes of operation:

- synchronous: block until data is available.
- asynchronous: return with E_PENDING

 © Microsoft Corporation 1996. All Rights Reserved.

The Compound File implementation provides a connection point that is called in the case of a pending operation. A sink (usually an Asynchronous Moniker like the URL Moniker) registered on this connection point can control the behavior in three ways:

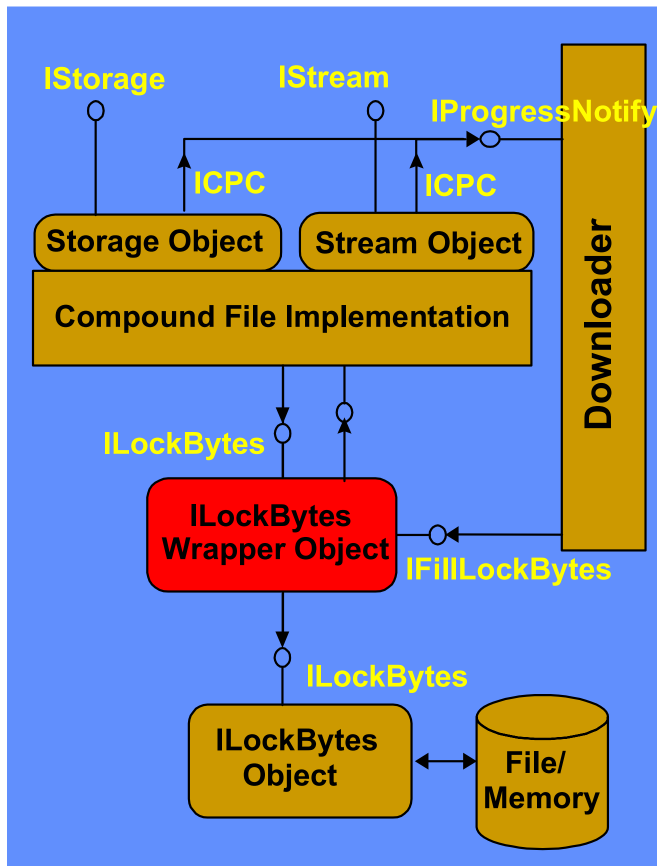
1. Return immediately from the callback and have the Compound File Implementation block the thread until the data arrives (Synchronous Mode)
2. Return immediately from the callback and propagate the error code back to the client (Asynchronous Mode)
3. Not return from the callback until data has arrived and indicate to the Compound File Implementation to retry the operation (Synchronous Mode with control over type of blocking).

The third option can be used to block using a modal message loop instead of completely blocking the thread. This allows the user interface to be updated and incoming OLE calls to be processed.

3.1 Architecture of Asynchronous Compound Files

A wrapper object keeps track of the data available in any Byte Array Object exposing an ILockBytes interface.

Asynchronous Compound Files also add a connection point to the existing storage and stream objects to allow for notification of availability/non-availability of data and control over synchronous or asynchronous mode.



3.2 Asynchronous Byte Array Wrapper Object

This specification defines an object, that allows for synchronous access to asynchronously arriving data. It can be layered on top of any Byte Array Object exposing ILockBytes.

 © Microsoft Corporation 1996. All Rights Reserved.

3.2.1 Interfaces

The object exposes the following interfaces:

- **ILockBytes** for use by the Compound File implementation
- **IFillLockBytes** for use by the downloading process.

The object uses ILockBytes exposed by a Byte Array Object.

ILockBytes

This interface provides access to the Byte Array functionality exposed by the wrapper object. The Compound File implementation will use this interface.

The definition of ILockBytes from the Win32 SDK (OBJIDL.IDL), repeated here for completeness:

```
[object, uuid(0000000a-0000-0000-C000-000000000046), pointer_default(unique)]
interface ILockBytes : IUnknown {
  [local] HRESULT __stdcall ReadAt([in] ULARGE_INTEGER ulOffset, [in] void *pv,
                                  [in] ULONG cb, [out] ULONG *pcbRead);
  [local] HRESULT __stdcall WriteAt([in] ULARGE_INTEGER ulOffset, [in] void const *pv,
                                   [in] ULONG cb, [out] ULONG *pcbWritten);
  HRESULT Flush();
  HRESULT SetSize([in] ULARGE_INTEGER cb);
  HRESULT LockRegion([in] ULARGE_INTEGER libOffset, [in] ULARGE_INTEGER cb,
                [in] DWORD dwLockType);
  HRESULT UnlockRegion([in] ULARGE_INTEGER libOffset, [in] ULARGE_INTEGER cb,
                 [in] DWORD dwLockType);

  HRESULT Stat([out] STATSTG *pstatstg, [in] DWORD grfStatFlag);
}
```

This interface behaves as the underlying ILockBytes-implementation, except for additional error codes:

ILockBytes::ReadAt and ILockBytes::WriteAt return an additional error code:

E_PENDING

Indicates that at least part of the requested bytes were not filled yet. Any available bytes are returned in pv and pcbRead contains the actual number of bytes read or 0 if no bytes were available.

ILockBytes::WriteAt may return an additional error code:

E_PENDING

Indicates that at least part of the bytes to be written were not filled yet. If part of the bytes were filled pcbWritten contains the actual number of bytes written.

IFillLockBytes

This interface is exposed by the Byte Array Wrapper object. The URL Moniker provided downloading code will call this interface, to asynchronously fill the Byte Array as data arrives.

```
[
  local,
  object,
  uuid(99caf010-415e-11cf-8814-00aa00b569f5),
  pointer_default(unique)
]

interface IFillLockBytes: IUnknown
{
  import "unknwn.idl";

  HRESULT FillAppend
  (
    [in] void const *pv,
    [in] ULONG cb,
    [out] ULONG *pcbWritten
```

 © Microsoft Corporation 1996. All Rights Reserved.

```
    );

    HRESULT FillAt
    (
        [in] ULARGE_INTEGER ulOffset,
        [in] void const *pv,
        [in] ULONG cb,
        [out] ULONG *pcbWritten
    );

    HRESULT SetFillSize
    (
        [in] ULARGE_INTEGER ulSize
    );

    HRESULT Terminate
    (
        [in] BOOL bCanceled
    );
}
```

IFillLockBytes::FillAppend

Writes a new block of bytes to the end of the underlying Byte Array. Subsequent calls to ILock-Bytes::ReadAt referring to the written data will be forwarded to the underlying Byte Array. This function is a helper function for the convenience of the downloading code: It (conceptually) encapsulates a call to ILockByte::Stat to obtain the current size and a call to IFillLockBytes::FillAt.

IFillLockBytes::FillAt

Writes a new block of bytes to the underlying Byte Array Object. Subsequent calls to ILock-Bytes::ReadAt referring to the written data will be forwarded to the underlying Byte Array.

This method is not currently implemented and will return E_NOTIMPL.

IFillLockBytes::SetFillSize

Sets the expected size of the Byte Array being wrapped. After SetFillSize is called, the wrapper fails any call to ILockBytes::ReadAt, that accesses data beyond the specified size with the same error as the underlying Byte Array Object. If SetFillSize is not called, the wrapper will return E_PENDING when accesses beyond the currently written data occur. SetFillSize can be called multiple times.

IFillLockBytes::Terminate

Indicates the successful or unsuccessful termination of the download. After a successful termination all calls to ILockBytes methods will be forwarded to the underlying Byte Array Object.

3.3 Progress Notification: IProgressNotify

The objects that expose IStream and IStorage provide a connection point for IProgressNotify. It allows complete control over the behavior in the case of unavailable data.

```
[
    local,
    object,
    uuid(a9d758a0-4617-11cf-95fc-00aa00680db4),
    pointer_default(unique)
]
interface IProgressNotify: IUnknown
{
    HRESULT OnProgress (
    [in]    DWORD     dwProgressCurrent,
    [in]    DWORD     dwProgressMaximum,
    [in]    BOOL      fAccurate,
    [in]    BOOL      fOwner
    );
}
```

 © Microsoft Corporation 1996. All Rights Reserved.

The Asynchronous Storage implementation calls IProgressNotify::OnProgress with the following information:

- dwProgressCurrent/dwProgressMaximum: the current and expected amount of data required to satisfy the operation in progress
- fAccurate: indicates if dwProgressCurrent/dwProgressMaximum provide accurate information or may still change significantly due to missing control structures indicating the actual position/length of missing data.
- fOwner: This flag indicates if this sink can actually influence the further handling of the pending operation. If this flag is TRUE, the sink may return the following success codes:
  STG_S_RETRYNOW: the Asynchronous Storage implementation will retry the operation immediately.
  STG_S_BLOCK: the Asynchronous Storage implementation blocks the thread until the required data arrives.
  STG_S_MONITORING: this passes control over the further handling of the pending operation to any additional sinks registered on the connection point.

Independent of the fOwner flag, the sink may return an error code (i.e. E_PENDING). The Asynchronous Storage implementation will abort the current operation and propagate the error code to the original caller.

If no sink is registered, the thread will block until the requested data becomes available or the download is canceled by the downloader.

Sinks are optionally inherited by any sub-storage or sub-stream of a given storage. This behavior is controlled through the ASYNC_MODE_COMPATABILITY flag passed to StgOpenAsyncDocfileOnIFillLockBytes.

3.4Asynchronous Docfile APIs

These APIs are declared in objbase.h and implemented in OLE32.DLL.

3.4.1Open Asynchronous Storage on Wrapper

```
WINOLEAPI StgOpenAsyncDocfileOnIFillLockBytes(
        [in] IFillLockBytes *pflb,          // Wrapper object exposing ILockBytes and IFillLockBytes:
                                            //      contains the compound file
        [in] DWORD grfMode,                 // Storage mode as in StgOpenStorageOnILockBytes (see below)
        [in] DWORD asyncFlags,              // Specific flags for asynchronous storage (see below)
        [out] IStorage **ppstgOpen);        // returned pointer to the Asynchronous Storage
```

StgOpenAsyncDocfileOnIFillLockBytes operates very much like StgOpenStorageOnILockBytes. Note that priority mode is not supported, nor are exclusions. It is expected that the most common grfMode will be STGM_DIRECT | STGM_READ | STGM_SHARE_EXCLUSIVE.

The asyncFlags parameter allows the caller to specify whether connection points from a storage would be inherited by its substorages and streams. ASYNC_MODE_COMPATIBILITY indicates that the connection point will be inherited; ASYNC_MODE_DEFAULT disables Connection Point inheritance.

3.4.2Create wrapper object on ILockBytes

```
HRESULT StgGetIFillLockBytesOnILockBytes(
      [in] ILockBytes *pilb,
      [out] IFillLockBytes **ppflb);
```

StgGetFillLockBytesOnILockBytes takes an arbitrary ILockBytes and creates a byte array wrapper object exposing ILockBytes and IFillLockBytes.

 © Microsoft Corporation 1996. All Rights Reserved.

3.4.3Create wrapper object on File

```
HRESULT StgGetIFillLockBytesOnFile(
        [in] OLECHAR const *pwcsName,
        [out] IFillLockBytes **ppflb);
```

StgGetFillLockBytesOnFile constructs an ILockBytes on a file, then wrap it in the same byte array wrapper object, and return the wrapper.

3.4.4Examples

These code fragments illustrate a possible use of Asynchronous Storage. It opens a local compound file and writes it to a temporary Asynchronous Storage.

```
//+---------------------------------------------------------------------------
//
//  Function:   OpenAsynchRoot
//
//  Synopsis:  Opens an asynchronous docfile and starts a thread to begin downloading
//
//  Arguments:  [szLocal] – local copy name
//              [szRemote] - remote copies name
//
//  Returns:
//
//  History:
//
//  Notes:
//
//----------------------------------------------------------------------------

BOOL OpenAsynchRoot(PCHAR szLocal, PCHAR szRemote)
{
   CHAR                 szDir[MAX_PATH+1];
   WCHAR                wszName[MAX_PATH+1];
   HRESULT              hRes;
   IFillLockBytes*      piflb;
   IStorage *           pIStorage;
   ULONG                ulRet = 0;

   GetCurrentDirectory (sizeof(szDir), szDir);
   wsprintfW(wszName, "%hs\\%hs", szDir, szLocal);

   hRes = StgGetIFillLockBytesOnFile(wszName, &piflb);
   //an error reporting macro that also returns out of the func.
   HRESWarnRtn2 ("OpenAsynchRoot", hRes,
            "StgGetIFillLockBytesOnFile failed");

    //some code to create a thread and pass it the name of  the remote file and the
    // IFillLockBytes

   ThreadArg * ptarg = new ThreadArg(szRemote,piflb);
   CThread * pthrd = new CThread(&DownLoadThread,(VOID *)ptarg);
   //start downloading
   pthrd->Go();

   // try to open the async stg until we stop getting pending msgs
   do {
        hRes = StgOpenAsyncDocfileOnIFillLockBytes(
                piflb,    //IFillLockBytes * pflb
                STGM_DIRECT | STGM_READWRITE | STGM_SHARE_EXCLUSIVE, // grfMode
                0,          // asyncFlags
                &pIStorage);
        if (E_PENDING == hRes) {
                Sleep(1000); //wait asecond and try again
                }
   } while (E_PENDING == hRes  );
```

© Microsoft Corporation 1996. All Rights Reserved.

```
  if (hRes != S_OK) {
       if (pthrd != NULL) delete pthrd;
       if (piflb != NULL) {
       ulRet = piflb->Release();
       }
  }

  HRESWarnRtn2 ("OpenAsynchRoot", hRes,
            "StgGetIFillLockBytesOnFile failed");

    return TRUE;
} //OpenAsynchRoot

//+--------------------------------------------------------------------------
//
//  Function:   DownLoadThread
//
//  Synopsis:
//
//  Arguments:  [pvoid] --
//
//  Returns:
//
//  History:
//
//  Notes:
//
//---------------------------------------------------------------------------

DWORD WINAPI DownLoadThread(VOID * pvoid)
{
  ThreadArg *   parg = (ThreadArg *)pvoid;
  CHAR          szDir[MAX_PATH+1];
  CHAR          szName[MAX_PATH+1];
  HANDLE        hFile = INVALID_HANDLE_VALUE;
  ULARGE_INTEGER liSize;
  ULARGE_INTEGER liIndex = {0,0};
  BYTE *        pBuffer = NULL;
  DWORD         cbRead;
  HRESULT       hRes;
  DWORD         dwRes = 0;

  __try {
       pBuffer = new BYTE[parg->_dwChunkSize];
       if (!pBuffer) {
       dwRes = 1;
              __leave;
       }

       GetCurrentDirectory (sizeof(szDir), szDir);
       wsprintf(szName, "%hs\\%hs", szDir, parg->_lpRemote);

       // open the actual file
       hFile = CreateFile(szName,GENERIC_READ,FILE_SHARE_READ,NULL,
       OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);
       if(INVALID_HANDLE_VALUE == hFile) {
              printf("Couldn't open %ls: Last Error: %x",szName,GetLastError());
              dwRes = 1;
              __leave;
              }

       liSize.LowPart = GetFileSize(hFile,&(liSize.HighPart));
       if (0xFFFFFFFF == liSize.LowPart) {
              printf("Couldn't get file size of %ls",szName);
              dwRes = 1;
       __leave;
              }

       //do a simulated download sleeping 15 seconds btwn 512 byte chunks
       for(; liIndex < liSize; liIndex+=512) {
```

 © Microsoft Corporation 1996. All Rights Reserved.

```
            if (!ReadFile(hFile,pBuffer,512 ,&cbRead,NULL)) {
                printf("Couldn't read %ls: Last Error: %x",szName,GetLastError());
                dwRes = 1;
                __leave;
                }
            //fill in the local copy
            hRes =  parg->_piflb->FillAppend((VOID *)pBuffer, 512, NULL);
            HRESWarnRtn2("DownLoadThread",hRes, "iFLockBytes::FillAppend failed");
        // sleep the give delay amount before next chunk
        Sleep(15000);
            } //endfor

        //notify IFillLockBytes that file is all copied down
        hRes = parg->_piflb->Terminate(FALSE);
        HRESWarnRtn2("DownLoadThread",hRes, "iFLockBytes::Terminate failed");
        } __finally {
        if (hFile != INVALID_HANDLE_VALUE) CloseHandle(hFile);
        if (pBuffer != NULL) delete[] pBuffer;
        if (parg != NULL) delete parg;
        }

  return dwRes;
} //DownLoadThread
```

4Optimization for Compound Files

Compound Files provide the functionality of a file within a file: the file contains directory sectors with the names of the elements (Storages and Streams), stream data is allocated in sectors, sector allocation is controlled by FAT sectors. This complex structure enables efficient transaction support and flexible allocation of data as space is needed.

A standard compound file may contain unused sectors, the streams may be fragmented. These seeming disadvantages offer a new flexibility for Internet scenarios. A file can be laid out to fit the applications need:

- A file can contain data in the order an application actually needs it: If an application only requires part of its data to display a first page of information, this data can be put to the beginning of the file even if it logically resides at the end of a stream.
- Data from different streams can be interleaved: Audio and Video streams are transparently interleaved and a sequential read operation retrieves the data simultaneously.

A new Compound File implementation exposes new APIs, that give applications flexible control over the layout of the compound file. Combining layout optimization with Asynchronous Storage allows application to effectively do progressive rendering. Even legacy applications can benefit, unless they load all their data into memory up-front.

This technology provides extremely flexible static optimization of compound files at author/publish time.

4.1ILayoutStorage

This interface is exposed by the root storage of the new compound files implementation. It allows applications to indicate the desired layout of a compound file.

The optimization interface is exposed on a special compound file implementation, which can be obtained by calling the following API, declared in objbase.h and implemented in dflayout.dll:

```
WINOLEAPI StgOpenLayoutDocfile(
    OLECHAR const *pwcsDfName, // name of the compound file to be optimized
    DWORD grfMode,                    //
    DWORD reserved,
    IStorage **ppstgOpen);
```

 © Microsoft Corporation 1996. All Rights Reserved.

StgOpenLayoutDocfile will open the compound file on a ILockBytes implementation capable of monitoring sector information. Note that priority mode is not supported, nor are exclusions. It is expected that the most common grfMode will be STGM_DIRECT | STGM_READ | STGM_SHARE_EXCLUSIVE.

Applications obtain this interface by calling IUnknown::QueryInterface on the object returned from StgOpenLayoutDocfile.

```
interface ILayoutStorage: IUnknown
{
	typedef struct tagStorageLayout
	{	DWORD			LayoutType;
		OLECHAR			*pwcsElementName;
		LARGE_INTEGER	cOffset;
		LARGE_INTEGER	cBytes;
	} StorageLayout;

	HRESULT __stdcall LayoutScript(
	[in]	StorageLayout *pStorageLayout,
	[in]	DWORD			nEntries,
	[in]	DWORD			glfInterleavedFlag);

	HRESULT __stdcall BeginMonitor(void);

	HRESULT __stdcall EndMonitor(void);

	HRESULT __stdcall ReLayoutDocfile(
	[in]	OLECHAR			*pwcsNewDfName);
}
```

Layout information can be provided by two basic methods: Scripting and monitoring. Both methods can be mixed to allow containers to provide scripted information for their own native data and monitor embedded objects they might not know.

4.1.1 Scripted layout: ILayoutStorage::LayoutScript

Scripted Layout lets applications provide explicit layout information to the Internet Compound File implementation. The applications describes the desired access pattern as an array of StorageLayout-elements:

Note: Scripted layout is not supported in NT 4.0, Beta 2. It will be supported in the final release.

StorageLayout

Each element describes one block of data to be accessed:

StorageLayout::LayoutType defines the type of this block:

- STGTY_STREAM: a block of data is to be read form the stream described in StorageLayout::pwcsElementName. The block starts at offset StorageLayout::cOffset and has a length of StorageLayout::cBytes..
- STGTY_STORAGE: the storage described in StorageLayout::pwcsElementName is to be opened. StorageLayout::cOffset and StorageLayout::cBytes should be 0
- STGTY_REPEAT: the following elements are to be repeated until the next element of type STGTY_REPEAT.
  In the opening STGTY_REPEAT element, StorageLayout::cBytes indicates the number of repetitions for the elements between the opening and the closing element. A value of STG_TOEND indicates repetition until the end of all streams referred. StorageLayout::cOffset should be 0.
  In the closing STGTY_REPEAT element, StorageLayout::cBytes and StorageLayout::cOffset should be 0.

Example

Basic structure of scripted layout optimization:

```
pRootStg->QueryInterface(IID_ILayoutStorage, (void**) &pLayout);
```

 © Microsoft Corporation 1996. All Rights Reserved.

```
StorageLayout arrScript[] =
{   //…
};

pLayout->LayoutScript(
        &arrScript,
        sizeof(arrScript)/sizeof(arrScript[0]),
        STG_LAYOUT_INTERLACED); // Interlace control structures

// Write new compound file with desired layout
pLayout->ReLayoutDocfile(L"Optimized File.doc");
```

Sample script arrays:

```
StorageLayout arrScript[] = {
        // Read first 2k of "WordDocument" stream
         {STGTY_STREAM, L"WordDocument", 0, 2048 },

        // Test if "ObjectPool\88112233" storage exists
         {STGTY_STORAGE, L"ObjectPool\\88112233",0,0},

        // Read 2k at offset 10480 of "WordDocument" stream
         {STGTY_STREAM, L"WordDocument", 10480, 2048 }

        // Interlace "Audio", "Video" and "Caption" streams
         {STGTY_REPEAT, NULL, 0, STG_TOEND},
                {STGTY_STREAM, L"Audio", 0, 2048}, //  2k of Audio
                {STGTY_STREAM, L"Video", 0, 65536},// 64k of Video
                {STGTY_STREAM, L"Caption", 0, 128},// 128b text
         {STGTY_REPEAT, NULL, 0, 0}
};

StorageLayout arrWord[] =
{
    { STGTY_STREAM,  L"WordDocument", 0, 2048},
    { STGTY_STREAM,  L"WordDocument", 12800, 2048},
    { STGTY_STREAM,  L"WordDocument", 14848, 346},
    { STGTY_STREAM,  L"WordDocument", 12288, 2048},
    { STGTY_STREAM,  L"WordDocument", 10752, 2048},
    { STGTY_STREAM,  L"WordDocument", 10240, 2048},
    { STGTY_STREAM,  L"WordDocument", 7680, 2048},
    { STGTY_STREAM,  L"WordDocument", 9728, 512},

    { STGTY_STREAM,  L"ObjectPool\\_823896884\\.PIC", 0, 76},
    { STGTY_STORAGE, L"ObjectPool\\_823896884\\.PRINT", 0, 0},
    { STGTY_STREAM,  L"ObjectPool\\_823896884\\.META", 0, 101896},

    { STGTY_STREAM,  L"WordDocument", 2048, 7*512},
    { STGTY_STREAM,  L"WordDocument", 7168, 3*512},

    { STGTY_STREAM,  L"ObjectPool\\_823617166\\.PIC", 0, 76},
    { STGTY_STORAGE, L"ObjectPool\\_823617166\\.PRINT", 0, 0},

    { STGTY_STREAM,  L"ObjectPool\\_823620610\\.PIC", 0, 76},
    { STGTY_STORAGE, L"ObjectPool\\_823620610\\.PRINT", 0, 0},

    { STGTY_STREAM,  L"WordDocument", 5632, 2048},
};
```

4.1.2Monitoring: ILayoutStorage::BeginMonitor/EndMonitor

After a call to ILayoutStorage::BeginMonitor, the compound file implementation takes any operation performed on the storage/stream objects as part of the desired access pattern. A call to ILayoutStorage::EndMonitor ends the monitoring. Multiple pairs of BeginMonitor/EndMonitor are permitted.

 © Microsoft Corporation 1996. All Rights Reserved.

Applications will usually use this mechanism to obtain the access pattern of embedded objects. The mechanism also enables generic layout tools that will simply launch a existing applications and monitor their access patterns.

Example

```
pRootStg->QueryInterface(IID_ILayoutStorage, &pLayout);

pLayout->BeginMonitor();
pRootStg->OpenStream(…, &pStream);
pStream->Read(…);
pStream->Seek(10480);
pStream->Read(…);
pRootStg->OpenStream(…, &pStream2);
pStream2->Seek(2048);
pStream2->Read(…);
pLayOut->EndMonitor();

// Write new compound file with desired layout
pLayout->ReLayoutDocfile(L"Optimized File");
```

4.1.3 Optimizing the file: ILayoutStorage::ReLayoutDocfile

After indicating the desired access pattern using one or both of the two methods described above, the application triggers the actual optimization by calling this ILayoutStorage::ReLayoutDocfile. If this function is not called before releasing the last pointer to the root storage the Compound File will not be altered.

4.1.4 Combining Scripting and Monitoring

```
pLayout->BeginMonitor();
(…)
pStream->Read(…);

pLayOut->LayoutScript(…);

pStream->Read();
(…)
pLayOut->EndMonitor();

pLayout->LayoutScript(…);
(…)
// Write new compound file with desired layout
pLayout->ReLayoutDocfile(L"Optimized File");
```

 © Microsoft Corporation 1996. All Rights Reserved.

5Index

ABINDF_ASYNCSTORAGE, 2
IBindStatusCallback, 2
IBindStatusCallback::OnDataAvailable, 2
IFillLockBytes, 4
ILockBytes, 4
Layout Tool, 10
StgCreateDocfileOnILockBytes, 2
StgOpenStorageOnILockBytes, 2
URL Moniker, 2