1COM Applications

All applications, that is, running programs that define a task or a process be they client or servers, have specific responsibilities. This chapter examines the roles and responsibilities of all COM applications and the necessary COM library support functions for those responsibilities.

In short, any application that makes use of COM, client or server, has three specific responsibilities to insure proper operation with other components:

1. On application startup, verify that the COM Library version is new enough to support the functionality expected by the application. In general, an application can use an updated version of the library but not an older one or one that has undergone a major version change.
2. On application startup, initialize the COM Library.
3. On application shutdown, uninitialize the COM Library to allow it to free resources and perform any cleanup operations as necessary.

Each of these responsibilities requires support from the COM Library itself as detailed in the following sections. For convenience, initialization and uninitialization are described together. Additional COM Library functions related to initialization and memory management are also given in this chapter.

1.1Verifying the COM Library Version

The COM Library defines a major version number and a minor version number and provide these in a header file that is compiled with the COM application. Any application must then compare these compiled numbers with the version of the available library and if the available library is incompatible the application cannot use COM. Similarly, a DLL should check the library version in its initialization code and fail loading if the library is incompatible or otherwise disable its COM functionality. The current major and minor version numbers are retrieved from COM Library with the function CoBuildVersion.

CoBuildVersion

DWORD CoBuildVersion(void)

Return the major and the minor version number of the Component Object Model library.

Argument	Type	Description
return value	DWORD	A 32 bit value whose high-order 16 bits are the major version number (rmm) and whose low-order 16 bits are the minor version number (rup).

An application or DLL can run against only one major version of the COM Library but can run against any minor version (possibly disabling specific minor features that are not available in a builds before a given minor number). Therefore during startup (initialization for DLLs), all COM applications must include code similar to the following:

```
DWORD    dwBuildVersion;
dwBuildVersion=CoBuildVersion();
if (HIWORD(dwBuildVersion)!=rmm)
   //Error:  Can't run against wrong major version
if (LOWORD(dwBuildVersion) < rup)
   //Disable features dependent on the rup version of COM (or simply fail)
//Continue initialization
```

1.2Library Initialization / Uninitialization

Once the application has determined that it can run against the currently available version of the COM Library, it must initialize the library through a function called CoInitialize. Calls made to CoInitialize must be matched with calls to CoUninitialize to allow the COM Library to perform any final cleanup.

CoInitialize

HRESULT CoInitialize(pReserved)

Initialize the Common Object Model library so that it can be used. With the exception of CoBuildVersion, this function must be called by applications before any other function in the library. Calls to CoInitialize must be balanced by corresponding calls to CoUninitialize. Typically, CoInitialize is called only once by the process that wants to use the COM library, although multiple calls can be made. Subsequent calls to CoInitialize return S_FALSE.

Argument	Type	Description
pReserved	void*	Reserved for future use. Presently, must be NULL.

Return Value	Meaning
S_OK	Success. Initialization has succeeded. This was the first initialization call in this process.
S_FALSE	Success. Initialization has succeeded, but this was not the first initialization call in this process.
E_UNEXPECTED	An unknown error occurred.

CoUninitialize

void CoUninitialize(void)

Shuts down the Component Object Model library, thus freeing any resources that it maintains. Since CoInitialize and CoUninitialize calls must be balanced, only the CoUninitialize call that corresponds to the CoInitialize call that actually did the initialization will uninitialize the library.

1.3Memory Management

As was articulated earlier in this specification, when ownership of allocated memory is passed through an interface, COM requires[1] that the memory be allocated with a specific "task allocator." Most general purpose access to the task allocator is provided through the IMalloc interface instance returned from CoGetMalloc. Simple shortcut allocation and freeing APIs are also provided in the form of CoTaskMemAlloc and CoTaskMemFree.

1.3.1IMalloc Interface

The *IMalloc* interface is an abstraction of familiar memory-allocation primitives that fit into the COM interface model. Like all other interface, it is derived from IUnknown and correspondingly includes the AddRef, Release, and QueryInterface member functions. The first three IMalloc-specific functions in this interface are merely simple abstractions of the familiar C-library functions malloc, realloc, and free.

```
[
   local,
   object,
   uuid(00000002-0000-0000-C000-000000000046)
]
interface IMalloc : IUnknown {
      void *    Alloc([in] ULONG cb);
      void *    Realloc([in] void * pv, [in] ULONG cb);
      void      Free([in] void* pv);
      ULONG     GetSize([in] void * pv);
      int       DidAlloc([in] void * pv);
      void      HeapMinimize(void);
      };
```

IMalloc::Alloc

void * IMalloc::Alloc(cb)

Allocate a memory block of at least cb bytes. The initial contents of the returned memory block are undefined. Specifically, it is not guaranteed that the block is zeroed. The block actually allocated may be

[1] In general, though, precisely, one can invent interfaces which choose to violate this rule. However, such interfaces are, for example, unlikely to have their remoting proxies and stubs generated with common tools.

larger than cb bytes because of space required for alignment and for maintenance information. If cb is 0, Alloc allocates a zero-length item and returns a valid pointer to that item. This function returns NULL if there is insufficient memory available.

Callers must always check the return from the this function, even if the amount of memory requested is small.

Argument	Type	Description
cb	ULONG	The number of bytes to allocate.
return value	void *	The allocated memory block, or NULL if insufficient memory exists.

IMalloc::Free

void IMalloc::Free(pv)

Deallocate a memory block. The pv argument points to a memory block previously allocated through a call to IMalloc::Alloc or IMalloc::Realloc. The number of bytes freed is the number of bytes with which the block was originally allocated (or reallocated, in the case of Realloc). After the call, the pv parameter is invalid, and can no longer be used. pv may be NULL, in which case this function is a no-op.

Argument	Type	Description
pv	void *	Pointer to the block to free. May be NULL.

IMalloc::Realloc

void * IMalloc::Realloc(pv, cb)

Change the size of a previously allocated memory block. The pv argument points to the beginning of the memory block. If pv is NULL, Realloc functions in the same way as IMalloc::Alloc and allocates a new block of cb bytes. If pv is not NULL, it should be a pointer returned by a prior call to IMalloc::Alloc.

The cb argument gives the new size of the block in bytes. The contents of the block are unchanged up to the shorter of the new and old sizes, although the new block may be in a different location. Because the new block can be in a new memory location, the pointer returned by Realloc is not guaranteed to be the pointer passed through the pv argument. If pv is not NULL and *cb* is 0, then the memory pointed to by pv is freed.

Realloc returns a void pointer to the reallocated (and possibly moved) memory block. The return value is NULL if the size is zero and the buffer argument is not NULL, or if there is not enough available memory to expand the block to the given size. In the first case, the original block is freed. In the second, the original block is unchanged.

The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than void, use a type cast on the return value.

Argument	Type	Description
pv	void *	Pointer to the block to reallocate. May be NULL.
cb	ULONG	The new size in bytes to allocate. May be zero.
return value	void *	The reallocated memory block, or NULL.

IMalloc::GetSize

ULONG IMalloc::GetSize(pv)

Return the size, in bytes, of the memory block allocated by a previous call to IMalloc::Alloc or IMalloc::Realloc on this memory manager.

Argument	Type	Description
pv	void *	The pointer to be tested. May be NULL, in which case -1 is returned.
return value	ULONG	The size of the allocated memory block

IMalloc::DidAlloc

int IMalloc::DidAlloc(pv)

This function answers as whether or not the indicated memory pointer pv was allocated by the given allocator, if the allocator is able to determine that fact (many memory allocators will not be able to do so).

The values 1 (one) and 0 (zero) are returned as "did alloc" and "did not alloc" answers respectively; -1 (minus one) is returned if the IMalloc implementation is unable to determine whether it allocated the pointer or not.

Argument	Type	Description
pv	void *	The pointer to be tested. May be NULL, in which case -1 is returned.
return value	int	-1, 0, 1

IMalloc::HeapMinimize

void IMalloc::HeapMinimize()

Minimize the heap as much as possible for this allocator by, for example, releasing unused memory in the heap to the operating system. This is useful in cases when a lot of allocations have been freed (using IMalloc::Free) and the application wants to release the freed memory back to the operating system so that it is available for other purposes.

1.3.2COM Library Memory Management Functions

CoGetMalloc

HRESULT CoGetMalloc(dwMemContext, ppMalloc)

This function retrieves from the COM library either the task memory allocator an optionally-provided shared memory allocator. The particular allocator of interest is indicated by the dwMemContext parameter. Legal values for this parameter are taken from the enumeration MEMCTX:

```
typedef enum tagMEMCTX {
    MEMCTX_TASK = 1,          // task (private) memory
    MEMCTX_SHARED = 2,        // shared memory (between processes)
    MEMCTX_MACSYSTEM = 3,     // on the mac, the system heap
    // these are mostly for internal use...
    MEMCTX_UNKNOWN = -1,      // unknown context (when asked about it)
    MEMCTX_SAME = -2,         // same context (as some other pointer)
  } MEMCTX;
```

MEMCTX_TASK returns the task allocator. If CoInitialize has not yet been called, NULL we be stored in ppMalloc and CO_E_NOTINITIALIZED returned from the function.

MEMCTX_SHARED returns an optionally-provided shared allocator; if the shared allocator is not supported, E_INVALIDARG is returned. When supported, the shared allocator returned by this function is an COM-provided implementation of IMalloc interface, one which allocates memory in such a way that it can be accessed by other process on the current machine simply by conveying the pointer to said applications.[2] Further, memory allocated by this shared allocator in one application may be freed by the shared allocator in another. Except when a NULL pointer is passed, the shared memory allocator never answers -1 to IMalloc::DidAlloc; it always indicates that either did or did not allocate the passed pointer.

[2] That is, the memory resides at the same address in all processes.

Argument	Type	Description
dwMemContext	DWORD	A value from the enumeration MEMCTX.
ppMalloc	IMalloc **	The place in which the memory allocator should be returned.

Return Value	Meaning
S_OK	Success. The requested allocator was returned.
CO_E_NOTINITIALIZED	The COM library has not been initialized.
E_INVALIDARG	An invalid argument was passed.
E_UNEXPECTED	An unknown error occurred.

CoGetCurrentProcess

DWORD CoGetCurrentProcess(void)

Return a value unique to the current process. More precisely, return a value unique to the current process to the degree that it will not be reused until 2^{32} further processes have been created on the current workstation.

Argument	Type	Description
return value	DWORD	A value unique to the current process.

CoTaskMemAlloc

LPVOID CoTaskMemAlloc(cb)

Semantically identical to retrieving the current task allocator with CoGetMalloc, invoking IMalloc::Alloc on that pointer with the same parameters, then releasing the IMalloc pointer.

Argument	Type	Description
cb	ULONG	The number of bytes to allocate.
return value	void *	The allocated memory block, or NULL if insufficient memory exists.

CoTaskMemFree

void CoTaskMemFree(pv)

Semantically identical to retrieving the current task allocator with CoGetMalloc, invoking IMalloc::Free on that pointer with the same parameters, then releasing the IMalloc pointer.

Argument	Type	Description
pv	void *	Pointer to the block to free. May be NULL.

CoTaskMemRealloc

void CoTaskMemRealloc(pv, cb)

Semantically identical to retrieving the current task allocator with CoGetMalloc, invoking IMalloc::Realloc on that pointer with the same parameters, then releasing the IMalloc pointer.

Argument	Type	Description
pv	void *	Pointer to the block to reallocate. May be NULL.
cb	ULONG	The new size in bytes to allocate. May be zero.
return value	void *	The reallocated memory block, or NULL.

1.4Memory Allocation Example

An object may need to pass memory between it and the client at some point in the object's lifetime—this applies to in-process as well as out-of-process servers. When such a situation arises the object must use the task allocator as described in Chapter 2. That is, the object must allocate memory whose ownership is transferred from one party to another through an interface function by using the local task allocator.

CoGetMalloc provides a convenient way for objects to allocate working memory as well. For example, when the TextRender object (see Chapter 3, "Designing and Implementing Objects") under consideration in this document loads text from a file in the function IPersistFile::Load (that is, CTextRender::Load) it will want to make a memory copy of that text. It would use the task allocator for this purpose as illustrated in the following code (unnecessary details of opening files and reading data are omitted for simplicity):

```
//Implementation of IPersistFile::Load
HRESULT CTextRender::Load(char *pszFile, DWORD grfMode) {
    int      hFile;
    DWORD    cch;
    IMalloc * pIMalloc;
    HRESULT  hr;

    /*
     * Open the file and seek to the end to set the
     * cch variable to the length of the file.
     */

    hr=CoGetMalloc(MEMCTX_TASK, &pIMalloc);

    if (FAILED(hr))
        //Close file and return failure

    psz=pIMalloc->Alloc(cch);
    pIMalloc->Release();

    if (NULL==psz)
        //Close file and return failure

    //Read text into psz buffer and close file

    //Save memory pointer and return success
    m_pszText=psz;
    return NOERROR;
    }
```

If an object will make many allocations throughout it's lifetime, it makes sense to call CoGetMalloc once when the object is created, store the IMalloc pointer in the object (m_pIMalloc or such), and call IMalloc::Release when the object is destroyed. Alternatively, the APIs CoTaskMemAlloc and its friends may be used.