Note: This file is also saved in Rich Text Format as 16BITOLE.RTF. We recommend that you use 16BITOLE.RTF if you have a word processor that can read Rich Text Format files.

Notes Concerning 16-Bit OLE

This edition of *Inside OLE* focuses on 32-bit operating systems. There are, however, still some important considerations for 16-bit OLE components, which we'll cover in this appendix.

Chapter 2

Globally Unique Identifiers (GUIDs)

In 16-bit OLE, *CoCreateGuid* is a duplicate implementation of *UUIDCreate*. It is not available elsewhere in the system, as it is under Win32.

Interface Definitions

The Windows 3.1 SDK for 16 bits uses a set of macros that expand to the same code; C and C++ differences are hidden in the macro expansions:

```
#undef  INTERFACE
#define INTERFACE   IUnknown
DECLARE_INTERFACE(IUnknown)
   {
   STDMETHOD(QueryInterface) (THIS_ REFIID riid, LPVOID FAR* ppvObj) PURE;
   STDMETHOD_(ULONG,AddRef) (THIS) PURE;
   STDMETHOD_(ULONG,Release) (THIS) PURE;
   };
```

Calling Conventions and Parameter Types

In 16-bit OLE, the standard call type is __*cdecl* rather than __*stdcall*, which is used in 32-bit OLE. Header files also include the __*far* keyword, which is necessary only on 16-bit platforms.

The macro *STDMETHOD(method)* expands to *HRESULT __cdecl method* whereas *STDMETHOD_(type, method)* expands to *type __cdecl method*. The macros STDMETHOD and STDMETHODIMP expand identically but also include __*export*.

Return Types: HRESULT *and* SCODE

Note #1: HRESULT and SCODE are different types on 16-bit platforms. To accommodate the differences, a few COM functions that are obsolete for Win32 are still used in the sample code to retain compatibility with Win16 where possible. These are *ResultFromScode*, which turns an SCODE into an HRESULT, *GetScode* which extracts an SCODE from an HRESULT, and *PropagateResult*, which changes the error value without changing the facility code so you can propagate an error from a low layer while changing the exact error code. Obviously, if you're writing only 32-bit code, you can ignore these functions completely. Otherwise, a little more work is involved because you have to use *ResultFromScode* when returning an error value and *GetScode* when you want to check for specific error or success codes. The only exception is NOERROR, which is interchangeable.

Note #2: *FormatMessage* is only available with 32-bit OLE.

Aggregation

The last two rules are different under 16-bit OLE: When a 16-bit outer object queries for an interface from the inner object, the outer object must immediately call *Release* through that pointer to fix the outer object's reference count in order to avoid a circular reference. In addition, the inner object must guarantee that all interface pointers remain valid as long as the object itself is valid, even if those interfaces have a zero reference count. The 32-bit change greatly simplifies the aggregation of remote (distributed) objects, which will never be an issue under 16-bits.

COM/OLE Task Requirements

There are two additional steps that 16-bit EXEs must perform prior to COM/OLE library initialization; the second also applies to 16-bit DLLs:

1. Call the Windows API *SetMessageQueue(96)* to set your application's message queue size to 96 messages, if possible. This is the size recommended for LRPC handling. This function is not necessary in Win32 because Win32 message queues size dynamically. In the sample code, the macro SETMESSAGEQUEUE(96) is used to make this call. This macro expands to nothing under Win32.

2. Verify the library build version by calling either *CoBuildVersion* or *OleBuildVersion*. (Which one doesn't matter.) This step is also compiled conditionally in the sample code.

The following two sections describe these steps in more detail.

Enlarge the Message Queue

OLE's LRPC implementation works on top of the Windows API function *PostMessage*. In a nutshell, when the user of an object in another application calls one of the object's member functions, the function generates an LRPC call, which in actuality is a *PostMessage* call from the first application's process space to the second application. To handle all the possible *PostMessage* traffic, Microsoft recommends that OLE applications with even the slightest chance of engaging in LRPC calls call *SetMessageQueue* set to 96 on startup. Something such as the following should, in fact, be your first step inside *WinMain* to ensure that no messages exist in your queue because *SetMessageQueue* will destroy anything already there:

```
int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hInstPrev
   , LPSTR pszCmdLine, int nCmdShow)
   {
   [variables, but NO code]
   int      cMsg=96;
  #ifndef WIN32
   //Enlarge the queue as large as we can starting from 96
   while (!SetMessageQueue(cMsg) && (cMsg-=8));
  #endif
   [Initialization code, message loop, etc.]
   }
```

If you don't enlarge your message queue sufficiently, the Com Library might reject some LRPC calls when your queue is full. Enlarging your message queue provides sufficient space for LRPC traffic.

This book's sample code uses the macro SETMESSAGEQUEUE (in INC\BOOK1632.H), which isolates the code above. The macro expands to nothing for Win32 targets but includes the SetMessageQueue call for 16-bit targets.

Verify the Library Build Version

Before using any other COM API function (*Co**) on Windows 3.1, a task should call *CoBuildVersion* (or *OleBuildVersion*) to get major and minor build numbers in a returned DWORD. This allows you to check whether the libraries on the system are new enough for your component's use. This step is necessary because the OLE DLLs were not shipped as part of the Windows 3.1 platform, but they are in 32-bit platforms such as Windows NT (3.5 and later) and Windows 95. When you do call this function, the high-order word of the return value is a major version number, and the low-order word is the minor version number.

This verification of version numbers applies not only to applications but also to DLLs that depend on certain versions of the OLE DLLs. Dependent modules should perform the same checks described here from within their entry code, failing initialization if the version numbers do not match expectations.

A component can run against only one major version of the libraries, but it can run against any minor version. The version numbers you can run against are compiled into your application as the symbols *rmm* (major) and *rup* (minor), defined in OLE2VER.H. (There is also a *rmj* symbol, which might look like the "major" number but is unfortunately not used this way.) These numbers are *build numbers* not *product release numbers*. Do not depend on any interpretation of these numbers. With these numbers, you must compare your *rmm* to the major version of the libraries, and if they do not match, you must fail loading your application as shown below.

```
#include <compobj.h>    //For Ole... functions, use OLE2.H
#include <ole2ver.h>
.
.
.
DWORD    dwVer;

dwVer=CoBuildVersion();    //Or OleBuildVersion

if (rmm==HIWORD(dwVer))
   {    //Major versions match.
       if (rup <= LOWORD(dwVer))
       {
       //Library is newer than or as old as the app; use normally.
       }
   else
       {
       /*
        * Component was written for newer libraries. Disable features
        * that depend on API or bug fixes in newer libraries or
        * simply fail altogether.
        */
       }
   }
else
   //Major version mismatch; fail loading component.
```

Minor version numbers are useful to applications that want to know whether the libraries they've loaded contain a particular function or have a specific bug fix. Let's say minor version

12 of OLE added a function that improves performance over minor version 11. If I load the minor version 11 libraries, I cannot attempt to call that version 12 function. If, however, I find that I am running against minor version 12, I can take advantage of what's available.

Call CoInitialize or OleInitialize

In 16-bit OLE, the argument to either of these functions is a pointer to an *IMalloc* implementation through which the caller can customize the task memory allocator. This allocator would then be returned through *CoGetMalloc*.

Memory Management

Note #1: Under 16-bit OLE, you can pass MEMCTX_TASK or MEMCTX_SHARED to *CoGetMalloc*. The latter accesses a shared memory allocator that is not supported under 32-bit OLE.

Note #2: The file INC\BOOK1632.H contains a macro implementation of *CoTaskMemFree* which is legal because it has no return value. But you can use this as a model for implementing your own *CoTaskMemAlloc* and *CoTaskMemRealloc* functions for 16-bits if you want.

The MALLOC1 and MALLOC2 Samples

The first edition of this book included illustrations of the Windows 3.1 HeapWalk utility, which shows the memory blocks quite readily. HeapWalk doesn’t exist in the Win32 SDK, so you have to look at the memory in a debugger.

Aggregation: Outer Object (KOALAA.CPP)

A 16-bit object that is not aggregatable itself has to fix its reference count in the same way. In addition, a 16-bit call to *m_pIAnimal->Release* here would need to be wrapped in *m_cRef++* and *m_cRef--* as well.

In addition, the 16-bit aggregation rules (described earlier in this file under "Aggregation") that differ from the 32-bit rules appear in code as follows:

```
//Cache a pointer to Animal's IAnimal
m_pIAnimal=NULL;
hr=m_pIUnknownAnimal->QueryInterface(IID_IAnimal, (void **)&m_pIAnimal);

if (SUCCEEDED(hr))
   {
   m_cRef++;
   m_pIAnimal->Release();     //m_pIAnimal still valid
   m_cRef--;
   }
```

16-bit code can use the 32-bit rules for caching pointers only if the code knows that the outermost unknown is guarded with the 32-bit technique. An object that is nonaggregatable can safely use the 32-bit method, but 16-bit aggregates that are aggregatable themselves cannot; they must use this older method.

The artificial reference count in the outer object's *Release* works fine with 16-bit code as well as 32-bit code.

Chapter 3

Type Library and Element Attributes

Under 16-bit OLE, the NLS API is not part of the operating system, so the API is included as part of OLE itself. This API is documented in the *OLE Programmer's Reference.*

Type Library Deployment

Under 16-bit OLE, you can have only one type library per module and a bug prevents loading a type library from an EXE. Both limitations are removed in 32-bit OLE, in which you can have multiple libraries per module, EXE or DLL.

Loading and Using a Type Library

LoadTypeLibFromResource is first available in OLE 2.02 on 16 and 32 bits. Under 16-bit OLE, this function can load only from DLLs and can load only one library given the resource identifier of one. Under 32-bit OLE, this function can load from DLLs or EXEs given any resource identifier.

Chapter 5

Introduction

Under Win16 the COM Library stands alone and is called COMPOBJ.DLL.

Where the Wild Things Are

16-32 bit Interoperability Between Clients and Components: Under a 32-bit operating systems such as Windows NT and Windows 95, or under Windows 3.1 with Win32s, it is possible to have clients and components that are a mixture of 16-bit and 32-bit versions. To accommodate this possibility, 32-bit OLE on Win32 platforms and under Win32s supports interoperability between any combination of like-bitness and any combination involving a local server. The table below summarizes the possible interoperability scenarios

	Server:			
Client	16 In-Proc	16 Local	32 In-Proc	32 Local

16-bit	Yes	Yes	No*	Yes
32-bit	No	Yes	Yes	Yes

* Windows NT 3.5 actually allows this combination for OLE-defined interfaces other than *IDispatch*, but such support doesn't exist in Windows 95 or in Windows NT 3.51 (a required upgrade). It is simply a complex testing and development problem, and there is little demand for the feature.

All of the local server variations are possible because the marshaling layer reduces everything to RPC, so the bitness of either process is not an issue. The like-bitness combinations for in-process servers work for custom interfaces as well. However, even if you provide your own marshaling proxy and stub for a custom interface, OLE allows it only between a 32-bit client and a 32-bit local server. In other words, custom interface marshaling cannot be used in a 16-16, 16-32, or 32-16 scenario.

Even when 16-32 bit interoperability is supported, there is the possibility that a 32-bit *int* or *UINT* argument passed to an interface member function could be truncated when it arrives in a 16-bit process as a 16-bit *int* or *UINT*. This is not known to be a problem with the OLE-defined interfaces but is something to consider if you encounter odd problems with incorrect argument values.

Also, 32-bit OLE gives different preferences to different types of servers depending on the type of client process. With a CLSCTX_SERVER passed to *CoCreateInstance* from a 32-bit client, OLE will attempt to load a server in the order of InprocServer32, LocalServer32, and then LocalServer16. For a 16-bit client, the order is InprocServer (16-bit), LocalServer32, and then LocalServer16.

The Mechanisms of Server, Client, and COM

16-bit executables must also check COM version numbers and enlarge the message queue as described earlier for Chapter 2.

Registry Entries

The 16-bit version of COM requires the registry keys InprocServer*,* InprocHandler*,* and LocalServer instead of those with the 32 suffix. In addition, 16-bit COM does not enforce the requirement for full pathnames.

Expose the Class Factory

Use the HeapWalk or WPS tools under Windows 3.1 to view loaded modules.

Chapter 6

Introduction

As mentioned earlier under "Where the Wild Things Are," *int* or *UINT* function arguments being passed from a 32-bit address space to a 16-bit address space might be truncated, amounting to a real "transmission loss." For that reason, Microsoft discourages any use of variable-size argument types, with the exception of HWND and other "handle" arguments, in any sort of interface function, which you'll see enforced in all OLE-defined interfaces. The problem usually doesn't show with handles, however, because handle values usually have 0 in the upper 16 bits anyway. Still, keep this in mind when designing your own interfaces.

Custom Interfaces and Standard Marshaling

The *ProxyStubClsid32* entry appears as *ProxyStubClsid* for 16-bit marshalers. Also, under 32-bit systems, 16-16 custom interfaces are not supported unless the 16-bit DLLs implement the 32-bit interfaces such as *IRpcChannelBuffer*.

The Easy Way: The MIDL Compiler

Standard marshaling support for custom interfaces must be done manually under 16 bits. There is no MIDL compiler available.

Limitations of Outgoing Calls

There are two other relevant errors for 16-bit systems:

RPC_E_CANTPOST_INSENDCALL

The caller is dispatching an intertask *SendMessage* call to a 16-bit process and cannot make an outgoing asynchronous call with *PostMessage*.

RPC_E_CANTTRANSMIT_CALL

The call was not transmitted properly to a 16-bit process because its message queue was full and was not emptied after yielding.

The OLE UI Library

Under Windows 3.1, use the MFCOLEUI.DLL included with Visual C++ 1.50 and 1.51 with the import library MFCOLEUI.LIB. You can also compile your own 16-bit UI DLL using sample code provided in the 16-bit OLE SDK.

A Simple Client-Side Message Filter: ObjectUser2

OLE doesn't provide marshaling support for *IPersist* by itself under most of its 16-bit versions. Therefore, to make this sample work, you'll need to modify EKoala3 to respond to *IPersistStorage*, returning only an *IPersist* pointer, and also modify ObjectUser2 to ask for *IPersistStorage* but only call *GetClassID*. This is a totally illegal thing to do in shipping code, but it will get you marshaling support for *IPersist.* We get away with it here because the client and server both know about this special use of the interface.

Implementing a Custom Interface with MIDL

These samples will not compile on 16 bits: MIDL is not provided there.

Chapter 9

Custom Monikers and Custom Marshaling

IMoniker pointers cannot be marshaled themselves under most versions of 16-bit OLE, with the exceptions of some later versions of OLE 2.02. Marshaling the persistent data—that is, an *IStream* that holds the data—is supported because it is required to support custom marshaling in the first place.

Creating and Using Standard Monikers: LinkUser

The samples only compile and run under 32 bits because they require the custom interface *IDescription*.

Chapter 10

The STGMEDIUM *Structure*

Note #1: TYMED_ENHMF is not valid under 16-bit Windows because enhanced metafiles are exclusive to Win32.

Note #2: TYMED_FILE is freed using *OpenFile(OF_DELETE, ...)* under 16-bit Windows.

Implementing a Data Object: DDataObj and EDataObj

16-bit compilations of EDataObj's code in *AdviseWndProc* will include a call to *PeekMessage* as well as to *TranslateMessage* and *DispatchMessage* for each iteration through the loop:

```
while (TRUE)
   {
  #ifdef EXEDATAOBJECT
  #ifndef WIN32
   MSG     msg;
   if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
       {
       TranslateMessage(&msg);
       DispatchMessage(&msg);
       }
   else
  #endif
  #endif
       {
       pDO->m_pIDataAdviseHolder->SendOnDataChange
           (pDO->m_pImpIDataObject, 0, ADVF_NODATA);
       if (++i >= iAdvise)
           break;
       }
   }
```

This allows the 16-bit LRPC layer to process some of the asynchronous *IAdviseSink* calls because LRPC uses *PostMessage* for such a purpose. If the data object never called *PeekMessage*, it would quickly fill the message queue and OLE would begin rejecting new calls. Obviously we don't need to do this with a DLL case, nor under a multitasking Win32 system in which the RPC layer for even a local server is not dependent on message loops.

Chapter 11

The RECTL Structure

On 16-bit platforms, RECTL is still a 32-bit structure, as shown in Chapter 11, but RECT is composed of 16-bit fields. The macros RECTLFROMRECT and RECTFROMRECTL in

INC\INOLE.H are independent of the underlying platform and offer a convenient means to convert between the two types.

Chapter 13

Two Issues

A third issue applies to 16-bit OLE.

***DoDragDrop* and Fast Keyboard Repeat Rates:** *DoDragDrop* calls *PeekMessage* internally to remove keyboard and mouse messages (client and nonclient alike). If a message exists, calls are made to source and target and the loop continues. On a machine with a very fast keyboard repeat rate, it is possible to flood the message queue with WM_KEYDOWN messages for VK_CONTROL, so that every iteration through the loop will catch a keyboard message and continue in the loop once again. This means that in some circumstances, letting up on the mouse button with the Ctrl key still down doesn't cause a drop until you release that key as well.

Chapter 14

The VARIANT and VARIANTARG Structures

VT_UI1 was not originally supported in a VARIANT under 16-bits. Such a data type is useful for creating arrays of binary data with the guarantee that nothing will attempt to perform ANSI to Unicode translations, which may happen with a *char* type.

Exceptions

Error objects are available only under 16-bit OLE with OLE 2.02 and later.

Active Automation Objects

Prior to OLE 2.02 on 16-bits, *RegisterActiveObject* could register only with a strong lock—that is, ACTIVEOBJECT_WEAK was not recognized.

The Dual Interface

Dual interfaces are available only under 16-bit OLE with OLE 2.02 and later. The dual interface is another reason that error objects were created because these interfaces allow a vtable-based interface to raise Automation exceptions.

Five Variations on the Theme of Implementing a Simple Automation Object

Variations III and IV require OLE 2.02 on 16-bit platforms.

Also, you will experience errors if you attempt to use a 32-bit Beeper DLL with a 16-bit controller such as Visual Basic 3 or DispTest. Match bitness when attempting to use such a controller with samples from this book.

Variation III: Exceptions Through Error Objects

Because error objects are supported only with OLE 2.02 and later, Beeper3 requires the header files and import libraries from OLE 2.02. Also note that the Thread-Local Storage described in the text for the Win32 version of this sample simply uses a global variable when compiled for 16-bits.

Variation IV: A Dual Interface

As with error objects, dual interfaces are available only with OLE 2.02 and later, so Beeper4 requires the header files and import libraries from OLE 2.02.

Chapter 17

Registry Entries for Content Objects

The default handler for 16-bit OLE is OLE2.DLL, which should appear under the InprocHandler entry if the server has no specific handler of its own.

Chapter 24

Control-Specific Registry Entries

16-bit controls use “ToolboxBitmap” instead of “ToolboxBitmap32”.

Appendix B:The Details of Standard Marshaling

The RPC Channel

Under 16-bit OLE, the RPC Channel implements the interface *IRpcChannel* instead of *IRpcChannelBuffer*. The former is considered obsolete.