Understanding DCOM

By William Rubin and
Marshall Brain

.....

Prentice Hall books are widely used by corporations and government agencies for training, marketing and resale.

The publisher offers discounts on this book when ordered in bulk quantities. For more information, contact Corporate Sales Department, Phone: 800-382-3419; fax: 201-236-7141;
email: corpsales@prenhall.com
Or write Corporate Sales Department, Prentice Hall PTR, One Lake Street, Upper Saddle River, NJ 07458.

ISBN 0-13-095966-9

CONTENTS

Preface xiii

ONE The Basics of COM *1*

Classes and Objects 1

How COM Is Different 3
COM can Run Across Processes 3
COM Methods Can Be Called Across a Network 4
COM Objects Must Be Unique Throughout the World 5
COM is Language Indpendent 5

COM Vocabulary 5

The Interface 7
Interfaces Isolate the Client From the Server 8
Imagine a Component 10
What's in a Name? 10
The Source of All Interfaces - IUnknown 10

A Typical COM Object 11

How to Be Unique - the GUID 12

A COM Server 14

Interactions Between Client and Server 15

Summary 16

TWO Understanding the Simplest COM Client *19*

Four Steps to Client Connectivity 20
Initializing the COM Subsystem: 21
Query COM for a Specific Interface 22
Execute a Method on the Interface. 24
Release the Interface 24

Summary 25

THREE Understanding a Simple COM Server *27*

Where's the Code? 28
Building a DLL-Based (In-Process) COM Server 29
Creating the Server Using the ATL Wizard 30
Adding a COM Object 33
Adding a Method to the Server 36
Running the Client and the Server 40
Summary 41

FOUR Creating your own COM Clients and Servers *43*

Server Side 43
Client Side 45

FIVE Understanding ATL-Generated Code *55*

The Main C++ Module 56
Object Maps 58
Export File 58
The COM Object - "CBeepObj" 60
Object Inheritance 61
The Class Definition 62
The Method 63
Server Registration 64
Registry Scripts 65
Summary 66

SIX Understanding the Client and Server *67*

Principles of COM 67
COM is About Interfaces 68
COM is Language-Independent 68
COM is Built Around the Concept of Transparency 69
Interfaces are Contracts Between the Client and Server 69
Software Changes. Interfaces Don't 70

Activation 71

More About Interfaces 73
VTABLES - Virtual Function Tables 75
The Class Factory 77
Singleton Classes 79
Understanding QueryInterface 81
Reference Counting with AddRef and Release 82

Method Calls 85
COM Identifiers: CLSID AND IID 87
CLSCTX -- Server Context 88

Inheritance 88

Summary 89

SEVEN An Introduction to MIDL *91*

Origins of the MIDL Compiler 91
Precisely Defining Interfaces with the IDL Language 92
MIDL Generated Headers 94
Automatically Generated Proxy/Stub Modules 94
Automatic Creation of Type Libraries 95

The IDL Language 95
Interfaces and Methods in IDL 97
The Component Class in IDL 100
Type Libraries in IDL 102

MIDL Post-Processing 103

Summary 105

EIGHT Defining and Using Interfaces *107*

Base Types 108
Attributes 109
Double Parameters 112
Boolean Parameters 113
Working with Strings 113
Arrays 119
Structures and Enumerations 121
Summary 123

NINE OLE Automation and Dual Interfaces *125*

IDL Definitions 126
The IDispatch Interface 127
Using Invoke 133
Using Type Libraries for Early Binding 136
Dual Interfaces 137
There is no Proxy/Stub DLL for Dispatch Interfaces 140
Properties 140
Adding Properties with the Class Wizard 142
Methods 144
The ISupportErrorInfo Interface 144
Summary 149

TEN COM Threading Models *151*

Synchronization and Marshaling 151
Threading Models 153
Apartment, Free, and Single Threads 155

The ATL Wizard and Threading Models 156
Apartment Threads 158
Single Threads 159
Free Threaded Servers 160
Both 161
Marshaling Between Threads 162
Using Apartment Threads 163
Free Threading Model 164
Testing the Different Models 165
Summary 166

ELEVEN The COM Registry *167*
The COM Registry Structure 168
Registration of CLSIDs 171
Registration of ProgIDs 172
Registration of AppIDs 174
Self-Registration in ATL Servers 174
The RGS File 175
Automatic Registration of Remote Servers 177
In-Process Servers 178
Using the Registry API 178
Summary 178

TWELVE Callback Interfaces *181*
Client and Server Confusion 183
Custom Callback Interfaces 183
A Callback Example 185
Create the Server 185
Add a COM Object to the Server 186
Adding the ICallBack Interface to IDL 187

Modify the Header 187
Adding the Advise Method to the Server 188
Adding the UnAdvise Method 189
Calling the Client from the Server 189
The Client Application 191
Create the Client Dialog Application 191
Adding the Callback COM Object 192
Linking to the Server Headers 194
COM Maps 194
Implementing the Callback Method 195
Adding the Object Map 195
Connecting to the Server 196
Cleaning Up 199
Adding the OnButton Code 199
A Chronology of Events 201
A Multi-Threaded Server 203
Starting the Worker Thread 205
Marshaling the Interface Between Threads 206
Starting the Worker Thread: Part 2 207
A Simple Worker Thread Class 208
Implementing the Worker Thread 209
All Good Threads Eventually Die 211
Summary 211

THIRTEEN Connection Points *213*
Modifying the Callback Server 215
Adding Connection Points to the Client Program 220
Add the Callback Object to the Client 221
Modifying the CpClient Application 221
Registering With the Server's Connection Point Interface 222
Adding the Now and Later Buttons 226
Using the Connection Point - the Server Side 226
Adding the Later2 Method 228
Summary 228

FOURTEEN Distributed COM *229*

An Overview of Remote Connections 229
Converting a Client for Remote Access 231
Adding Security 234
Security Concepts 234
Access Permissions 235
Launch Permissions 236
Authentication 237
Impersonation 237
Identity 238
Custom Security 239
CoInitializeSecurity 239
Disconnection 242
Using the Registry for Remote Connections 243
Installing the Server on a Remote Computer 244

FIFTEEN ATL and Compiler Support *245*

C++ SDK Programming 245
MFC COM 246
ATL - The Choice for Servers 246
Basic Templates 247
A Simple Template Example 248
Template Classes 250
Native Compiler Directives 253
The #IMPORT Directive 253
Namespace Declarations 254
Smart Interface Pointers 255
Smart Pointer Classes 256
Watch Out for Destructors 257
Smart Pointer Error Handling 258
How the IMPORT Directive Works 260
Raw and Wrapper Methods 260
Summary 261

SIXTEEN Other Topics *263*

Errors 263
Information Code 265
Facility Code 265
Customer Code Flag and Reserved bits 266
Severity Code 266
Looking Up HRESULTS 266
SCODES 267

Displaying Error Messages 267
Using FormatMessage 268

Aggregation and Containment 269

Building a COM Object with MFC 271
Adding Code for the Nested Classes 273
Accessing the Nested Class 275

APPENDIX COM Error Handling *277*

Sources of Information 278

Common Error Messages 279

DCOM Errors 285
Get It Working Locally 285
Be Sure You Can Connect 286
Try Using a TCP/IP Address 287
Use TRACERT 287
Windows 95/98 Systems Will Not Launch Servers 288
Security is Tough 288

Using the OLE/COM Object Viewer 289

Index 291

PREFACE

The goal of this book is to make COM and DCOM comprehendible to a normal person. If you have tried to learn COM and found its complexity to be totally unbelievable, or if you have ever tried to work with COM code and felt like you needed a Ph.D. in quantum physics just to get by, then you know exactly what this goal means. This book makes COM simple and accessible to the normal developer.

To meet the goal, this book does many things in a way that is much different from other books. Here are three of the most important differences:

1. This book is designed to clarify rather than to obfuscate. The basic principles of COM are straightforward, so this book starts at the beginning and presents them in a straightforward manner.
2. This book uses the simplest possible examples and presents them one concept at a time. Rather than trying to cram 116 concepts into a single 50 page example program, we have purposefully presented just one concept in each chapter. For example, chapter 2 shows you that you can create a complete, working, fully functional COM client with 10 lines of code. And when you look at it, it will actually make sense!
3. This book is not 1,200 pages long. You can actually make your way through this entire book and all of its examples in a handful of days. Once you have done that you will know and understand all of the vocabulary and all of the concepts needed to use COM on a daily basis.

Think of this book as the ideal starting point. Once you have read this book, all of the COM articles in the MSDN CD and all of the information on the Web will be understandable to you. You will be able to expand your knowledge rapidly. You will have the perfect mental framework to allow you to make sense of all the details.

Each chapter in this book explains an important COM topic in a way that will allow you to understand it. Here is a quick tour of what you will learn:

- Chapter 1: This chapter introduces you to the COM vocabulary and concepts that you need in order to get started.
- Chapter 2: This chapter presents a simple, working COM client. The example is only about 10 lines long. You will be amazed at how easy it is to connect to a COM server!
- Chapter 3: This chapter shows that you can create a complete COM server with the ATL wizard and about 6 lines of code. You can then connect client to server.
- Chapter 4: The previous two chapters will stun you. They will demonstrate that you can create complete and working COM systems with just 15 or 20 lines of code. And you will actually be able to understand it! This chapter recaps so that you can catch your breath, and shows you some extra error-handling code to make problem diagnosis easier.
- Chapter 5: This chapter delves into the code produced by the ATL wizard so that it makes sense.
- Chapter 6: This chapter gives you additional detail on the interactions between client and server so that you have a better understanding of things like singleton classes and method calls.
- Chapter 7: This chapter introduces you to MIDL and the IDL language.
- Chapter 8: This chapter shows you how to use MIDL to pass all different types of parameters.
- Chapter 9: This chapter shows you how to access your COM servers from VB and other languages.

- Chapter 10: This chapter clarifies the COM threading models. If you have ever wondered about “apartment threads”, this chapter will make threading incredibly easy!
- Chapter 11: This chapter uncovers the link between COM and the registry so you can see what is going on.
- Chapter 12: This chapter demystifies COM callbacks so you can implement bi-directional communication in your COM applications.
- Chapter 13: This chapter explains connection points, a more advanced form of bi-directional communication.
- Chapter 14: This chapter shows how to use your COM objects on the network and delves into a number of security topics that often get in the way.
- Chapter 15: This chapter further clarifies ATL, smart pointers, import libraries and such.
- Chapter 16: This chapter offers a collection of information on things like COM error codes and MFC support for COM.
- Error Appendix: Possibly the most valuable section of the book, this appendix offers guidelines and strategies for debugging COM applications that don’t work. COM uses a number of interacting components, so bugs can be hard to pin down. This chapter shows you how!

Read this book twice. The first time through you can load your brain with the individual concepts and techniques. The second time through you can link it all together into an integrated whole. Once you have done that, you will be startled at how much you understand about COM, and how easy it is to use COM on a daily basis!

For additional information, please see our web site at:

http://www.iftech.com/dcom

It contains an extensive resource center that will further accelerate your learning process.

ONE

The Basics of COM

Understanding how COM works can be intimidating at first. One reason for this intimidation is the fact that COM uses its own vocabulary. A second reason is that COM contains a number of new concepts. One of the easiest ways to master the vocabulary and concepts is to compare COM objects to normal C++ objects to identify the similarities and differences. You can also map unfamiliar concepts from COM into the standard C++ model that you already understand. This will give you a comfortable starting point, from which we'll look at COM's fundamental concepts. Once we have done this, the examples presented in the following sections will be extremely easy to understand.

Classes and Objects

Imagine that you have created a simple class in C++ called xxx. It has several member functions, named MethodA, MethodB and MethodC. Each member function accepts parameters and returns a result. The class declaration is shown here:

```
class xxx {
public:
  int MethodA(int a);
  int MethodB(float b);
  float MethodC(float c);
};
```

The class declaration itself describes the class. When you need to use the class, you must create an instance of the object. Instantiations are the actual objects; classes are just the definitions. Each object is created either as a variable (local or global) or it is created dynamically using the new statement. The new statement dynamically creates the variable on the heap and returns a pointer to it. When you call member functions, you do so by dereferencing the pointer. For example:

```
xxx *px;            // pointer to xxx class
px = new xxx;       // create object on heap
px->MethodA(1);     // call method
delete px;          // free object
```

It is important for you to understand and recognize that COM follows this same objected oriented model. COM has classes, member functions and instantiations just like C++ objects do. Although you never call new on a COM object, you must still create it in memory. You access COM objects with pointers, and you must de-allocate them when you are finished.

When we write COM code, we won't be using new and delete. Although we're going to use C++ as our language, we'll have a whole new syntax. COM is implemented by calls to the COM API, which provides functions that create and destroy COM objects. Here's an example COM program written in pseudo-COM code.

```
ixx *pi                    // pointer to COM interface
CoCreateInstance(,,,,&pi)   // create interface
pi->MethodA();              // call method
pi->Release();              // free interface
```

In this example, we'll call class ixx an "interface". The variable pi is a pointer to the interface. The method CoCreateInstance creates an instance of type ixx. This interface pointer is used to make method calls. Release deletes the interface.

I've purposely omitted the parameters to CoCreateInstance. I did this so as not to obscure the basic simplicity of the program. CoCreateInstance takes a number of arguments, all of which need some more detailed coverage. None of that matters at this moment, however. The point to notice is that the basic steps in calling a COM object are identical to the steps taken in C++. The syntax is simply a little different.

Now let's take a step back and look at some of the bigger differences between COM and C++.

How COM Is Different

COM is not C++, and for good reason. COM objects are somewhat more complicated then their C++ brethren. Most of this complication is necessary because of network considerations. There are four basic factors dictating the design of COM:

- C++ objects always run in the same process space. COM objects can run across processes or across computers.
- COM methods can be called across a network.
- C++ method names must be unique in a given process space. COM object names must be unique throughout the world.
- COM servers may be written in a variety of different languages and on entirely different operating systems, while C++ objects are always written in C++.

Let's look at what these differences between COM and C++ mean to you as a programmer.

COM can Run Across Processes

In COM, you as the programmer are allowed to create objects in other processes, or on any machine on the network. That does not mean that you will always do it (in many cases you won't).

However, the possibility means that you can't create a COM object using the normal C++ new statement, and calling its methods with local procedure calls won't suffice.

To create a COM object, some executing entity (an EXE or a Service) will have to perform remote memory allocation and object creation. This is a very complex task. By remote, we mean in another process or on another machine. This problem is solved by creating a concept called a *COM server*. This server will have to maintain tight communication with the client.

COM Methods Can Be Called Across a Network

If you have access to a machine on the network, and if a COM server for the object you want to use has been installed on that machine, then you can create the COM object on that computer. Of course, you must the proper privileges, and everything has to be set-up correctly on both the server and client computer. But if everything is configured properly and a network connection exists, activating a COM server on one machine from another machine is easy.

Since your COM object will not necessarily be on the local machine, you need a good way to "point to" it, even though its memory is somewhere else. Technically, there is no way to do this. In practice, it can be simulated by introducing a whole new level of objects. One of the ways COM does this is with a concept called a proxy/stub. We'll discuss proxy/stubs in some detail later.

Another important issue is passing data between the COM client and its COM server. When data is passed between processes, threads, or over a network, it is called "marshaling". Again, the proxy/stub takes care of the marshaling for you. COM can also marshal data for certain types of interface using Type Libraries and the Automation marshaller. The Automation marshaller does not need to be specifically built for each COM server - it is a general tool.

COM Objects Must Be Unique Throughout the World

OM objects must be unique throughout the world. This may seem like an exaggeration at first, but consider the Internet to be a worldwide network. Even if you're working on a single computer, COM must handle the possibility. Uniqueness is the issue. In C++ all classes are handled unequivocally by the compiler. The compiler can see the class definition for every class used in a program and can match up all references to it to make sure they conform to the class exactly. The compiler can also guarantee that there is only one class of a given name. In COM there must be a good way to get a similarly unequivocal match. COM must guarantee that there will only be one object of a given name even though the total number of objects available on a worldwide network is huge. This problem is solved by creating a concept called a GUID.

COM is Language Indpendent

COM servers may be written with a different language and an entirely different operating system. COM objects have the capability of being remotely accessible. That means they may be in a different thread, process, or even on a different computer. The other computer may even be running under a different operating system. There needs to be a good way to transmit parameters over the network to objects on other machines. This problem is solved by creating a new way to carefully specify the interface between the client and server. There is also a new compiler called MIDL (Microsoft Interface Definition Language). This compiler makes it possible to generically specify the interface between the server and client. MIDL defines COM objects, interfaces, methods and parameters.

COM Vocabulary

One of the problems we're going to have is keeping track of two sets of terminology. You're probably already familiar with C++

and some Object Oriented terminology. This table provides a rough equivalency between COM and conventional terminology.

Concept	Conventional (C++/OOP)	COM
Client	A program that requests services from a server.	A program that calls COM methods on a COM object running on a COM server.
Server	A program that "serves" other programs.	A program that makes COM objects available to a COM client.
Interface	None.	A pointer to a group of functions that are called through COM.
Class	A data type. Defines a group of methods and data that are used together.	The definition of an object that implements one or more COM interfaces. Also, "coclass".
Object	An instance of a class.	The instance of a coclass.
Marshaling	None.	Moving data (parameters) between client and server.

Table 1.1 A comparison of conventional C++ terminology with COM terminology

You'll notice the concepts of Interface and marshaling don't translate well into the C++ model. The closest thing to an interface in C++ is the export definitions of a DLL. DLL's do many of the same things that COM does when dealing with a tightly coupled (in-process) COM server. Marshaling in C++ is almost entirely manual. If you're trying to copy data between processes and computers, you'll have to write the code using some sort of inter-process communication. You have several choices, including sockets, the clipboard, and mailslots. In COM marshaling is generally handled automatically by MIDL.

The Interface

Thus far, we've been using the word "interface" pretty loosely. My dictionary (1947 American College Dictionary) defines an interface as follows:

"Interface, n. a surface regarded as the common boundary of two bodies or surfaces"

That's actually a useful general description. In COM "interface" has a very specific meaning and COM interfaces are a completely new concept, not available in C++. The concept of an interface is initially hard to understand for many people because an interface is a ghostlike entity that never has a concrete existence. It's sort of like an abstract class but not exactly.

At its simplest, an interface is nothing but a named collection of functions. In C++, a class (using this terminology) is allowed only one interface. The member functions of that interface are all the public member functions of the class. In other words, the interface is the publicly visible part of the class. In C++ there is almost no distinction between an interface and a class. Here's an example C++ class:

```
class yyy {
public:
  int DoThis();
private:
  void Helper1();
  int count;
  int x,y,z;
};
```

When someone tries to use this class, they only have access to the public members. (For the moment we're ignoring protected members and inheritance.) They can't call Helper1, or use any of the private variables. To the consumer of this class, the definition looks like this:

```
class yyy {
  int DoThis();
};
```

This public subset of the class is the 'interface' to the outside world. Essentially the interface hides the guts of the class from the consumer.

This C++ analogy only goes so far. A COM interface is not a C++ class. COM interfaces and classes have their own special set of rules and conventions.

COM allows a coclass (COM class) to have multiple interfaces, each interface having its own name and its own collection of functions. The reason for this feature is to allow for more complex and functional objects. This is another concept that is alien to C++. (Perhaps multiple interfaces could be envisioned as a union of two class definitions - something that isn't allowed in C++.)

Interfaces Isolate the Client From the Server

One of the cardinal rules of COM is that you can only access a COM object through an interface. The client program is completely isolated from the server's implementation through interfaces. This is an extremely important point. Let's look at a common everyday example to try to understand the point.

When you get into a car, you are faced with a variety of user interface. There is one interface that allows you to drive the car. Another allows you to work the headlights. Another controls the radio. And so on...

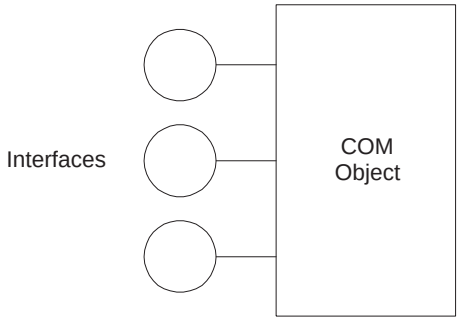

Figure 1–1 COM objects expose their functionality in one or more interfaces. An interface is a collection of functions.

There are many kinds of cars, but not all of them have radios. Therefore, they do not all implement the radio interface, although they do support the driving interface. In all cars that do have radios the capabilities of the radio are the same. A person driving a car without a radio can still drive, but cannot hear music. In a car that does have a radio, the radio interface is available.

Driving	Radio
Left()	On()
Right()	Off()
Slower()	Louder()
Faster()	Softer()
Forward()	NextStation()
Reverse()	PrevStation()

Table 1.2 Typical interfaces that a driver finds inside a car. If the car does not have a radio, then the radio interface is not available but the driver can still drive.

COM supports this same sort of model for COM classes. A COM object can support a collection of interfaces, each of which has a name. For COM objects that you create yourself, you will often define and use just a single COM interface. But many existing COM objects support multiple COM interfaces depending on the features they support.

Another important distinction is that the driving interface is not the car. The driving interface doesn't tell you anything about the brakes, or the wheels, or the engine of the car. You don't drive the engine for example, you use the faster and slower methods (accelerator and brakes) of the driving interface. You don't really care how the slower (brake) method is implemented, as long as the car slows down. Whether the car has hydraulic or air brakes isn't important. The interface isolates you from the implemenation details.

Imagine a Component

When you're building a COM object, you are very concerned about how the interface works. The user of the interface, however, shouldn't be concerned about its implementation. Like the brakes on a car, the user cares only that the interface works, not about the details behind the interface.

This isolation of interface and implementation is crucial for COM. By isolating the interface from its implementation, we can build components. Components can be replaced and re-used. This both simplifies and multiplies the usefulness of the object.

What's in a Name?

One important fact to recognize is that a named COM interface is unique. That is, a programmer is allowed to make an assumption in COM that if he accesses an interface of a specific name, the member functions and parameters of that interface will be exactly the same in all COM objects that implement the interface. So, following our example, the interfaces named "driving" and "radio" will have exactly the same member function signature in any COM object that implements them. If you want to change the member functions of an interface in any way, you have to create a new interface with a new name.

The Source of All Interfaces - IUnknown

Traditional explanations of COM start out with a thorough description of the IUnknown interface. IUnknown is the fundamental basis for all COM interfaces. Despite its importance, you don't need to know about IUnknown to understand the interface concept. The implementation of IUnknown is hidden by the higher level abstractions we'll be using to build our COM objects. Actually, paying too much attention to IUnknown can be confusing. Let's deal with it at a high level here so you understand the concepts.

IUnknown is like an abstract base class in C++. All COM interfaces must inherit from IUnknown. IUnknown handles the creation and management of the interface. The methods of IUnknown are used to create, reference count, and release a COM

object. All COM interfaces implement these 3 methods and they are used internally by COM to manage interfaces.

A Typical COM Object

Now let's put all of these new concepts together and describe a typical COM object and a program that wants to access it. In the next section and the following chapters we will make this real by implementing the actual code for the object.

Imagine that you want to create the simplest possible COM object. This object will support a single interface, and that interface will contain a single function. The purpose of the function is also extremely simple - it beeps. When a programmer creates this COM object and calls the member function in the single interface the object supports, the machine on which the COM object exists will beep. Let's further imagine that you want to run this COM object on one machine, but call it from another over the network.

Here are the things you need to do to create this simple COM object:

- You need to create the COM object and give it a name. This object will be implemented inside a COM server that is aware of this object.
- You need to define the interface and give it a name.
- You need to define the function in the interface and give it a name.
- You'll need to install the COM server.

For this example, let's call the COM object Beeper, the interface IBeep and the function Beep. One problem you immediately run into in naming these objects is the fact that all machines in the COM universe are allowed to support multiple COM servers, each containing one or more COM objects, with each COM object implementing one or more interfaces. These servers are created by a variety of programmers, and there is nothing to stop the programmers from choosing identical names. In the same way, COM objects are exposing one or more named interfaces,

again created by multiple programmers who could randomly choose identical names. Something must be done to prevent name collision, or things could get very confusing. The concept of a GUID, or a Globally Unique IDentifier, solves the "how do we keep all of these names unique" problem.

How to Be Unique - the GUID

There are really only two definitive ways to ensure that a name is unique:

1. You register the names with some quasi-governmental organization.
2. You use a special algorithm that generates unique numbers that are guaranteed to be unique world-wide (no small task).

The first approach is how domain names are managed on the network. This approach has the problem that you must pay $50 to register a new name and it takes several days for registration to take effect.

The second approach is far cleaner for developers. If you can invent an algorithm that is guaranteed to create a unique name each time anyone on the planet calls it, the problem is solved. Actually, this problem was originally addressed by the Open Software Foundation (OSF). OSF came up with an algorithm that combines a network address, the time (in 100 nanosecond increments), and a counter. The result is a 128-bit number that is unique.

The number 2 raised to the 128 power is an extremely large number. You could identify each nanosecond since the beginning of the universe - and still have 39 bits left over. OSF called this the UUID, for Universally Unique Identifier. Microsoft uses this same algorithm for the COM naming standard. In COM Microsoft decided to re-christen it as a Globally Unique Identifier: GUID.

The convention for writing GUID's is in hexadecimal. Case isn't important. A typical GUID looks like this:

```
"50709330-F93A-11D0-BCE4-204C4F4F5020"
```

Since there is no standard 128-bit data type in C++, we use a structure. Although the GUID structure consists of four different fields, you'll probably never need to manipulate its members. The structure is always used in its entirety.

```
typedef struct _GUID
{
  unsigned long Data1;
  unsigned short Data2;
  unsigned short Data3;
  unsigned char Data4[8];
} GUID;
```

The common pronunciation of GUID is "gwid", so it sounds like ‘squid’. Some people prefer the more awkward pronunciation of "goo-wid" (sounds like ‘druid’).

GUIDs are generated by a program called GUIDGEN. In GUIDGEN you push a button to generate a new GUID. You are guaranteed that each GUID you generate will be unique, no matter how many GUIDs you generate, and how many people on the planet generate them. This can work because of the following assumption: all machines on the Internet have, by definition, a unique IP address. Therefore, your machine must be on the network in order for GUIDGEN to work to its full potential. Actually, if you don't have a network address GUIDGEN will fake one, but you reduce the probability of uniqueness.

Both COM objects and COM interfaces have GUIDs to identify them. So the name "Beeper" that we choose for our object would actually be irrelevant. The object is named by its GUID. We call the object's GUID the class ID for the object. We could then use a #define or a const to relate the name “Beeper” to the GUID so that we don't have 128-bit values floating throughout the code. In the same way the interface would have a GUID. Note that many different COM objects created by many different programmers might support the same IBeep interface, and they would all use the same GUID to name it. If it is not the same

GUID, then as far as COM is concerned it is a different interface. The GUID is the name.

A COM Server

The COM server is the program that implements COM interfaces and classes. COM Servers come in three basic configurations.

- In-process, or DLL servers
- Stand-alone EXE servers
- Windows NT based services.

COM objects are the same regardless of the type of server. The COM interfaces and coclasses don't care what type of server is being used. To the client program, the type of server is almost entirely transparent. Writing the actual server however, can be significantly different for each configuration:

- In-Process servers are implemented as Dynamic Link Libraries (DLL's). This means that the server is dynamically loaded into your process at run-time. The COM server becomes part of your application, and COM operations are performed within application threads. Traditionally this is how many COM objects have been implemented because performance is fantastic - there is minimal overhead for a COM function call but you get all of the design and reuse advantages of COM. COM automatically handles the loading and unloading of the DLL.
- An out-of-process server has a more clear-cut distinction between the client and server. This type of server runs as a separate executable (EXE) program, and therefore in a private process space. The starting and stopping of the EXE server is handled by the Windows Service Control Manager (SCM). Calls to COM interfaces are handled through inter-process communication mechanisms. The server can be running on the local computer or on a remote computer. If the COM server is on a remote computer, we refer to it as "Distributed COM", or DCOM.

- Windows NT offers the concept of a service. A service is a program that is automatically managed by Windows NT, and is not associated with the desktop user. This means services can start automatically at boot time and can run even if nobody is logged on to Windows NT. Services offer an excellent way to run COM server applications.
- There is a fourth type of server, called a "surrogate". This is essentially a program that allows an in-process server to run remotely. Surrogates are useful when making a DLL-based COM server available over the network.

Interactions Between Client and Server

In COM, the client program drives everything. Servers are entirely passive, only responding to requests. This means COM servers behave in a synchronous manner toward individual method calls from the client.

- The client program starts the server.
- The client requests COM objects and interfaces.
- The client originates all method calls to the server.
- The client releases server interfaces, allowing the server to shut down.

This distinction is important. There are ways to simulate calls going from server to client, but they are odd to implement and fairly complex (They are called callbacks and are discussed later). In general, the server does nothing without a client request.

Table 1.3 is a typical interaction between a COM client and server. In COM you must take a client-centric approach.

Client Request	Server Response
Requests access to a specific COM interface, specifying the COM class and interface (by GUID)	• Starts the server (if required). If it is an In-Process server, the DLL will be loaded. Executable servers will be run by the SCM. • Creates the requested COM object. • Creates an interface to the COM object. • Increments the reference count of active interfaces. • Returns the interface to the client.
Calls a method of the interface.	Executes the method on a COM object.
Release the interface	• Decrements the interface's reference count. • If the reference count is zero, it may delete the COM object. • If there are no more active connections, shut down the server. Some servers do not shut themselves down.

Table 1.3 Interactions between a COM client and Server.

Summary

We've tried to look at COM from several points of view. C++ is the native language of COM, but it's important to see beyond the similarities. COM has many analogues in C++, but it has important differences. COM offers a whole new way of communicating between clients and servers.

The interface is one of the most important COM concepts. All COM interactions go through interfaces, and they shape that interaction. Because interfaces don't have a direct C++ counterpart, they are sometimes difficult for people to grasp. We've also introduced the concept of the GUID. GUIDs are ubiquitous in COM, and offer a great way to identify entities on a large network.

COM servers are merely the vehicles for delivering COM components. Everything is focused on the delivery of COM com-

ponents to a client application. In the following chapters, we'll create a simple client and server application to demonstrate these concepts.

T W O

Understanding the Simplest COM Client

The most straightforward way to begin understanding COM is to look at it from the perspective of a client application. Ultimately, the goal of COM programming is to make useful objects available to client applications. Once you understand the client, then understanding servers becomes significantly easier. Keeping clients and servers straight can be confusing, and COM tends to make the picture more complex when you are first learning the details.

Therefore, let's start with the simplest definition: A COM client is a program that uses COM to call methods on a COM server. A straightforward example of this client/server relationship would be a User Interface application (the client) that calls methods on another application (the server). If the User Interface application calls those methods using COM, then the user interface application is, by definition, a COM client.

We are belaboring this point for good reason - the distinction between COM servers and clients can get (and often is) much more complex. Many times, the application client will be a COM server, and the application's server will be a COM client. It's quite common for an application to be both a COM client and server. In this chapter, we will keep the distinction as simple as possible and deal with a pure COM client.

Four Steps to Client Connectivity

A client programmer takes four basic steps when using COM to communicate with a server. Of course, real-life clients do many more things, but when you peel back the complexity, you'll always find these four steps at the core. In this section we will present COM at its lowest level - using simple C++ calls.

Here is a summary of the steps we are going to perform:

1. Initialize the COM subsystem and close it when finished.
2. Query COM for a specific interfaces on a server.
3. Execute methods on the interface.
4. Release the interface.

For the sake of this example, we will assume an extremely simple COM server. We'll assume the server has already been written and save its description for the next chapter.

The server has one interface called IBeep. That interface has just one method, called Beep. Beep takes one parameter: a duration. The goal in this section is to write the simplest COM client possible to attach to the server and call the Beep method.

Following is the C++ code that implements these four steps. This is a real, working COM client application.

```
#include "..\BeepServer\BeepServer.h"

// GUIDS defined in the server
const IID IID_IBeepObj =
{0x89547ECD,0x36F1,0x11D2,{0x85,0xDA,0xD7,0x43,0xB2,0x
  32,0x69,0x28}};
const CLSID CLSID_BeepObj =
{0x89547ECE,0x36F1,0x11D2,{0x85,0xDA,0xD7,0x43,0xB2,0x
  32,0x69,0x28}};

int main(int argc, char* argv[])
{
  HRESULT hr;       // COM error code
  IBeepObj *IBeep; // pointer to interface

  hr = CoInitialize(0); // initialize COM
```

```
  if (SUCCEEDED(hr)) // macro to check for success
  {
    hr = CoCreateInstance(
      CLSID_BeepObj,     // COM class id
      NULL,                // outer unknown
      CLSCTX_INPROC_SERVER, // server INFO
      IID_IBeepObj,      // interface id
      (void**)&IBeep ); // pointer to interface

    if (SUCCEEDED(hr))
    {
      hr = IBeep->Beep(800); // call the method
      hr = IBeep->Release(); // release interface
    }
  }
  CoUninitialize(); // close COM
  return 0;
}
```

The header "BeepServer.h" is created when we compile the server. BeepServer is the in-process COM server we are going to write in the next chapter. Several header files are generated automatically by the compiler when compiling the server. This particular header file defines the interface IBeepObj. Compilation of the server code also generates the GUIDs seen at the top of this program. We've just pasted them in here from the server project.

Let's look at each of the 4 steps in detail.

Initializing the COM Subsystem:

This is the easy step. The COM method we need is CoInitialize().

```
CoInitialize(0);
```

This function takes one parameter and that parameter is always a zero - a legacy from its origins in OLE. The CoInitialize function initializes the COM library. You need to call this function before you do anything else. When we get into more sophisticated applications, we will be using the extended version, CoInitializeEx.

Call CoUninitialize() when you're completely finished with COM. This function de-allocates the COM library. I often include these calls in the InitInstance() and ExitInstance() functions of my MFC applications.

Most COM functions return an error code called an HRESULT. This error value contains several fields which define the severity, facility, and type of error. We use the SUCCEEDED macro because there are several different success codes that COM can return. It's not safe to just check for the normal success code (S_OK). We will discuss HRESULT's later in some detail.

Query COM for a Specific Interface

What a COM client is looking for are useful functions that it can call to accomplish its goals. In COM you access a set of useful functions through an interface. An interface, in its simplest form, is nothing but a collection of one or more related functions. When we “get” an interface from a COM server, we're really getting a pointer to a set of functions.

You can obtain an interface pointer by using the CoCreateInstance() function. This is an extremely powerful function that interacts with the COM subsystem to do the following:

- Locate the server.
- Start, load, or connect to the server.
- Create a COM object on the server.
- Return a pointer to an interface to the COM object.

There are two data types important to finding and accessing interfaces: CLSID and IID. Both of these types are Globally Unique ID's (GUID's). GUID's are used to uniquely identify all COM classes and interfaces.

In order to get a specific class and interface you need its GUID. There are many ways to get a GUID. Commonly we'll get the CLSID and IID from the header files in the server. In our example, we've defined the GUIDs with #define statements at the beginning of the source code simply to make them explicit and obvious. There are also facilities to look up GUIDs using the common name of the interface.

The function that gives us an interface pointer is CoCreateInstance.

```
hr = CoCreateInstance(
  CLSID_BeepObj,         // COM class id
  NULL,                  // outer unknown
  CLSCTX_INPROC_SERVER, // server INFO
  IID_IBeepObj,          // interface id
  (void**)&IBeep );      // pointer to interface
```

The first parameter is a GUID that uniquely specifies a COM class that the client wants to use. This GUID is the COM class identifier, or CLSID. Every COM class on the planet has its own unique CLSID. COM will use this ID to automatically locate a server that can create the requested COM object. Once the server is connected, it will create the object.

The second parameter is a pointer to what's called the “outer unknown”. We're not using this parameter, so we pass in a NULL. The outer unknown will be important when we explore the concept known as "aggregation". Aggregation allows one interface to directly call another COM interface without the client knowing it's happening. Aggregation and containment are two methods used by interfaces to call other interfaces.

The third parameter defines the COM Class Context, or CLSCTX. This parameter controls the scope of the server. Depending on the value here, we control whether the server will be an In-Process Server, an EXE, or on a remote computer. The CLSCTX is a bit-mask, so you can combine several values. We're using CLSCTX_INPROC_SERVER - the server will run on our local computer and connect to the client as a DLL. We've chosen an In-Process server in this example because it is the easiest to implement.

Normally the client wouldn't care about how the server was implemented. In this case it would use the value CLSCTX_SERVER, which will use either a local or in-process server, whichever is available.

Next is the interface identifier, or IID. This is another GUID - this time identifying the interface we're requesting. The IID we

request must be one supported by the COM class specified with the CLSID. Again, the value of the IID is usually provided either by a header file, or by looking it up using the interface name. In our code it is defined explicitly to make it obvious.

The last parameter is a pointer to an interface. CoCreateInstance() will create the requested class object and interface, and return a pointer to the interface. This parameter is the whole reason for the CoCreateInstance call. We can then use the interface pointer to call methods on the server.

Execute a Method on the Interface.

CoCreateInstance() uses COM to create a pointer to the IBeep interface. We can pretend the interface is a pointer to a normal C++ class, but in reality it isn't. Actually, the interface pointer points to a structure called a VTABLE, which is a table of function addresses. We can use the -> operator to access the interface pointer.

Because our example uses an In-Process server, it will load into our process as a DLL. Regardless of the details of the interface object, the whole purpose of getting this interface was to call a method on the server.

```
hr = IBeep->Beep(800);
```

Beep() executes on the server - it will cause the computer to beep. If we had a remote server, one which is running on another computer, that computer would beep.

Methods of an interface usually have parameters. These parameters must be of one of the types allowed by COM. There are many rules that control the parameters allowed for an interface. We will discuss these in detail in the section on MIDL, which is COM's interface definition tool.

Release the Interface

It's an axiom of C++ programming that everything that gets allocated should be de-allocated. Because we didn't create the interface with new, we can't remove it with delete. All COM

interfaces have a method called Release() which disconnects the object and deletes it. Releasing an interface is important because it allows the server to clean up. If you create an interface with CoCreateInstance, you'll need to call Release().

Summary

In this chapter we've looked at the simplest COM client. COM is a client driven system. Everything is oriented to making component objects easily available to the client. You should be impressed at the simplicity of the client program. The four steps defined here allow you to use a huge number of components, in a wide range of applications.

Some of the steps, such as CoInitialize() and CoUninitialize() are elementary. Some of the other steps don't make a lot of sense at first glance. It is only important for you to understand, at a high level, all of the things that are going on in this code. The details will clarify themselves as we go through further examples.

Visual C++ Version 5 and 6 simplify the client program further by using “smart pointers” and the #import directive. We’ve presented this example in a low level C++ format to better illustrate the concepts. We'll discuss smart pointers and imports in chapter 15.

In the next chapter, we'll build a simple in-process server to manage the IBeep interface. We’ll get into the interesting details of interfaces and activation in later chapters. See also Chapter 4 for an expansion on this example.

T H R E E

Understanding a Simple COM Server

So far we've looked at how to use COM through a client application. To the client, the mechanics of COM programming are pretty simple. The client application asks the COM subsystem for a particular component, and it is magically delivered.

There's a lot of code required to make all this behind-the-scenes component management work. The actual implementation of the object requires a complex choreography of system components and standardized application modules. Even using MFC the task is complex. Most professional developers don't have the time to slog through this process. As soon as the COM standard was published, it was quickly clear that it wasn't practical for developers to write this code themselves.

When you look at the actual code required to implement COM, you realize that most of it is repetitive boilerplate. The traditional C++ approach to this type of complexity problem would be to create a COM class library. And in fact, the MFC OLE classes provide most of COMs features.

There are however, several reasons why MFC and OLE were not a good choice for COM components. With the introduction of ActiveX and Microsoft's Internet strategy, it was important for COM objects to be very compact and fast. ActiveX requires that COM objects be copied across the network fairly quickly. If

you've worked much with MFC you'll know it is anything but compact (especially when statically linked). It just isn't practical to transmit huge MFC objects across a network.

Perhaps the biggest problem with the MFC/OLE approach to COM components is the complexity. OLE programming is difficult, and most programmers never get very far with it. The huge number of books about OLE is a testament to the fact that it is hard to use.

Because of the pain associated with OLE development, Microsoft created a new tool called ATL (Active Template Library). For COM programming, ATL is definitely the most practical tool to use at the present time. In fact, using the ATL wizard makes writing COM servers quite easy if you don't have any interest in looking under the hood.

The examples here are built around ATL and the ATL Application Wizard. This chapter describes how to build an ATL-based server and gives a summary of the code that the wizard generates.

Where's the Code?

One of the things that takes some getting used to when writing ATL servers is that they don't look like traditional programs. A COM server written by ATL is really a collaboration between several disparate components:

- Your application
- The COM subsystem
- ATL template classes
- "IDL" code and MIDL Generated "C" headers and programs
- The system registry

It can be difficult to look at an ATL-based COM application and see it as a unified whole. Even when you know what it's doing, there are still big chunks of the application that you can't see. Most of the real server logic is hidden deep within the ATL header files. You won't find a single main() function that man-

ages and controls the server. What you will find is a thin shell that makes calls to standard ATL objects.

In the following section we're going to put together all the pieces required to get the server running. First we will create the server using the ATL COM AppWizard. The second step will be to add a COM object and a Method. We'll write an In-Process server because it's one of the simpler COM servers to implement. Our apartment-threaded in-process server also avoids having to build a proxy and stub object.

Building a DLL-Based (In-Process) COM Server

An In-Process server is a COM library that gets loaded into your program at run-time. In other words, it's a COM object in a Dynamic Link Library (DLL). A DLL isn't really a server in the traditional sense, because it loads directly into the client's address space. If you're familiar with DLLs, you already know a lot about how the COM object gets loaded and mapped into the calling program.

Normally a DLL is loaded when LoadLibrary() is called. In COM, you never explicitly call LoadLibrary(). Everything starts automatically when the client program calls CoCreateInstance(). One of the parameters to CoCreateInstance is the GUID of the COM class you want. When the server gets created at compile time, it registers all the COM objects it supports. When the client needs the object, COM locates the server DLL and automatically loads it. Once loaded, the DLL has a class factory to create the COM object.

CoCreateInstance() returns a pointer to the COM object, which is in turn used to call a method (in the example described here, the method is called Beep().) A nice feature of COM is that the DLL can be automatically unloaded when it's not needed. After the object is released and CoUninitialize() is called, FreeLibrary() will be called to unload the server DLL.

If you didn't follow all that, it's easier than it sounds. You don't have to know anything about DLL's to use COM. All you have to do is call CoCreateInstance(). One of the advatages of

COM is that it hides these details so you don't have to worry about this type of issue.

There are advantages and disadvantages to In-process COM servers. If dynamic linking is an important part of your system design, you'll find that COM offers an excellent way to manage DLL's. Some experienced programmers write all their DLL's as In-process COM servers. COM handles all the chores involved with the loading, unloading, and exporting DLL functions and COM function calls have very little additional overhead.

Our main reason for selecting an In-process server is somewhat more prosaic: It makes the example simpler. We won't have to worry about starting remote servers (EXE or service) because our server is automatically loaded when needed. We also avoid building a proxy/stub DLL to do the marshaling.

Unfortunately, because the In-Process server is so tightly bound to our client, a number of the important "distributed" aspects of COM are not going to be exposed. A DLL server shares memory with it's client, whereas a distributed server would be much more removed from the client. The process of passing data between a distributed client and server is called marshaling. Marshaling imposes important limitations on COM's capabilities that we won't have to worry about with an apartment-threaded in-proc server. We will expose and study these details in later chapters.

Creating the Server Using the ATL Wizard

We're going to create a very simple COM server in this example in order to eliminate clutter and help you to understand the fundamental principles behind COM very quickly. The server will only have one method - Beep(). All that this method will do is sound a single beep; not a very useful method. What we're really going to accomplish is to set up all the parts of a working server. Once the infrastructure is in place, adding methods to do something useful will be extremely straightforward.

The ATL AppWizard is an easy way to quickly generate a working COM server. The Wizard will allow us to select all the

basic options, and will generate most of the code we need. Below is the step-by step process for creating the server. In this process we will call the server BeepServer. All COM servers must have at least one interface, and our interface will be called IBeepObj. You can name your COM interfaces almost anything you want, but you should always prefix them with an 'I' if you want to follow standard naming conventions.

NOTE: If you find the difference between a COM "Object" , "Class", and "Interface" confusing at this point, you're not alone. The terminology can be uncomfortable initially, especially for C++ programmers. The feelings of confusion will subside as you work through examples. The word "coclass" for COM class is used in most Microsoft documentation to distinguish a COM class from a normal C++ class.

Here are the steps for creating a new COM server with the ATL Wizard using Visual C++ version 6 (it looks nearly identical in version 5 as well):

1. First, create a new "ATL COM AppWizard" project. Select File/New from the main menu.

2. Select the "Projects" tab of the "New" dialog. Choose "ATL COM AppWizard" from the list of project types. Select the following options and press OK.
   a. Project Name: BeepServer
   b. Create New Workspace
   c. Location: Your working directory.

3. At the first AppWizard dialog we'll create a DLL based (In-process) server. Enter the following settings :
   a. Dynamic Link Library
   b. Don't allow merging proxy/stub code
   c. Don't support MFC

4. Press Finish.

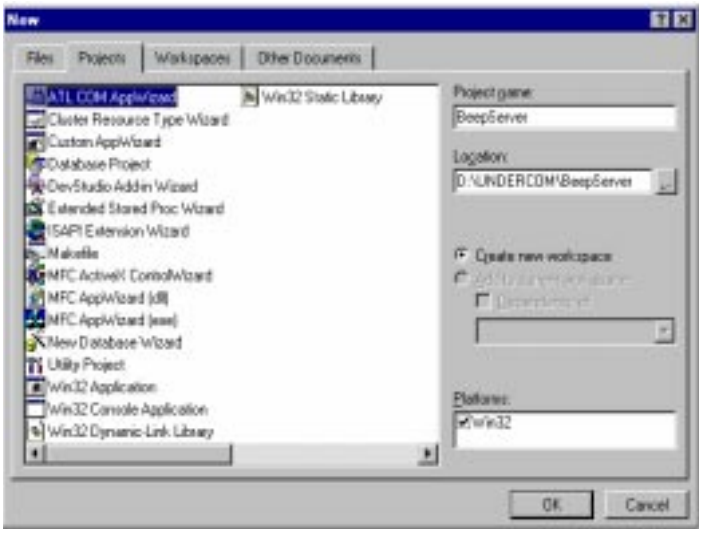

Figure 3–1 Accessing the ATL Wizard

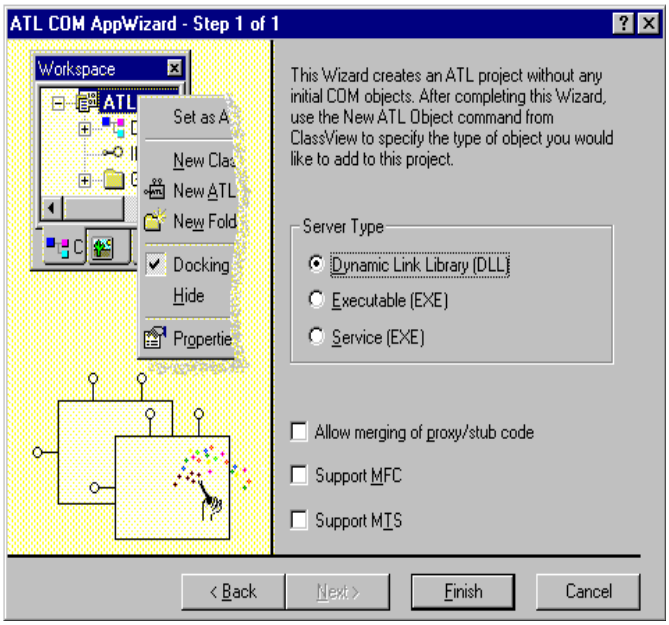

Figure 3–2 Creating a DLL server

The AppWizard creates a project with all the necessary files for a DLL-based COM server. Although this server will compile and run, it's just an empty shell. For it to be useful it will need a COM interface and the class to support the interface. We'll also have to write the methods in the interface.

Adding a COM Object

Now we'll proceed with the definition of the COM object, the interface, and the methods. This class is named BeepObj and has an interface called IBeepObj:

1. Look at the "Class View" tab. Initially it only has a single item in the list. Right click on "BeepServer Classes" item.

2. Select "New ATL ObjectÖ". This step can also be done from the main menu. Select the "New ATL Object" on the Insert menu item.

3. At the Object Wizard dialog select "Objects". Choose "Simple Object" and press Next.

4. Choose the Names tab. Enter short name for the object: BeepObj. All the other selections are filled in automatically with standard names.

5. Press the "Attributes" tab and select: Apartment Threading, Custom Interface, No Aggregation.

6. Press OK. This will create the COM Object.

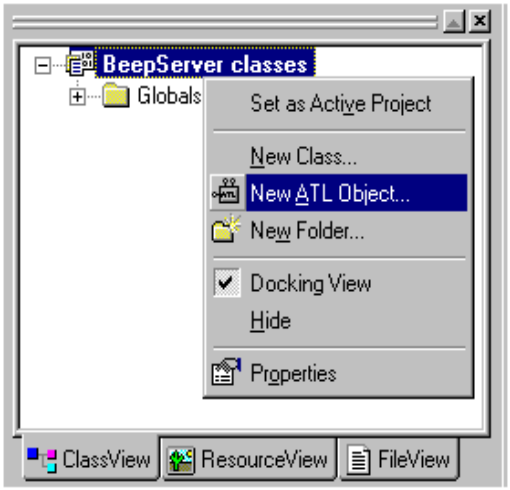

Figure 3–3 Adding a new object to the server

Figure 3–4 Adding a new object

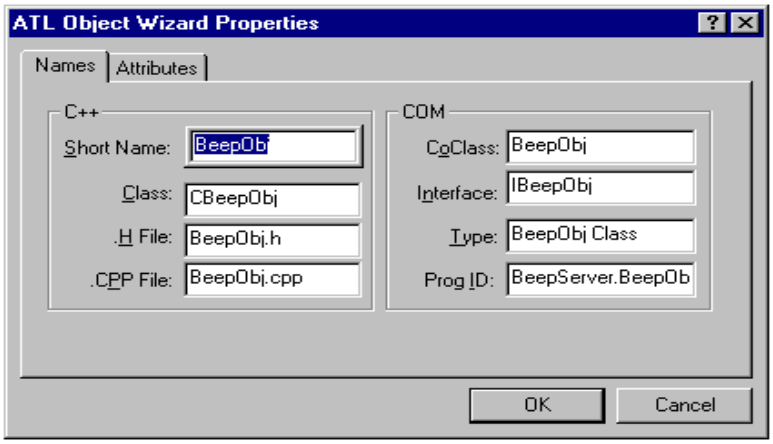

Figure 3–5 Specifying the object naming

Figure 3–6 Specifying the threading model and other parameters

Adding a Method to the Server

We have now created an empty COM object. As of yet, it's still a useless object because it doesn't do anything. We will create a simple method called Beep() which causes the system to beep once. Our COM method will call the Win32 API function ::Beep(), which does pretty much what you would expect.

1. Go to "Class View" tab. Select the IBeepObj interface. This interface is represented by a small icon that resembles a spoon.

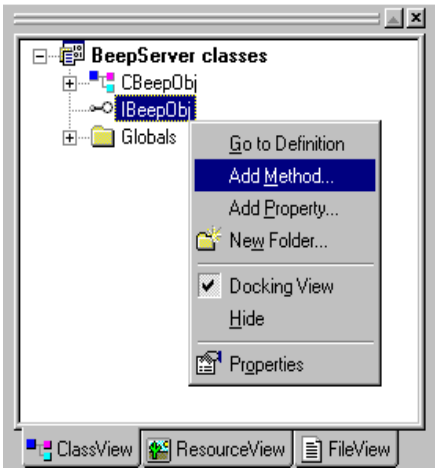

Figure 3–7 Adding a methos

2. Right click the IBeepObj interface. Select "Add Method" from the menu.
3. At the "Add Method to Interface" dialog, enter the following and press OK. Add the method "Beep" and give it a single [in] parameter for the duration. This will be the length of the beep, in milliseconds.

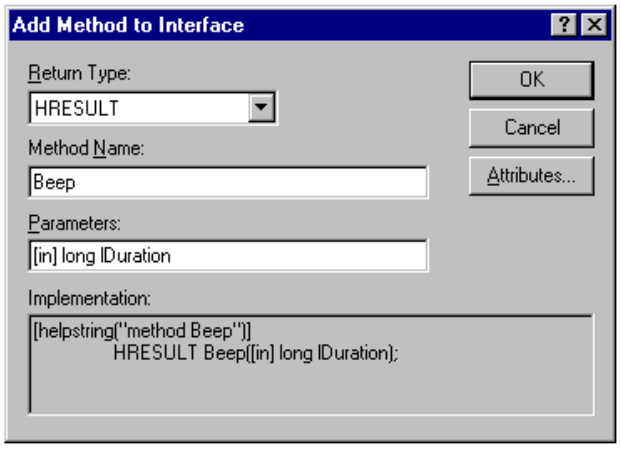

Figure 3–8 Specifying the method's name and parameters

4. "Add Method" has created the MIDL definition of the method we defined. This definition is written in IDL, and describes the method to the MIDL compiler. If you want to see the IDL code, double click the "IBeepObj" interface at the "Class View" tab. This will open and display the file BeepServer.IDL. No changes are necessary to this file, but here's what our interface definition should look like.

```
interface IBeepObj : IUnknown
{
[helpstring("method Beep")]
   HRESULT Beep([in] LONG duration);
};
```

The syntax of IDL is quite similar to C++. This line is the equivalent to a C++ function prototype. We will cover the syntax of IDL in Chapter 7.

5. Now we're going to write the C++ code for the method. The AppWizard has already written the empty shell of our C++

function, and has added it to the class definition in the header file (BeepServer.H).

Open the source file BeepObj.CPP. Find the //TODO: line and add the call to the API Beep function. Modify the Beep() method as follows:

```
STDMETHODIMP CBeepObj::Beep(LONG duration)
{
     // TODO: Add your implementation code here
     ::Beep( 550, duration );
     return S_OK;
}
```

6. Save the files and build the project.

We now have a complete COM server. When the project finishes building, you should see the following messages:

```
----Configuration: BeepServer - Win32 Debug----
Creating Type Library...

Microsoft (R) MIDL Compiler Version 5.01.0158
Copyright (c) Microsoft Corp 1991-1997. All rights
  reserved.
Processing D:\UnderCOM\BeepServer\BeepServer.idl
BeepServer.idl
Processing C:\Program Files\Microsoft Visual Stu-
  dio\VC98\INCLUDE\oaidl.idloaidl.idl
.
.
Compiling resources...
Compiling...
StdAfx.cppCompiling...
BeepServer.cpp
BeepObj.cpp
Generating Code...
Linking...
  Creating library  Debug/BeepServer.lib and object
  Debug/BeepServer.exp
Performing registration
```

```
BeepServer.dll - 0 error(s), 0 warning(s)
```

This means that the Developer Studio has completed the following steps:

- Executed the MIDL compiler to generate code and type libraries
- Compiled the source files
- Linked the project to create BeepServer.DLL
- Registered COM components
- Registered the DLL with RegSvr32 so it will automatically load when needed.

Let's look at the project that we've created. While we've been clicking buttons, the ATL AppWizard has been generating files. If you look at the "FileView" tab, the following files have been created:

Source File	Description
BeepServer.dsw	Project workspace
BeepServer.dsp	Project File
BeepServer.plg	Project log file. Contains detailed error information about project build.
BeepServer.cpp	DLL Main routines. Implementation of DLL Exports
BeepServer.h	MIDL generated file containing the definitions for the interfaces
BeepServer.def	Declares the standard DLL module parameters: DllCanUnloadNow, DllGetClassObject, DllUnregisterServer
BeepServer.idl	IDL source for BeepServer.dll. The IDL files define all the COM components.
BeepServer.rc	Resource file. The main resource here is IDR_BEEPDLLOBJ which defines the registry scripts used to load COM information into the registry.
Resource.h	Microsoft Developer Studio generated include file.
StdAfx.cpp	Source for precompiled header.
Stdafx.h	Standard header

BeepServer.tlb	Type Library generated by MIDL. This file is a binary description of COM interfaces and objects. The TypeLib is very useful as an alternative method of connecting a client.
BeepObj.cpp	Implementation of CBeepObj. This file contains all the actual C++ code for the methods in the COM BeepObj object.
BeepObj.h	Definition of BeepObj COM object.
BeepObj.rgs	Registry script used to register COM components in registry. Registration is automatic when the server project is built.
BeepServer_i.c	Contains the actual definitions of the IID's and CLSID's. This file is often included in cpp code.
	There are several other proxy/stub files that are generated by MIDL.

Table 3.1 All the files created by the ATL wizard

In just a few minutes, we have created a complete COM server application. Back in the days before wizards, writing a server would have taken hours. Of course the down-side of wizards is that we now have a large block of code that we don't fully understand. In Chapter 5 we will look at the generated modules in detail, and then as a whole working application.

Running the Client and the Server

Now that we have compiled the server and we have a working client (from the previous chapter), we can run the two of them together. In theory, all that you have to do is run the client. Because the server DLL was automatically registered in the registry as part of the build process, the client will automatically find and load the server and then call its Beep method. You will hear the appropriate “beep” sound. If there is a problem you will get no textual complaint from the client (as it contains no error checking code - see the next chapter to correct that problem...) but it will not beep.

If you had trouble building the client or the server (that is, if any errors or warnings were generated during the build or link process), one thing to check is to make sure that both the client and server are being built as normal Win32 Debug configurations. Sometimes the system will default to odd Unicode release builds. In the Build menu you can check and change the active configuration to “Win32 Debug”.

If both client and server build fine but the client does not beep, that means that either the client could not find or could not start the server. Assuming that you built the server as described above and there were no errors, we know it exists. The problem almost certainly is occuring because the GUIDs do not match between the client and the server. Recall that we used statically declared GUIDS in the client in Chapter 2 to make the GUIDs more obvious. That works fine if you are pulling the code off the CD, but will be a problem if you generated the server with the ATL wizard yourself. To solve this problem, look for the “_i.c” file that MIDL generated in the server directory. In that file is an IID GUID and a CLSID GUID. Copy them into the appropriate spot in the client application, rebuild and try again. You should hear the appropriate beep when the client executes. Now that you can see where the GUIDs are coming from, you may want to modify the client so it #includes the “_i.c” file and use the GUIDs directly from there.

Summary

The server code was almost entirely generated by the ATL wizards. It provides a working implementation of the server. We examined a DLL based server, but the process is almost identical for all server types. This framework is an excellent way to quickly develop a server application because you don't have to know the myriad of details required to make it work.

F O U R

Creating your own COM Clients and Servers

Based on the previous three chapters, you can see that it is extremely easy to create COM clients and servers. In fact, you were probably stunned by how little code was actually required. Just a handful of lines on both the client and server sides yields a complete COM application. You can now see why many developers use COM whenever they want to create a DLL - it only takes about 2 minutes to set up an in-proc COM DLL with the ATL wizard and get it working.

The purpose of this chapter is to review the steps you need to take to create your own COM servers and use them in real applications you create. As you will recall, the client code previously presented was a bit sparse. We will expand on it a bit, look at the code you need to embed in any client to activate the server properly, and then look at an MFC application that lets you try out some of the error modes that a COM client may typically encounter.

Server Side

As we saw in Chapter 3, the ATL Wizard makes COM server creation extremely easy. The first step to creating any COM server,

however, relies solely on you. You need to select one or more pieces of functionality that you want to separate from the main body of an application's code. You often want to separate the functionality in order to make it reusable across multiple applications. But you may also want to do it because it allows a team of programmers to divide easily into separate working groups, or because it makes code development or maintenance easier. Whatever the reason, defining the functionality for the COM server is the first step.

One thing that makes defining the boundary easy is the fact that, in the simplest case, a COM server can act almost identically to a normal C++ class. Like a class, you instantiate a COM class and then start calling its methods. The syntax of COM instantiation and method calling is slightly different from the syntax in C++, but the ideas are identical. If a COM server has only one interface, then it is, for all practical purposes, a class. You still have to obey the rules of COM when accessing the object, but the concepts are the same.

Once you have decided on the functionality and the methods that will be used to access that functionality, you are ready to build your server. As we in Chapter 3, there are 4 basic steps you must take to create a server:

1. Use the ATL Wizard to create the shell for your COM server. You choose whether you want the server to be a DLL, an EXE or a server.
2. Create a new COM object inside the server shell with the ATL object wizard. You will choose the threading model. This creates the interface into which you can install your methods.
3. Add the methods to your object and declare their parameters.
4. Write the code for your methods.

Each of these tasks has been described in detail in the previous chapter. Once you have completed these steps you are ready to compile your COM object and use it.

After reading the previous chapter, one question frequently asked concerns threading models. Specifically, what is the differ-

ence between apartment-threaded and free-threaded COM objects? Chapter 10 contains a complete description, but the easiest way to understand the difference is to think of apartment-threaded COM objects as single-threaded, while free-threaded COM objects as multi-threaded.

In apartment threading, method calls from multiple clients are serialized in the COM object on the server. That is, each individual method call completes its execution before the next method call can begin. Apartment-threaded COM objects are therefore inherently thread safe. Free threaded COM objects can have multiple method calls executing in the COM object at the same time. Each method call from each client runs on a different thread. In a free-threaded COM object you therefore have to pay attention to multi-threading issues such as synchronization.

Initially you will want to use apartment threading because it makes your life easier, but over time the move to free threading can sometimes make things more flexible, responsive and efficient.

Client Side

The client presented in chapter 2 has the benefits of clarity and compactness. However, it contains no error-checking code and that makes it insufficient in a real application. Let's review that code, however, because it is so simple and it shows the exact steps that you must take to create a successful client:

```
void main()
{
        HRESULT hr;             // COM error code
        IBeepDllObj *IBeep;     // pointer to interface

        hr = CoInitialize(0);   // initialize COM
        if (SUCCEEDED(hr))      // check for success
        {
            hr = CoCreateInstance(
               clsid,                 // COM class id
               NULL,                  // outer unknown
```

```
                CLSCTX_INPROC_SERVER,   // server INFO
                iid,                    // interface id
                (void**)&IBeep );       // interface

            if (SUCCEEDED(hr))
            {
                // call the method
                hr = IBeep->Beep(800);
                // release the interface when done
                // calling its methods
                hr = IBeep->Release();
            }
        CoUninitialize();               // close COM
}
```

The call to CoInitialize and CoCreateInstance initializes COM and gets a pointer to the necessary interface. Then you can call methods on the interface. When you are done calling methods you release the interface and call CoUninitialize to finish with COM. That's all there is to it.

That would be all there is to it, that is, if things always worked as planned. There are a number of things that can go wrong when a COM client tries to start a COM server. Some of the more common include:

- The client could not start COM
- The client could not locate the requested server
- The client could locate the requested server but it did not start properly
- The client could not find the requested interface
- The client could not find the requested function
- The client could find the requested function but it failed when called
- The client could not clean up properly

In order to track these potential problems, you have to check things every step of the way by looking at hr values. The above code does the checking, but it is difficult to tell what has gone wrong because the code is completely silent if an error occurs. The following function remedies that situation:

```
// This function displays detailed information con-
  tained in an HRESULT.
BOOL ShowStatus(HRESULT hr)
{
  // construct a _com_error using the HRESULT
  _com_error e(hr);

  // Show the hr as a decimal number
  cout << "hr as decimal: " << hr << endl;
  // Show the 1st 16 bits (SCODE)
  cout << "SCODE: " << HRESULT_CODE( hr ) << endl;
  // Show facility code as a decimal number
  cout << "Facility: " << HRESULT_FACILITY( hr ) <<
    endl;
  // Show the severity bit
  cout << "Severity: " << HRESULT_SEVERITY( hr ) <<
    endl;
  // Use the _com_error object to
  // format a message string. This is
  // much easier than using ::FormatMessage
  cout << "Message string: " << e.ErrorMessage() <<
    endl;
  return TRUE;
}
```

This function dismantles an HRESULT and prints all of its components, including the extremely useful ErrorMessage value. You can call the function at any time with this function call:

```
// display HRESULT on screen
ShowStatus( hr );
```

See Chapter 16 for details on HRESULTS. See the error appendix for details on overcoming COM and DCOM errors.

To fully explore the different error modes of a simple COM program, the CD contains an MFC client and a sample server. The client is a simple MFC dialog application designed to let you simulate several possible errors and see the effect they have on the HRESULT. When the client runs it will look like this:

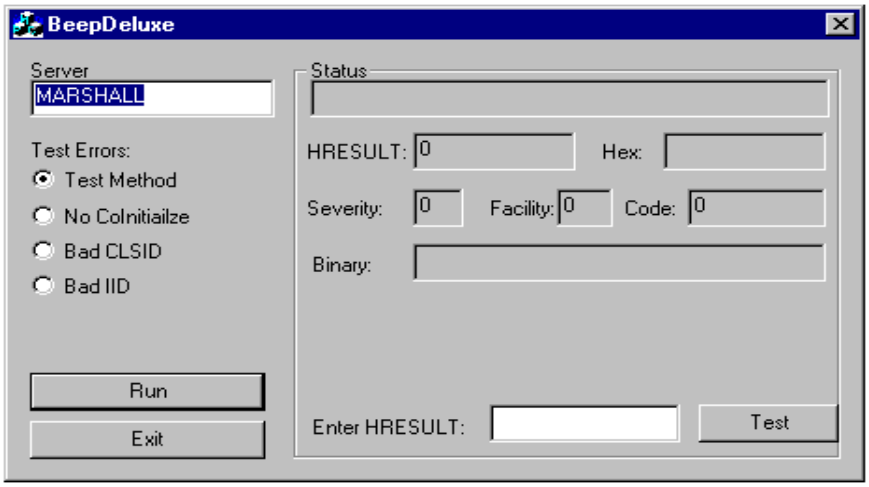

Figure 4–1 Dialog from the test client, a simple MFC application that allows you to simulate different COM errors and see the effects.

You can see that the radio buttons on the left hand side let you experiment with a lack of a CoInitialize function, a bad class ID and a bad interface ID. If you click the Run button, the area on the right will show the effect of the different errors on the HRESULT returned by different functions in the client.

[Note - Some readers initially have trouble compiling or linking this code. For some reason VC++ v6 will sometimes default to a very odd Unicode Release build instead of the expected Win32 Debug build. Use the Active Configuration... option in the Build menu to check the configuration and to set it to Win32 Debug if it is incorrect.]

When you explore the client code in this example, you will find that it is a somewhat more robust version of the standard client code we used above. For example, it sets default security using the CoInitializeSecurity function to introduce you to that function (see Chapter 14 for details), and it also makes use of the

CoCreateInstanceEx function so that remote servers on other machines can be called (see chapter 14 for details).

Let's look at the basic plan of the client. It starts with code generated by the MFC App Wizard, with the request that the App Wizard generate a simple dialog application. The resource file of this application was modified to match the dialog seen above. The bulk of the application is a single function, OnButtonRun, that is activated when the user clicks the Run button:

```
// This method displays detailed information contained
  in an HRESULT.
BOOL CBeepDeluxeDlg::ShowStatus(HRESULT hr)
{
  // construct a _com_error using the HRESULT
  _com_error e(hr);

  // The hr as a decimal number
  m_nHR = hr;
  // The hr as a hex number
  m_strHRX.Format( "%x", hr );
  m_strHRX = "0x" + m_strHRX;
  // show the 1st 16 bits (SCODE)
  m_nCode = HRESULT_CODE( hr );
  // Show facility code as a decimal number
  m_nFac = HRESULT_FACILITY( hr );
  // Show the severity bit
  m_nSev = HRESULT_SEVERITY( hr );
  // Use the _com_error object to
  // format a message string. This is
  // Much easier then using ::FormatMessage
  m_strStatus= e.ErrorMessage();
  // show bits
  m_strBinary = HrToBits( hr );
  return TRUE;
}

// This method converts an HRESULT into
// a string of 1's and 0's
CString CBeepDeluxeDlg::HrToBits( HRESULT hr)
{
```

```
  char temp[32 + 8  + 1 ]; // 32 bits + 8 spaces + NULL
  unsigned long mask = 0x80000000; // bit mask (msb)
  int count = 0;

  // ensure that there is a null terminator at the end
  memset( temp, 0, sizeof(temp));

  // loop all 32 bits
  for( int i=31; i>=0; i-- )
  {
    // set the character value to 0 or 1 if bit is set
    temp[count++] = (hr & mask) ? '1' : '0';

    // put 1 space every 4 characters
    if ((i%4) == 0)  // mod operator
    {
      temp[count++] = ' ';
    }

    // shift bitmask 1 right
    mask = mask >> 1;
  }

  return temp;
}

// Execute an extensive test of the COM client
void CBeepDeluxeDlg::OnButtonRun()
{
  CWaitCursor cur; // show hourglass
  UpdateData( TRUE ); // update variables

  // status value
  HRESULT hr = S_OK;
  // define a test clsid
  CLSID clsid = BAD_GUID;

  // remote server info
  COSERVERINFO cs;
  // Init structures to zero
  memset(&cs, 0, sizeof(cs));
```

```
// Allocate the server name in
// the COSERVERINFO structure
cs.pwszName = m_strServer.AllocSysString();

// structure for CoCreateInstanceEx
MULTI_QI qi[1];
memset(qi, 0, sizeof(qi));

// Initialize COM
if (m_nRadio != 1)
{
  hr = CoInitialize(0);
}

// set CLSID
if (m_nRadio != 2)
{
  clsid = CLSID_DispatchTypes;
}

// set IID
if (m_nRadio != 3)
{
  // Fill the qi with a valid interface
  qi[0].pIID = &IID_IDispatchTypes;
}
else
{
  // send it a bad Interface
  qi[0].pIID = &BAD_GUID;
}

// set a low level of security
hr = CoInitializeSecurity(NULL, -1, NULL, NULL,
  RPC_C_AUTHN_LEVEL_NONE,
  RPC_C_IMP_LEVEL_IMPERSONATE,
  NULL,
  EOAC_NONE,
  NULL);

if (SUCCEEDED(hr))
```

```
  {
    // get the interface pointer
    hr = CoCreateInstanceEx( clsid, NULL,
  CLSCTX_SERVER,
          &cs, 1, qi );

    // call method if the interface was created
    if (SUCCEEDED(hr))
    {
      // extract the interface from the QI structure
      IDispatchTypes *pI =
        (IDispatchTypes*)qi[0].pItf;

      // call method
      hr = pI->Beep(100);

      // The HRESULT will be displayed later if there
  was an error

      // release the pointer even if there was an
  error on the method
      pI->Release();
    }
  }

  // display HRESULT on screen
  ShowStatus( hr );

  // close COM
  CoUninitialize();

  // Update screen
  UpdateData( FALSE );
}
```

The main feature of this program is the top part of the OnButtonRun function, which selectively breaks different parts of the application in response to the radio button settings. Then the HRESULT value is dismantled and displayed.

When creating your own MFC clients, you will want to follow this same general plan. You may want to place the CoInitialize and CoUninitialize functions elsewhere in the application so they are not called constantly (for example, in InitInstance and ExitInstance). You may wish to do the same with the call to CoInitializeSecurity, CoCreateInstanceEx and Release depending on how many calls you are planning to make to an interface. If you are calling a large number of functions in an interface, clearly you will want to call CoCreateInstanceEx and Release only once.

F I V E

Understanding ATL-Generated Code

The source code for our server DLLs is being generated by ATL. For many people it is perfectly OK to never look at the code ATL created. For others, "not knowing" the details of this code is unacceptable. This chapter gives you a quick tour of the code produced by ATL. The code for the server DLL that is now sitting on your hard drive really resides in three different types of files.

- First, there are the traditional C++ source and header files. Initially, all of this code is generated by the ATL wizards.
- The Beep method was added by the "Add Method" dialog, which modified the MIDL interface definition. The MIDL source code is in an IDL file - in this example it's BeepServer.IDL. The MIDL compiler will use this file to create several output files. These files will take care of much of the grunt work of implementing the server. As we add methods to the COM object, we'll be adding definitions the IDL file.
- The third group of source files are automatically generated MIDL output files created by the MIDL compiler. These files are source code files, but because they are automatically generated by the MIDL compiler from IDL

source code, these files are never modified directly either by wizards or by developers. You might call them "second generation files" - the wizard created an IDL file and the MIDL compiler created source code files from that IDL file. The files created by the MIDL compiler include:

1. BeepServer.RGS - Registration script for the server.
2. BeepServer.h - This file contains definitions for the COM components.
3. BeepServer_i.c - GUID structures for the COM components.
4. Proxy/Stub files - This includes "C" source code, DLL definitions, and makefile (.mk) for the Proxy and Stub.

The ATL wizard also creates an application "resource". If you look in the project resources, you'll find it under "REGISTRY". This resource contains the registration script defined in BeepServer.RGS. The name of the resource is IDR_BEEPOBJ.

We look at all of these different components in the sections below. See also Chapter 15 for additional details on ATL.

The Main C++ Module

When we ran the ATL COM AppWizard, we chose to create a DLL-based server and we chose not to use MFC. The first selection screen of the wizard determined the overall configuration of the server.

The AppWizard created a standard DLL module. This type of standard DLL does not have a WinMain application loop, but it does have a DllMain function used for the initialization of the DLL when it gets loaded:

```
CComModule _Module;

BEGIN_OBJECT_MAP(ObjectMap)
  OBJECT_ENTRY(CLSID_BeepObj, CBeepObj)
END_OBJECT_MAP()
```

```
////////////////////////////////////////////////
// DLL Entry Point

extern "C"
BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwRea-
  son, LPVOID /*lpReserved*/)
{
  if (dwReason == DLL_PROCESS_ATTACH)
  {
    _Module.Init(ObjectMap, hInstance);
    DisableThreadLibraryCalls(hInstance);
  }
  else if (dwReason == DLL_PROCESS_DETACH)
    _Module.Term();
  return TRUE;    // ok
}
```

Really all the DllMain function does is check if a client is attaching to the DLL and then does some initialization. At first glance, there's no obvious indication that this is a COM application at all.

The COM portion of our new server is encapsulated in the ATL class CComModule. CComModule is the ATL server base class. It contains all the COM logic for registering and running servers, as well as starting and maintaining COM objects. CComModule is defined in the header file "atlbase.h". This code declares a global CComModule object in the following line:

```
CComModule _Module;
```

This single object contains much of the COM server functionality for our application. Its creation and initialization at the start of program execution sets a chain of events in motion.

ATL requires that your server always name its global CComModule object "_Module". It's possible to override CComModule with your own class, but you aren't allowed to change the name.

If we had chosen an executable-based server, or even a DLL with MFC, this code would be significantly different. There would still be a CComModule-based global object, but the entry

point of the program would have been WinMain(). Choosing a MFC-based DLL would have created a CWinApp-based main object.

Object Maps

The CComModule is connected to our COM object (CBeepObj) by the object map seen in the previous section. An object map defines an array of all the COM objects the server controls. The object map is defined in code using the OBJECT_MAP macros. Here is our DLL's object map:

```
BEGIN_OBJECT_MAP(ObjectMap)
  OBJECT_ENTRY(CLSID_BeepObj, CBeepObj)
END_OBJECT_MAP()
```

The OBJECT_ENTRY macro associates the CLSID of the object with a C++ class. It's common for a server to contain more than one COM object - in that case there will be an OBJECT_ENTRY for each one.

Export File

Our In-Process DLL, like most DLLs, has an export file. The export file will be used by the client to connect to the exported functions in our DLL. These definitions are in the file BeepServer.def:

```
; BeepServer.def : Declares the module parameters.

LIBRARY      "BeepServer.DLL"

EXPORTS
  DllCanUnloadNow     @1 PRIVATE
  DllGetClassObject   @2 PRIVATE
  DllRegisterServer   @3 PRIVATE
  DllUnregisterServer@4 PRIVATE
```

It is important to note what is not exported: there are no custom methods here. There is no export for the "Beep" method. These are the only exports you should see in a COM DLL.

Looking into the BeepServer.CPP file, we see that the implementation of these four functions is handled by the COM application class. Here's the code for DllRegisterServer:

```
// DllRegisterServer - Adds entries to the system reg-
  istry
STDAPI DllRegisterServer(void)
{
  // registers object, typelib and all interfaces in
  typelib
  return _Module.RegisterServer(TRUE);
}
```

In this case, the DLL just calls ATL's CComModule::RegisterServer() method. CComModule implements the server registration in a way that is compatible with In-Process, Local, and Remote COM servers. The other three exported DLL functions are equally spartan. The actual implementation is hidden in the ATL templates.

Most of the code described above is DLL specific code. You will only get this configuration if you choose to create a DLL-based server. None of the code in the main module is COM specific. The main module is entirely devoted to the infrastructure required to deliver COM objects in a DLL, and this code will be significantly different depending on the type of server. The actual code inside the server is much more uniform. The implementation of a coclass and interface is identical regardless of the type of server (DLL, EXE, server) you create. You should be able to take a coclass from a DLL server and implement it in an EXE-based server with few changes.

The COM Object - "CBeepObj"

A COM server has to implement at least one COM object. We are using a single object named "CBeepObj". One of the most interesting things about this object is that the code was entirely generated by ATL wizards. It is quite remarkable how compact this object definition turns out to be. The class definition is found in BeepObj.h:

```
// BeepObj.h : Declaration of the CBeepObj
#include "resource.h"       // main symbols
/////////////////////////////////////////////////////
// CBeepObj
class ATL_NO_VTABLE CBeepObj :
  public CComObjectRootEx,
  public CComCoClass<CBEEPOBJ, &CLSID_BeepObj,
  public IBeepObj
{
public:
  CBeepObj()
  {
  }

DECLARE_REGISTRY_RESOURCEID(IDR_BEEPOBJ)

BEGIN_COM_MAP(CBeepObj)
  COM_INTERFACE_ENTRY(IBeepObj)
END_COM_MAP()

// IBeepObj
public:
  STDMETHOD(Beep)(/*[in]*/ long lDuration);
};
```

This simple header file defines a tremendous amount of functionality, as described in the following sections.

Object Inheritance

Probably the first thing you noticed about this code is the multiple inheritance. Our COM object has three base classes. These base classes are template classes which implement the standard COM functionality for our object. Each of these classes defines a specific COM behavior.

CComObjectRootEx<> (and CComObjectRoot<>) are the root ATL object class. These classes handle all the reference counting and management of the COM class. This includes the implementation of the three required IUnknown interface functions: QueryInterface(), AddRef(), and Release(). When our CBeepObj object is created by the server, this base class will keep track of it throughout its lifetime.

The template for CComObjectRootEx specifies the argument CComSingleThreadModel. Single threading means that the COM object won't have to handle access by multiple threads. During the setup of this object we specified "Apartment threading". Apartment threading uses a windows message loop to synchronize access to the COM object. This approach is the easiest because it eliminates many threading issues.

CComCoClass<> defines the Class factories that create ATL COM objects. Class factories are special COM classes that are used to create COM objects. The CComCoClass uses a default type of class factory and allows aggregation.

IBeepObj is the interface this server implements. An interface is defined as a C++ struct (recall that structs in C++ act like a class but can have only public members). If you dig into the automatically generated file BeepServer.h, you'll find that MIDL has created a definition of our interface.

```
interface DECLSPEC_UUID(
"36ECA947-5DC5-11D1-BD6F-204C4F4F5020")
    IBeepObj : public IUnknown
    {
    public:
        virtual /* [helpstring] */ HRESULT
        STDMETHODCALLTYPE Beep(
```

```
                /* [in] */ long lDuration) = 0;
};
```

The DECLSPEC_UUID macro lets the compiler associate a GUID with the interface name. Note that our single method "Beep" is defined as a pure virtual function. When the CBeepObj is defined, it will have to provide an implementation of that function.

One peculiar thing about the Class definition of CBeepObj is the ATL_NO_VTABLE attribute. This macro is an optimization that allows for faster object initialization.

The Class Definition

Our object uses a default constructor. You can add special initialization here if required, but there are some limitations. One consequence of using the ATL_NO_VTABLE, is that you aren't allowed to call any virtual methods in the constructor. A better place for complex initialization would be in the FinalConstruct method (which is inherited from CComObjectRootEx.) If you want to use FinalConstruct, override ATL's default by declaring it in the class definition. It will be called automatically by the ATL framework. (FinalConstruct is often used to create aggregated objects.)

The DECLARE_REGISTRY_RESOURCEID() macro is used to register the COM object in the system registry. The parameter to this macro, IDR_BEEPOBJ, points to a resource in the project. This is a special kind of resource that loads the MIDL generated ".rgs" file.

BEGIN_COM_MAP is a macro that defines an array of COM interfaces that the CComObjectRoot<> class will manage. This class has one interface, IBeepObj. IBeepObj is our custom interface. It's common for COM objects implement more than one interface. All supported interfaces would show up here, as well as in the class inheritance at the top of the class definition.

The Method

At last, we get to the methods. As an application programmer, our main interest will be in this section of the code. Our single Beep() method is defined in the line:

```
STDMETHOD(Beep)(/*[in]*/ LONG duration);
```

STDMETHOD is an OLE macro that translates to the following:

```
typedef LONG HRESULT;
#define STDMETHODCALLTYPE       __stdcall
#define STDMETHOD(method)       virtual HRESULT STD-
  METHODCALLTYPE method
```

We could have written the definition in a more familliar C++ style as follows:

```
virtual long _stdcall Beep(long lDuration);
```

We'll find the code for this method in the BeepObj.cpp module. Because this COM object has only one method, the COM object's source code is pretty sparse. All the COM logic of the object was defined in the ATL template classes. We're left with just the actual application code. When you are writing real applications, most of your attention will be focused on this module.

```
STDMETHODIMP CBeepObj::Beep(long lDuration)
{
  ::Beep( 660, lDuration );
  return S_OK;
}
```

Again, the function definition translates into a standard function call.

```
long _stdcall CBeepObj::Beep( long lDuration )
```

The API beep routine takes two parameters: the frequency of the beep and its duration in milliseconds. If you're working with Windows 95, these two parameters are ignored and you get the default beep. The scope operator "::" is important, but it's easily forgotten. If you neglect it, the method will be calling itself.

The _stdcall tag tells the compiler that the object uses standard windows calling conventions. By default C and C++ use the __cdecl calling convention. These directives tell the compiler which order it will use for placing parameters on, and removing them from, the stack. Win32 COM uses the _stdcall attribute. Other operating systems may not use the same calling conventions. Notice that our Beep() method returns a status of S_OK. This doesn't mean that the caller will always get a successful return status - remember that calls to COM methods aren't like standard C++ function calls. There is an entire COM layer between the calling program (client) and the COM server.

It's entirely possible that the CBeepObj::Beep() method would return S_OK, but the connection would be lost in the middle of a COM call. Although the function would return S_OK, the calling client would get some sort of RPC error indicating the failure. Even the function result has to be sent through COM back to the client!

In this example the COM server is running as an In-Process server. Being a DLL, the linkage is so tight that there's very little chance of transmission error. In future examples, where our COM server is running on a remote computer, things will be very different. Network errors are all-too-common, and you need to design your applications to handle them.

Server Registration

The COM subsystem uses the Windows registry to locate and start all COM objects. Each COM server is responsible for self-registering, or writing it's entries into the registry. Thankfully, this task has been mostly automated by ATL, MIDL and the ATL wizard. One of the files created by MIDL is a registry script. This

script contains the definitions required for the successful operation of our server. Here is the generated script:

```
HKCR
{
    BeepObj.BeepObj.1 = s 'BeepObj Class'
    {
        CLSID = s '{861BFE30-56B9-11D1-BD65-
  204C4F4F5020}'
    }
    BeepObj.BeepObj = s 'BeepObj Class'
    {
        CurVer = s 'BeepObj.BeepObj.1'
    }
    NoRemove CLSID
    {
        ForceRemove {
        861BFE30-56B9-11D1-BD65-204C4F4F5020}
          = s 'BeepObj Class'
        {
            ProgID = s 'BeepObj.BeepObj.1'
            VersionIndependentProgID =
              s 'BeepObj.BeepObj'
            ForceRemove 'Programmable'
            InprocServer32 = s '%MODULE%'
            {
                val ThreadingModel = s 'Apartment'
            }
        }
    }
}
```

Registry Scripts

You may be familiar with .REG scripts for the registry. RGS scripts are similar but use a completely different syntax and are only used by ATL for object registration. The syntax allows for simple variable substitution, as in the %MODLUE% variable.

These scripts are invoked by the ATL Registry Component (Registrar). This was defined with a macro in the object header:

```
DECLARE_REGISTRY_RESOURCEID(IDR_BEEPOBJ)
```

Basically, this script is used to load registry settings when the server calls CComModule::RegisterServer(), and to remove them when CComModule::UnregisterServer() is called. All COM registry keys are located in HKEY_CLASSES_ROOT. Here are the registry keys being set:

- BeepObj.BeepObj.1 - Current version of the class
- BeepObj.BeepObj - Identifies the COM object by name
- CLSID - The unique class identifier for the Object. This key has several sub-keys.
  1. ProgID - The programmatic identifier.
  2. VersionIndependentProgID - Associates a ProgID with a CLSID.
  3. InprocServer32 - defines the server type (as a DLL). This will be different, depending on whether this is a In-Process, Local, or Remote server.
  4. ThreadingModel - The COM threading model of the object.
  5. TypeLib - The GUID of the type library of the server.

Summary

This chapter has provided a quick tour of most of the ATL code related to the Beep server. Do you now know everthing about ATL? No. But you now have a number of landmarks that will help you in navigating the code that the ATL wizard generates. See Chapter 15 for additional details.

S I X

Understanding the Client and Server

In the previous chapters we built simple client and server applications. The emphasis on was on getting a sample application up-and-running as quickly as possible. That's a great place to start. After all, building working components is ultimately what you want to get out of this book.

This chapter deals with some of the behind-the-scenes detail of what is going on. We are going to make short work of these subjects. That isn't because they aren't important, but because this book focuses on the practical implementation of COM. It's my experience that the theoretical parts of COM tend to obscure its simplicity. Once you are able to create useful clients and servers, the details of COM's implementation become more useful.

A certain amount of the theory of COM is necessary to properly use it. I've attempted to distill it into a few short but pithy segments.

Principles of COM

Let's start this discussion with five design principles that everyone who uses COM should understand:

- COM is about interfaces
- COM is language-independent.
- COM is built around the concept of transparency
- Interfaces are contracts between the client and server.
- COM is a "standard", not a compiler or language.

COM is About Interfaces

As we've said before, all COM interaction is through interfaces. It's a point worth repeating. You won't find any shortcuts or end-runs around this basic principle. The rationale behind interfaces is that it is critical to isolate a component from its user (client). Total isolation dramatically limits the amount of coupling between the client and server. In many ways COM was mis-named - it should have been called "i++".

COM is Language-Independent

Sometimes we programmers are so wrapped up in a particular language that we begin to see every programming problem in terms of it. I've written this book with a strong slant towards C++, and more especially Microsoft's Visual C++. There's a reason for this: you have to implement COM in some language, and C++ is a very good choice.

You can, however, write perfectly good COM programs in Java or C, or even Visual Basic. This means COM methods must be usable from many different languages. This includes languages like Visual Basic and Java that don't have pointers. The concept of an interface is easily expressed as a pointer, but it can be implemented without them. The most common expression of this we're likely to see is the use of the IDispatch interface in Visual Basic.

One of the essential parts of the COM standard is that it specifies how clients and servers communicate. In Visual C++, every COM method is called with "Pascal" calling conventions. While this isn't an absolute requirement of COM, it is a generally observed convention.

COM is Built Around the Concept of Transparency

In many cases, the COM server and client are running as different processes. Your program normally doesn't have access to address space on the other process. There are ways to get around this limitation, but not if the server is running on a computer elsewhere on the network. You can't even assume the computer you're connecting to is running Windows. A client can't directly access pointers, devices, or anything else on a remote computer running a DCOM server.

COM is therefore built around the concept of local/remote transparency. What transparency means is that the client should need to know nothing about how the server is implemented. This enormously simplifies the task of writing client programs. With COM, both an In-Process server and a remote server behave exactly the same as far as the client is concerned. Of course, there are real differences between an In-Process (DLL) client and a server running on a remote computer, but they aren't important to the client.

Much of the design of COM is aimed at hiding local/remote differences. Interfaces, for example, provide a mask that hides a great deal of behind-the-scenes implementation. COM defines the communication protocols and provides standard ways of connecting to other computers.

Interfaces are Contracts Between the Client and Server

A contract is an agreement between two or more parties to do (or not do) some definite thing. A good contract allows both parties to work independently without concern about the rules changing.

Even so, contracts are not perfect and they often have to be flexible. For example, you have to check that the server supports all the interfaces you are calling every time you connect. Once you've found an interface, the COM contract guarantees that the interface you want to use hasn't changed its methods or parameters. If an interface is available it should always behave in a predictable way. COM guarantees this simply by declaring that

Interfaces never change. If this seems like a dangerous method of enforcement, bear the following in mind:

COM IS A STANDARD, NOT A COMPILER OR LANGUAGE.

Actually, COM is a model. By model, we mean an 'ideal' standard for comparison. Unfortunately, the word model has a number of other meanings. A standard is a set of rules everybody agrees on. After all, even a computer language is actually just a special type of a standard. Usually language compilers provide you with nice features such as type and syntax checking. COM is a more loosely defined standard. It defines how clients and servers can communicate. If everybody follows the standard, communication will succeed.

The C++ compiler won't do any COM syntax and type checking for you. You have to know and follow the rules. Luckily, there is a tool that checks COM rules. It is an 'interface' compiler called MIDL. We've mentioned MIDL before several times; it is a compiler-like tool that generates COM-compliant code. You don't have to use MIDL. I'm not using MIDL in the simple client example seen in Chapter 2, mostly because it hides many important aspects of COM. When we get to more sophisticated applications (as we will in subsequent chapters), we'll use MIDL whenever possible. However, MIDL cannot guarantee that an interface has the right functions. Only programmers can guarantee that by following and enforcing conventions among themselves.

Software Changes. Interfaces Don't

This brings up the obvious question: What happens when you need to enhance or change an interface? There are two answers to this question depending on where you are in the software development cycle.

We talk about an interface being 'published'. This doesn't mean it has been submitted to some academic COM journal, rather that it has been made known to other users. This may mean a software release, or some written documentation, or even a conversation with fellow developers. In any case, once

people are using your interface, you cannot change it. Obviously, interfaces are going to need enhancement. This is accomplished by creating a completely new interface, and giving it a new name. One example of this process can be seen in the interface IClassFactory. When Microsoft needed to add licensing to this interface, they created a new one called IClassFactoryEx (Ex for extended). The extended interface is quite similar, and may even be implemented by the same coclass. We can rest assured that the original IClassFactory interface hasn't changed and will continue to function normally in older code. The new interface is something completely separate with a new name and a new GUID.

If you're in the midst of developing an interface and it hasn't been published, feel free to change it. The COM police aren't going to knock down your doors. It is incumbent on you to ensure that both the client and server know about the changes. If you change an interface on the server and don't update your client, there will obviously be consequences. Once the interface has been published, however, you need to leave it alone or you're going to have angry users.

Activation

The first time you successfully execute a COM client and server, you realize there's a lot going on to create components and start servers. When we ran the client application in the previous section there were several pieces of hand waving, and these pieces require some explanation if you want completely understand what is happening. Let's look at what happened "behind the scenes" so that it is clear that COM is doing nothing magic.

When the client application first called the server, a rather large collection of software components made the connection possible. The following figure shows you the components:

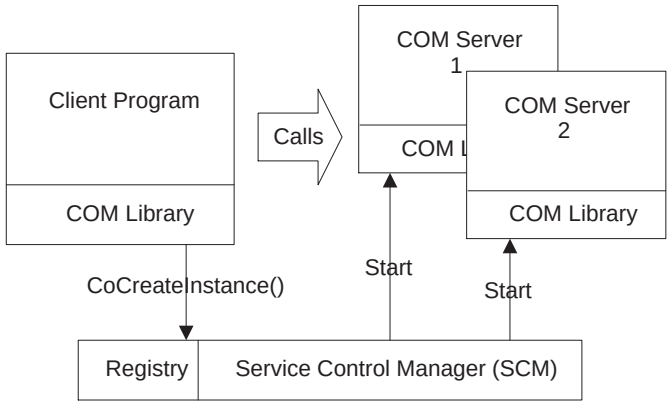

Figure 6–1 Components involved in COM interactions

There are several important components in this picture:

- The client and server applications.
- The COM library.
- The Service Control Manager.
- The Windows Registry.

The COM library is loaded into both the client and server modules as a DLL. The COM library contains all the "Co" API functions, like CoInitialize(). Currently the COM library is implemented in the OLE32.DLL module.

When the client calls CoCreateInstance(), it is calling a method in the COM library. CoCreateInstance does a number of things, but the first is to locate the requested components of the server in the Windows Registry. All the functionality of locating and starting COM components is handled by a COM "manager" application called the Service Control Manager (SCM). (in Windows NT the SCM is part of the RPCSS service.)

The SCM retrieves information about the server from the Window registry. The registry holds all of the GUIDs for all of the COM servers and interfaces supported by a given machine. The registry also maps those GUIDs to specific programs and

Services on the machine so that the COM servers can start automatically when they are called.

The Registry has entries for the different COM servers, their classes and interfaces. From this registry information, the SCM can start the correct server for any object requested by the client application. The SCM works with the COM library to create COM objects and return pointers to interfaces in objects.

For an in-process server, starting the server is rather simple. It involves locating and loading a DLL that contains the requested coclass. The SCM gets the path to the DLL from the registry, looking it up by the the GUID. For out-of-process servers, the SCM will actually start a new process for the server. For servers on remote computers, the SCM will send requests to the remote computer's SCM.

Once the server is started, and the client has a pointer to the interface, the SCM drops out of the picture. The SCM is only responsible for the activation of the server. Once everything is started, the client and server handle their own interaction. Like most networking, just getting communications started is a major task. Once communication is established, things tend to run quite well by themselves.

More About Interfaces

The end product of CoCreateInstance is a pointer to an interface. For the C++ programmer, an interface pointer looks exactly like a pointer to a C++ class. Do not be deceived: a COM interface is not a C++ class. An interface is a binary object with a rigidly defined internal structure. Although it looks a lot like a class, it lives by a different set of rules. This point seems esoteric, but it is very important.

Because of the special condition imposed on coclasses, you must follow these rules when you create a COM interface :

- All Interfaces must implement methods called QueryInterface(), AddRef(), and Release(). In that exact order. This fact is hidden by high level tools like ATL, but it has been

happening behind the scenes because of the activities of MIDL. These are the "Big Three" methods in all interfaces.
- Other methods follow, starting in the 4th position.
- Interfaces never change once they are published.
- Interfaces are strongly typed. There can be no ambiguity in parameters.
- Interfaces are named I*.

Here is how we define a simple interface with a single method. This definition was written in straight C++.

```
interface IBeep: public IUnknown
{
public:
  HRESULT QueryInterface(REFIID, void**);
  ULONG AddRef();
  ULONG Release();
  HRESULT Beep();
};
```

All COM interfaces are based on IUnknown. IUnknown always has three methods, QueryInterface, AddRef, and Release. These methods are pure virtual, which means they have no code associated with them. We also sometimes call this a pure abstract class. These three methods MUST be defined in our implementation of IBeep or the compiler will complain. IUnknown is defined in several of the standard headers. The definition is as follows:

```
#define interface struct

interface IUnknown
{
public:
  virtual HRESULT QueryInterface(REFIID, void**)=0;
  virtual ULONG AddRef()=0;
  virtual ULONG Release()=0;
};
```

You may not have seen the keyword 'interface' used before in C++. You'll be seeing a lot of it in COM programming. Here's how we defined interface:

```
#define interface struct
```

Unlike in the "C" language, in C++ a struct is the same as a class, except it has only public methods. Our definition of IUnknown would work exactly the same if we had written "class IUnknown" instead of using interface. In Visual C++, we use "interface" as a convention to remind us that COM has special rules. The definition of "interface" is compiler dependent, so this might not be true for other C++ implementations. The layout of the Interface is extremely important. All COM interfaces are defined in such a way that they provide QueryInterface, AddRef, and Release, and in this exact order. When the compiler processes this code, it will implement the interface using a C++ VTABLE structure.

VTABLES - Virtual Function Tables

A VTABLE, or virtual function table, is a pointer to an array of function pointers. Therefore, in all COM objects the first element points to QueryInterface, the second pointer points to AddRef, and the third points to Release. User defined methods can follow.

A VTABLE looks like this:

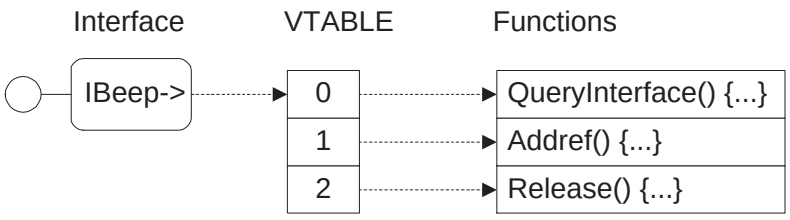

Figure 6–2 VTABLE structure

When you call a method through a VTABLE, you're adding a level of indirection. The interface pointer (IBeep->) points to the entry point in the VTABLE. The VTABLE points to the actual function. Calling functions through a VTABLE is very efficient.

Another way to look at a VTABLE is as an array of pointers. In this array, the first 3 elements are always the same. The function at element 0 is a pointer to a method you can use to discover other interfaces. This function is known as QueryInterface. The 2nd element is the address of a function that increments the reference count of the interface. The 3rd element, points to a function that decrements the reference count. The first 3 elements are all part of IUnknown, and every interface implements them.

After the first 3 elements of the array, we have pointers to interface-specific functions. In our example program, the 4th element points to the function Beep. All of the subsequent elements of the array point to custom methods of the interface. These methods are not implemented by the interface. It simply points to the address of the function in the coclass. The COM coclass is responsible for actually implementing the body of the functions.

Let's look at what happens when you call a method, QueryInterface for example. The program locates QueryInterface by looking in the first entry in the VTABLE. The program knows it's the first one because of the interface definition. This entry points to the actual location in memory of the method called QueryInterface(). After this, it follows standard COM calling conventions to pass parameters and execute the method.

Why is the order of these functions important? This gets back to the issue of language independence. You can use an interface even if you don't know its definition. However, you can only call the three standard methods QueryInterface, AddRef, and Release. This is possible because ALL COM interfaces have the same VTABLE footprint as the IUnknown interface.

To understand why this generic structure is useful, let's look at the methods in IUnknown. QueryInterface, AddRef, and Release. In our simple example client, we only see one of these functions - Release. Remember there's a lot going on behind the scenes in a COM program. We treat CoCreateInstance as if it

were a black box: Somehow it creates a pointer to a COM interface which our application can use. CoCreateInstance actually performs four distinct steps:

- Get a pointer to an object that can create the interface - the Class Factory.
- Create the interface with QueryInterface().
- Increment the reference count of the interface with AddRef()
- Destroy the class factory object with Release();

Maybe you're now starting to see how we use the three methods of IUnknown. They are being called all the time -- behind the scenes. Let's take a closer look at a class factory.

The Class Factory

A class factory is an object that knows how to create one or more COM objects. You call QueryInterface() on the class factory object to get a specific interface. You can write COM programs for years and never see a class factory. As far as the COM applications programmer is concerned, the class factory is just another part of the plumbing. If you're using ATL to generate your COM servers, the class factory object is hidden. ATL creates a default class factory that works for most COM objects. When you look into the actual code of CoCreateInstance, you'll find it's using a class factory. Here's the manual way of getting an interface. There's usually no reason to do this explicitly, unless you're optimizing the creation of interface objects. Looking at the code, however, sheds some light on what CoCreateInstance() really does.

```
// clsid - class that implements our interface
// pOuterUnk is NULL
// dwClsContext is the server context
// iid is the interface we're trying to create.
// pUnk will be returned
HRESULT CoCreateInstance( const CLSID& clsid,
          IUnknown *pOuterUnk,
          DWORKD dwClsContext,
          const IID& iid,
```

```
          void **pUnk )
{
  HRESULT hr;

  // return NULL if we can't create object
  *pUnk = NULL;

  IClassFactory *pFac; // a required COM interface

  // get a pointer to special class factory interface
  hr = CoGetClassObject( clsid, CLSCTX_LOCAL_SERVER,
    NULL, IID_IClassFactory, (void**)&pFac );

  if (SUCCEEDED(hr))
  {
    // use the class factory to get
    // the unknown interface
    hr = pFac->CreateInstance(pOuterUn, iid, pUnk );

    // release the factory
    pFac->Release();
  }

  // pUnk points to our interface
  return hr;
}
```

You probably noticed that the class factory is actually a COM interface. We had to call CoCreateInstance to get the class factory interface. Once you get the interface, you call its CreateInstance member. As you can see, the class factory does its job and then conveniently disappears.

There may be times when you'll want to override the default factory. One example might be a server that produces a large number of interfaces. For efficiency, you would want to keep this class factory in memory until it was finished with its work. To override the default, you'll need to write a custom implementation of IClassFactory.

We've now explained CoCreateInstance, but we've introduced two new mysterious functions: CoGetClassObject() and

CreateInstance(). CreateInstance() is a method of the COM standard interface IClassFactory. It creates a generic interface which we can use to get the IBeep interface. CoGetClassObject() is a bigger problem. A proper discussion of CoGetClassObject() belongs in the server side of our COM application. For now, we can think of it as the function that locates, starts, and requests a COM class from the server. The actual code of the class factory interface is implemented by the ATL template class CoComObject. CoComObject uses the macro DEFAULT_CLASSFACTORY, which implements the actual class factory.

Singleton Classes

ATL implements class factories through several macros.

DECLARE_CLASSFACTORY	The object will have standard behavior. CComCoClass uses this macro to declare the default class factory.
DECLARE_CLASSFACTORY_EX(cf)	Use this macro to override the default class factory. To use this you would write your own class factory that derived from CComClassFactory and override CreateInstance.
DECLARE_CLASSFACTORY2(lic)	Controls creation through a license. Uses the CComClassFactory2 template.
DECLARE_CLASSFACTORY_SINGLETON	Creates a singleton object. See the discussion below.

Table 6.1 Different class factory options

One of the more commonly used of these macros is DECLARE_CLASSFACTORY_SINGLETON. If you include this macro in your class header, the class will become a singleton.

A singleton object is a class that is only created once on a server. The single instance is shared by all clients that request it.

Singletons are a lot like global variables, in that everyone connected to the COM server shares them. Depending on the configuration of the COM server, the singleton can also be 'global' for the server computer. If your server has some shared resource that you want all clients to use, a singleton class might be a good choice.

Singleton objects are a lot more complicated than they may appear. You must be very careful in your application design and recognize the possible difficulties that singletons can present.

The most obvious problem with singletons is that they can easily become a resource bottleneck. Every client will have to share access to this single resource, and performance may suffer. You need to be sure the singleton object doesn't get tied up with time consuming processing.

There are a host of threading problems associated with singletons. Unless the object is free threaded, you're going to have threading issues. If your singleton keeps callback or connection points, it will not automatically call these interfaces on the proper thread, and you'll get errors. Despite this issue, you should probably implement your singletons as free threaded. That means you'll have to ensure that the code you write is completely thread safe.

Singletons also may not be unique. You often can't count on an object being the one-and-only instance of its class. This is especially true for in-process servers. In this case, the singleton isn't unique on the server computer. There will be a separate copy with each in-process DLL that gets loaded. If you're expecting one instance per computer, this won't work.

Finally, even out-of-process (EXE) servers may have multiple instances. Sometimes a server can be started for multiple login accounts. This means your singleton class can experience unexpected behavior depending on which severs get started.

Despite all of these caveats, there are places where a singleton class is appropriate. In general you will create it as part of a COM server implemented as an NT service and use it on the network to coordinate the activites of multiple clients.

Understanding QueryInterface

Interfaces are the most important concept in COM. At its lowest level, QueryInterface is extremely important in the implementation of interfaces. This function is being called behind the scenes, so we often don't see it in client programs. When using COM at the application level, we are more likely to see interfaces created through CoCreateInstance. If you delve very far into CoCreateInstance, you'll see that it is calling QueryInterface. If you start looking at the ATL generated code, you'll see that calls to QueryInterface are quite common. Although it is often hidden, it is important to understand what QueryInterface does, as well as the rules associated with it.

The purpose of QueryInterface is to get an interface from a COM object. Every COM object supports at least two interfaces. The first interface is always IUnknown. The second interface is whatever useful interface the object was designed to support. Many COM objects support several useful interfaces.

Once you have connected to IUnknown, you can get any of the other interfaces in a COM object. You pass in the IID of the requested interface, and QueryInterface will return a pointer to that interface. You can call any function in the given interface using that pointer. If the COM object doesn't support the requested interface, it returns the error E_NOINTERFACE.

```
hr = CoCreateInstance(
  clsid,           // COM class id
  NULL,            // outer unknown
  CLSCTX_SERVER,   // server INFO
  ID_IUnknown,     // interface id
  (void**)&IUnk );    // pointer to interface

if (SUCCEEDED(hr))
{
   IBeepDllObj *pBeep;

   hr=IUnk->QueryInface(
      IID_IbeepDllObj,(void**)&pBeep );
...
```

One of the interesting things about interfaces is that QueryInterface works backwards too. If you have the IBeepObj object, you can ask it for the IUnknown interface.

```
IUnknown *pUnk;

// Query IBeep for IUnknown interface
hr = pBeep->QueryInterface(
        IID_IUnknown,(void**)&pUnk);
```

In fact, you can get any interface from any other interface. For example, take a COM object that supports 3 interfaces, IUnknown, IA, and IB. We can query the IUnknown for either IA or IB. We could also query IA for IB, and vice versa. Obviously, you can't query any of these interfaces for IX, which isn't supported by the COM object.

Here are some of the rules of that you need to keep in mind when using QueryInterface:

- All COM objects support IUnknown. If it doesn't support IUnknown, it's not a COM object.
- You always get the same IUnknown interface. If you call QueryInterface multiple times for IUnknown, you will always get the same pointer.
- You can get any interface of a COM object from any other interface.
- There is no way to get a list of interfaces from an interface. (While this may sound interesting, it would be useless.)
- You can get an interface from itself. You can query interface IX for interface IX.
- Once published, interfaces never change.
- If you obtain a pointer to an interface once, you can always get it. See the previous rule.

Reference Counting with AddRef and Release

COM has no equivalent to the C++ "delete" keyword. Although there are several ways to create COM interfaces, there is no way

to explicitly delete them. This is because COM objects are responsible for managing their own lifetime.

COM uses the standard technique of reference counting to decide when to delete objects. The first time a client requests a specific interface, COM will automatically create a COM object that supports it. Once created, QueryInterface is called to get an interface pointer from the object. When you create an interface with CoCreateInstance or QueryInterface, AddRef is automatically called to increment the reference count. Each new interface increments the reference count.

When Release is called, the count decrements. When the count reaches zero, that means nobody is using the object anymore. At this point the object calls "delete" on itself.

Here is a fictional implementations if IUnknown and its three methods:

```
HRESULT _stdcall CSimple::AddRef()
{
  return m_nRefCount++;// increment count
}
HRESULT _stdcall CSimple::Release()
{
  if (--m_nRefCount == 0) // decrement count
  {
    delete this;// delete self
    return 0;
  }
  return m_nRefCount;
}
HRESULT _stdcall CSimple::QueryInterface(
const IID &iid, void **ppi )
{
  // make a copy of the "this", cast as an interface
  if (iid==IID_IUnknown)
    *ppi = static_cast< ISimple *>(this);
  else if (iid==IID_ISimple)
    *ppi = static_cast< ISimple *>(this);
  else
  {
    // invalid interface requested
```

```
    *ppi = NULL;
    return E_NOINTERFACE;
  }
  // automatically increment counter
  static_cast<IUnknown*>(*ppi)->AddRef();
  return S_OK;
}
```

As you can see, these methods don't do anything fancy. Every time a copy of the interface is made, the object increments its counter. As interfaces are released, the count decrements. When the count reaches zero, the object deletes itself. QueryInterface automatically calls AddRef, so you don't need to explicitly call it (so does CreateInstance.)

The "++" and "--" operators aren't thread safe, so this code could fail with free threaded applications. For this reason the API methods InterlockedIncrement and InterlockedDecrement are often used instead.

Reference counting offers some significant advantages to the client program. It relieves the client of any knowledge of the COM object's state. The client program's only responsibility is to call Release for each new or copied interface.

Obviously, if someone forgets to call Release, the object won't be destroyed. This means it will stay around for the life of the server. Worse yet, if Release is called too often, the object will destroy itself prematurely. Here are some basic rules for when to call AddRef and Release.

1. Do not call AddRef after functions that return interface pointers, such as QueryInterface, CoCreateInstance, and CreateInstance. It has already been called.
2. Call AddRef if you make a copy of a (non-null) interface pointer.
3. Call Release once for each AddRef that is called. (See #1)

Method Calls

Ultimately, the result of calling QueryInterface is that we end up with an interface pointer. In the previous section, the program was able to use the interface pointer to call a function in the interface.

```
pIf->Beep();
```

This code looks unremarkable. In our example code, we used an in-process server implemented as a DLL. For a DLL, calling a function is just a pointer away. This interface pointer is much more interesting if the server is an out-of-process server, or even a remote server. For a remote server, clearly pIf-> is not just a normal function pointer. This interface pointer can reach across process boundaries, and even the network to call its methods.

When thinking about a pointer, we normally think of it containing a memory address. Obviously it is impossible to directly access memory across the network. The way COM gets around this limitation is rather ingenious. COM handles its client/server communication through a pair of hidden communications objects. What appears to be a pointer directly to the server is actually a pointer into a communications object known as a proxy. The proxy's purpose is to modulate the flow of data between the client and server using a process known as marshaling. The proxy has a counterpart class, called a stub, which handles the server's end of the communication. The proxy and stub are either implemented as a separate DLL or built into the server and client applications.

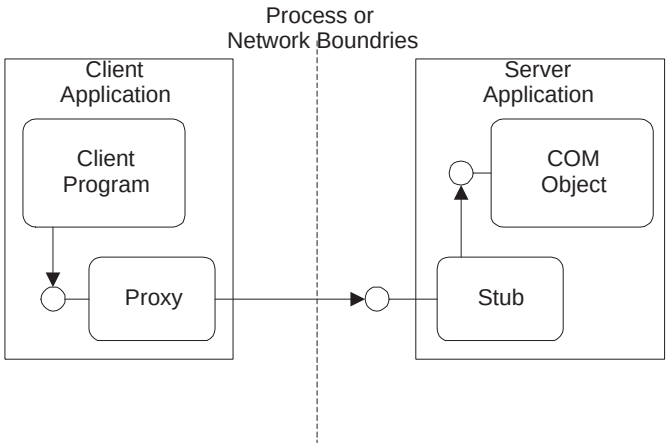

Figure 6–3 Relationship between the proxy and stub

The client program communicates with the proxy as if it were communicating directly with the server. The proxy in turn, works closely with the stub. Together, the proxy and stub handle all communication between the client and server. The server sees its input as if it were direct from the client, and the client can call functions using a pointer as though the server were in a DLL. The proxy and stub are transparent to each side of the application. In this way, the proxy and stub hide all the details of interprocess communication.

If you've ever done any communications programming, you will realize that the code hidden inside the proxy and stub is quite involved. The implementation of marshaling (which in COM means, "moving data across process or network boundaries") is not trivial. Luckily, you don't have to write marshaling code. We're going to generate our server using MIDL, which will automatically create the Proxy and Stub. Because of MIDL, the process is completely invisible and you generally do not have to think about it.

Not all COM interfaces use a proxy and stub. Some simple server models don't require any marshaling of data. Other inter-

faces use a type of marshaling known as "type library" marshaling. These servers are commonly known as dual or "dispatch" interfaces. One of the most important features of COM is that our client application can ignore all the infrastructure required to use the interface. The generation of Proxies and Stubs and all the marshaling will be taken care by the server. Happily, we can pretend our interface is a simple pointer. COM hides the implementation of the server from the client programmer.

COM Identifiers: CLSID AND IID

COM makes heavy use of GUIDs to identify items. When you call CoCreateInstance to get a specific interface, you need two pieces of information.

- The COM class that implements the interface
- The specific interface you wish to access

These two pieces of information are uniquely specified by the CLSID and IID respectively. Getting back to the example program, take a look at the first parameter of CoCreateInstance(). It's defined as a reference to a CLSID. Now, as we stated earlier, a CLSID is a type of GUID. If you look in AFXWIN.H you'll see that GUID, IID and CLSID are all the same structure.

```
typedef struct _GUID GUID;
typedef GUID IID;
typedef GUID CLSID;
```

Here's the initialization of the CLSID and IID struct. Note that there is only 1 hex digit different, but that's enough to make the two GUID's completely unique. They're so close because they were generated at about the same time.

```
IID iid =
  {0x50709330,0xF93A,0x11D0,{0xBC,0xE4,0x20,0x4C,0x4F,
  0x4F,0x50,0x20}};
CLSID clsid =
  {0x50709331,0xF93A,0x11D0,{0xBC,0xE4,0x20,0x4C,0x4F,
  0x4F,0x50,0x20}};
```

From the client's point of view, the COM class isn't especially important. As far as we're concerned, it's just a way to identify the server that will create our interface. If you're writing a COM Server, your perspective will be completely different. The COM Class is fundamental in the server.

The IID is the unique ID, which identifies the COM interface. It's important to note that an interface can be implemented by several servers. For example, an interface called IFly might be implemented by a coclass called CKite , CJet, and CGull.

CLSCTX -- Server Context

The CLSCTX parameter defines how the server will run. The three most common forms of this parameter are:

- CLSCTX_INPROC_SERVER : In-process server. The COM server is a DLL.
- CLSCTX_LOCAL_SERVER : Out-of-process server. The server runs on the same machine as a separate EXE or an NT service.
- CLSCTX_REMOTE_SERVER : The server runs on a remote machine as a separate EXE or an NT service.

In our first examples we're using CLSCTX_INPROC_SERVER, which means the server will run as part of our client process. Commonly, a client will use CLSCTX_SERVER, which allows either INPROC_SERVER and LOCAL_SERVER. This is the client's way of saying it doesn't care how the server is implemented.

Inheritance

One of the accepted principals of Object Oriented programming is the concept of inheritance. C++ supports and encourages inheritance, offering a rich set of techniques for its implementation.

COM is often criticized because it does not support inheritance. This is not an oversight on the part of COM's designers, but a necessary compromise. To understand this point of view, we have to look at the design goals of COM. COM is designed to

make components available across processes, networks, and even operating systems. This means COM has to ensure consistency and simplicity in components and their interfaces.

Object Oriented computer languages such as C++ are designed for a different purpose. They are designed to work on a single computer, and within a single process. C++ is optimized to efficiently create complex applications. A C++ compiler never has to work across a network, or run simultaneously across different operating systems (by this I mean more than simple network file access). C++ does not have built-in networking; it's just a compiler.

Another related issue is stability. Because COM is distributed, it needs to have a higher level of stability. Inheritance is, by definition, a very tight form of coupling. Coupling can introduce a level of instability into applications. When a base class changes, it can have severe repercussions on the classes that use it. Instability is contrary to the design principles of COM.

As we've said so many times, COM is built around interfaces. COM's answer to inheritance is interface inheritance. This means that you can inherit an interface layout, but you will have to implement the interface in your COM class. There is no special limitation on the C++ class that implements an interface, other than the fact that it must have a proper VTABLE structure. For a Visual C++ implementation, a coclass is just a C++ class, and you're free to inherit from whatever base class you desire.

Summary

In this chapter we have discussed a number of the details that apply “behind the scenes” to COM applications. Much of this information will make it easier to understand what is happening when a client connects to COM server, but most of these implementation details are hidden. Because they are hidden these details generally do not matter, but they may matter when a COM client or server contains a bug causing a failure. See Chapter 16 and the error appendix for details.

S E V E N

An Introduction to MIDL

MIDL stands for Microsoft Interface Definition Language. MIDL is a special interface 'language' and a compiler that generates, or “emits,” COM code. MIDL provides a standard way of defining COM interfaces and objects. The code generated by the MIDL compiler takes care of much of the grunt work of developing COM applications.

In then next three chapters we'll look at how MIDL fits in with the COM development process. We'll also look at MIDL's capabilities and syntax, as well as how to define and use a number common interfaces and their parameters.

Origins of the MIDL Compiler

As with much of COM, MIDL evolved from the Open Software Foundations Distributed Computing Environment, also known as DCE. The DCE way of calling procedures across networks is called RPC (Remote Procedure Calls). RPC is a useful standard, but it never became hugely popular because of implementation problems.

RPCs use an interface language called IDL. MIDL is just an 'Enhancement' of the IDL language that includes Microsoft's

COM extensions. Much of the COM IDL syntax is identical to RPC , and MIDL has the capability of processing RPC definitions.

COM and RPCs are actually quite closely tied together on Microsoft platforms. At a low level, COM uses RPCs as its communication method. This is, however, just a matter of convenience - COM can be implemented with almost any communication method if you are willing to write the marshaling code yourself.

MIDL is a language compiler. The source files of MIDL usually have the extension of "IDL". The MIDL compiler uses a syntax that is somewhat similar to C++, but it has a number of important extensions for COM.

Unlike a traditional compiler, MIDL does not generate object code (you can't link it). The output consists of several header files and a type library. These header files will be included into a C++ program and used to create object code. In many ways, the MIDL compiler is a code generator. It's interesting to note that MIDL generates stock C++ code wired straight into the Win32 API. It doesn't use ATL or MFC.

It's reasonable to ask why we need a special language for COM interfaces. After all, MIDL itself generates C++ code. The client and server will probably be implemented in a language such as C++, why can't we just use C++ syntax and write the MIDL output ourselves? To answer that question, let's look at some of the special abilities of MIDL. While it is technically possible to write MIDL's output "by hand", it wouldn't be much fun.

Precisely Defining Interfaces with the IDL Language

The level of precision required to define a COM interface is quite high. When working with remote objects you have to be very precise about how you pass data.

As C++ programmers, we commonly work with function (method) calls. When dealing with functions, we don't normally use the word interface. Every C++ function is like a COM method - it has a name, a return type, and a parameter list. If you think about it, C++ header files are similar to COM interface definitions because they expose functions to the outside world.

If we're working with C++ objects, we have class definitions. The C++ class is roughly equivalent to a COM interface definition. In the same way a class encapsulates a group of functions, so does the COM interface. C++ classes are of course much richer than COM interfaces because they can define data members and public and private members. A better C++ analogue to COM interfaces is a 'struct', which defines only public members.

There are several important differences between C++ classes and COM interfaces. One difference is that interfaces don't say anything about implementation. The very rough COM equivalent to a C++ object is a coclass, which behaves like an object (it can be instantiated on a server).

Defining interfaces (object definitions) in C++ is relatively easy because we're working in a controlled environment. If function parameters don't match, the compiler will let you know. If modules are incompatible, the linker will catch the problem. Of course, run-time errors are a lot harder to catch, but you still have extensive error handling, especially if you're testing your code in debug mode.

As the caller (client) and object (server) get farther away, the communication of data becomes more problematic. DLLs, for example, require more precise definitions than local function calls. A DLL must be able to dynamically load into memory, export some of its functions, and use an agreed-upon calling standard.

When the client and server are completely removed, as they are in distributed COM, there's a lot of room for miscommunication. The IDL language contains a rich set of attributes that precisely define the method of communication. Not only do you define objects and methods, but you explicitly describe how to transfer data.

Actually, you can define COM interfaces and objects without MIDL but it requires some complex and unusual syntax. In fact, MIDL converts your IDL definitions into "C" headers. If you take a look in these automatically generated header files, you'll find a lot of compiler directives and a level of complexity you've probably never seen before.

MIDL Generated Headers

IDL acts as the COM 'Data Dictionary'. You write your COM definitions in IDL, and run them through the MIDL compiler. MIDL takes your definitions and generates "C" and C++ language headers.

These header files are very useful to both the client and server application. You include the MIDL generated headers for their function prototypes and 'const' definitions. The same file is included in both the client and server, ensuring that they are compatible.

MIDL generates two basic types of header. The first is a standard C/C++ "H" header file. If you look in this file you'll see #include statements, #defines, typedefs, and object definitions - all the usual stuff in a header file. You also see a number of obscure compiler directives and macro's. There are also a number of #ifdef's and conditional compile statements (which seem to be one of the normal characteristics of computer generated code.)

Another header file, the "i.c" file, contains "C" definitions of all the GUID's used by the application. These GUID's are defined as "const" types. GUID's are quite long and difficult to type, so this header prevents a lot of mistakes.

Because "const" definitions are stored as variables, MIDL puts them in a "C" module, rather that a header. Our beep server example would generate a file named "BeepServer_i.c". Unlike a normal "C" file, this file is actually intended to be included with a "#include".

Automatically Generated Proxy/Stub Modules

In this book we do not spend much time talking about Proxy/Stubs and marshaling. This is because these topics are handled automatically by the MIDL compiler. You have to be aware that marshaling is going on, but you won't have to actually implement it.

One of the most endearing qualities of MIDL is that it writes Proxy and Stub definitions based on your IDL code. This is called "Standard Marshaling". If you write your own marshaling

it's called "Custom Marshaling". The need to use custom marshaling is rare in normal COM applications. The implementation can be complex and time consuming. I cover some simple marshaling using the CoMarshalInterThreadInterfaceInStream method call. Kraig Brockschmidt's book *Inside Ole* is a good reference for this topic.

Automatic Creation of Type Libraries

A type library is a data file that contains the description of COM objects and interfaces. In other words, a type library is the binary representation of the IDL file. Type libraries are used heavily by dual and IDispatch interfaces. The #import directive in C++ (V6 and V5) directly imports the type library, and is probably the easiest way to use COM in a client.

Type libraries are an interesting subject unto themselves. We cover them in more detail in Chapter 9.

MIDL can generate a type library for your COM application. Back in the earlier days of OLE, type libraries were generated by a utility named MKTYPLIB. MKTYPLIB used a language called ODL (Object Description Language) that looks remarkably like IDL. Actually, MIDL can quite happily process ODL source code.

The IDL Language

IDL is designed specifically to define all the aspects of COM communication. For C++ programmers, the syntax will be familiar. IDL uses "C" constructs for almost everything, but adds several COM specific attributes.

Unlike C++, IDL just supports definitions. You can't actually write programs in IDL. The source files have an extension of "IDL". You can look at IDL files as the COM equivalent of ".H" files in C++.

Although it's an "interface" definition language, IDL does a lot more than define interfaces. Here are some of the things you define with IDL.

- COM interfaces

- Individual Interface Method (Function) definitions.
- Parameters - Detailed information about how parameters are passed through COM.
- COM Class definitions (coclass)
- Type libraries
- Types - A variety of data types

The MIDL compiler can be invoked directly from a command prompt:

```
C:\> midl BeepServer.idl
```

MIDL will process the source files and generate output. The MIDL compiler has been integrated with Version 6.0 of visual studio. This is a big change from earlier versions of Developer studio, making COM a more integrated part of the development environment. You can see the MIDL settings by opening the project settings and selecting the IDL file of your project. There is a special MIDL tab under settings.

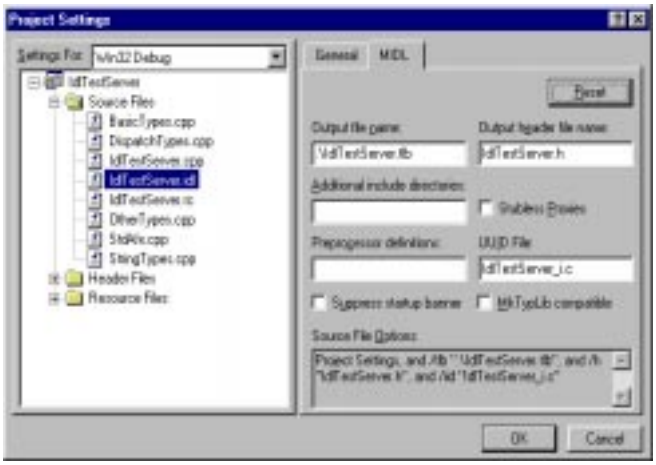

Figure 7–1 MIDL settings

In previous versions in the MIDL command was built into the “Custom Build” step of the project. When you use the ATL AppWizard to generate a project, it will include the execution of the MIDL compiler. If you want to see where the MIDL command resides, do the following: 1) Click on the “FileView” tab of the workspace. Highlight the IDL source file. 2) select the “Project” menu, and look at the “settings”. The “Project Settings” dialog will be displayed. 3) Find the IDL source file under the source files, and look at its custom build tab. What you’ll see is a command under the build commands that looks like the following:

```
midl /Oicf /h "BeepServer.h" /iid "BeepServer_i.c"
  "BeepServer.idl"
```

The workspace automatically executes this command whenever the IDL source is modified. If you're curious about the MIDL command line, there is some help available on the options.

Interfaces and Methods in IDL

When you start to look at IDL source code, you'll notice the similarities with C++. MIDL actually uses the C++ pre-processor to parse the source file, you'll immediately recognize that it accepts C++ style comments.

The interface definition is divided into two units -- attributes and definitions. The attributes are always enclosed in square brackets. In the following example, the interface has three attributes, a uuid, a help string, and a pointer_default. The significant attribute is the uuid. Uuid is the GUID that identifies this interface. The Wizards also generate a lot of help attributes like helpstring and helpcontext. The Help information is mostly used in type libraries, and is displayed by object browsers like the one in Visual Basic.

```
// A simple interface
[
  uuid(36ECA947-5DC5-11D1-BD6F-204C4F4F5020),
  helpstring("IBeep Interface"),
```

```
  pointer_default(unique)
]
interface IBeep : IUnknown
{
};
```

There are a number of other interface attributes. One of these is the “dual” attribute , which means the interface supports both a custom COM interface (IUnknown), and an OLE automation interface (IDispatch). Dual interfaces offer the flexibility of connecting through the early binding IUnknown interface, or a late binding IDispatch interface. See chapter 9 for a detailed discussion of Dispatch interfaces.

COM allows interface inheritance. All interfaces MUST implement the methods of IUnknown. The IBeep interface has IUnknown as part of IBeep's inheritance. In COM, there is no concept of public and private. Anything we can define in an interface is, by default, public.

Often you will see interfaces that inherit from the IDispatch interface. These interfaces usually have the "Dual" or "oleautomation" attribute. Like all COM interfaces, IDispatch also inherits from IUnknown.

There's nothing to stop you from inheriting from your own custom interface, as long as that interface implements IUnknown. Multiple inheritance is not allowed for IDL interfaces, but objects often do have multiple interfaces. If you really want to see multiple inheritance, you'll get plenty in ATL - in ATL inheritance is used as a mechanism to bring multiple interfaces into an object.

As you get into the OLE world, you'll find there are numerous standard interfaces you need to implement. One typical OLE interface is IPersistFile. OLE requires that you implement this interface for several types of objects that must be able store themselves and be retrieved from disk files. This interface is well known: it's GUID (always 0000010b-0000-0000-C000-000000000046) and methods are defined in the standard IDL files in the C++ include directories. You just have to include the inter-

face with an "import" statement. IPersistFIle is defined in "OBJIDL.IDL". The statement looks like this:

```
import "oaidl.idl";
```

If you implement IPersistFile in your COM object, you'll need to include it in a COM class and implement all its methods. Remember that MIDL just provides definitions. This isn't "real" object inheritance, where you inherit all the behaviors of an interface. You're just inheriting the definitions.

Let's take a look at some of the more common interface attributes.

Attribute	Usage
Dual	Supports both dispatch (IDispatch) and custom (IUnknown) interface. Custom interfaces are sometimes called VTABLE interfaces.
Oleautomation	The interface is compatible with OLE Automation. The parameters and return types in the interface must be automation types.
Uuid	Associates a unique identifier (IID) with the interface. This id distinguishes the interface from all other interfaces.
Version	Associates a version with the interface. It looks useful but this attribute is not implemented for COM interfaces. It is valid for RPC interfaces.
Pointer_default()	Specifies the default pointer attributes applied to pointers in the interface. Pointer attributes can be **ref**, **unique**, and **ptr**.

Table 7.1 Interface Attributes

Interfaces are just containers for a group of methods. An interface can have any number of methods. Here's a very simple MIDL method definition:

```
HRESULT Beep([in] long lDuration);
```

MIDL generates a header that defines this method in the C++ code. Here's what the C++ equivalent of this statement looks like:

```
virtual HRESULT _stdcall Beep(/*[in]*/ long lDura-
  tion);
```

As you can see, attributes have no equivalent in C++. The attributes are retained in C++ headers as comments. While it may not be used in the C++ headers, the attributes are very important for other aspects of the code generation. In this case MIDL used the "[in]" attribute to produce marshaling code which transfers a variable of type "long" to the server.

The return code of a COM method is almost always an HRESULT. The two well-known exceptions to this rule are the methods of IUnknown - AddRef() and Release(). The return values for AddRef and Release are used only for debugging reference counting problems. See chapter 4, chapter 16 and the error appendix for more information on HRESULTs.

There are only a few method attributes you are likely to see. These attributes are all associated with Dispatch interfaces. The most common is the “ID” attribute, used to specify the dispatch ID.

The Component Class in IDL

A COM class definition is called a coclass, for "component class". A coclass is a top-level object in a COM object hierarchy. Here's a typical coclass definition:

```
[
  uuid(543FB20E-6281-11D1-BD74-204C4F4F5020)
```

```
]
coclass BasicTypes
{
  [default] interface IBasicTypes;
  interface IMoreStuff;
};
```

Syntactically, there's not much to a coclass definition. Each coclass has a GUID and a list of interfaces. The attribute [default] looks important, but it is intended for use by macro languages such as WordBasic. Strangely enough, a coclass can have two default interfaces - one for an outgoing interface (source), another for the incoming (sink) interface. See chapter 13 on call-backs for more information about sources and sinks. For the most part, we can ignore the default attribute.

When we write source code, the coclass usually maps directly into a C++ object. There's no rule that says a coclass corresponds to a C++ class. Actually, you can write COM definitions in "C" without any class definitions at all. However, since we're using ATL, there will be a one-to-one correspondence between the two. The C++ object will be the coclass name with a preceding "C". In this case it's "CBasicTypes". Here's the code generated by the ATL wizard:

```
class ATL_NO_VTABLE CBasicTypes :
  public CComObjectRootEx,
  public CComCoClass<CBASICTYPES, &CLSID_BasicTypes,
  public IBasicTypes,
  public IMoreStuff
{
. . . class definition
};
```

Notice that two of the base classes for CBasicTypes are "IBasicTypes" and "IMoreStuff". These are the two interfaces included in the coclass. In ATL each interface in the coclass will be included in the class definition. This means the interface will have to implement the C++ code for all the methods in both interfaces.

Another consequence of grouping interfaces in a coclass is that they share the same IUnknown interface. It's one of the fundamental rules of COM that all interfaces in a COM class will return the same IUnknown pointer when you call QueryInterface().

Type Libraries in IDL

As we said earlier, one of the nice features of MIDL is that it automatically creates type libraries. Actually, the generation of type libraries isn't quite automatic - you have to insert a definition in the IDL source.

```
[
  uuid(543FB200-6281-11D1-BD74-204C4F4F5020),
  version(1.0),
  helpstring("IdlTest 1.0 Type Library")
]
library IDLTESTLib
{
  importlib("stdole32.tlb");
  importlib("stdole2.tlb");

  [
    uuid(543FB20E-6281-11D1-BD74-204C4F4F5020)
  ]
  coclass BasicTypes
  {
    [default] interface IBasicTypes;
  };

}
```

Anything included between the curly braces of the library definition will be include in the library. In this case, the interface IBasicTypes was defined outside the library, and included in the coclass definition. MIDL will also pull in the definition of IBasicTypes in the library.

MIDL will only generate proxy/stub implementations for definitions outside the library statement. Sometimes you will see

items such as interfaces defined inside the library definition. These objects will only be seen from inside the type library.

The "importlib" statements are used to include the type library definitions from another type library. If a client needs these definitions, it will have to get them from the external library. In this case, the standard OLE libraries are imported.

MIDL Post-Processing

The ATL application wizard automatically includes the build command for the IDL file in the project. In version 4 and 5, the custom build step was responsible for invoking the MIDL compiler. With the Version 6, the MIDL step was integrated with developer studio.

The custom build is still responsible for server registration. For EXE servers, the custom build executes the server program with the '/RegServer' command. Note that '$(TargetPath)' and '$(OutDir)' are environment variables that the build process will replace with the name and output directory of your server.

Commands:

```
"$(TargetPath)" /RegServer
echo regsvr32 exec. time > "$(OutDir)\regsvr32.trg"
echo Server registration done!
```

Outputs:

```
\$(OutDir)\regsvr32.trg
```

For a DLL based in-process server, the registration is handled exclusively by REGSVR32. The build command would be as follows:

```
regsvr32 /s /c "$(TargetPath)"
```

The TRG file is a dummy file that's necessary for the proper execution of the NMAKE and REGSVR32 command. This dummy file is created when the custom build step executes, and gets the

current timestamp. One of the rules of makefiles is that there is an output file for each action. The build process can then use the timestamp of this file to determine if it needs to execute the custom build command. If you look in the TRG file you'll find there isn't anything useful there.

THE POST BUILD STEP

The custom build step is somewhat obsolete for registering a server. The ATL wizard still generates a custom build step, so it is the default for most projects. A better way to do this would be to add a command to the Settings/Post Build Step tab. You can remove the three commands and the dummy target (TRG) file from the custom build. Add the following to the post-build settings tab instead.

```
$(TargetPath) /RegServer
echo "Registration Complete"
```

You can also automatically include the proxy/stub DLL build in this step. The following commands will build the proxy/stub, and register it using the REGSVR32 command. Once again, the post build step doesnít require a target. We're using the '$(WkspName)' variable, and appending 'PS' to the end. You may also want to add an 'echo' statement to inform the user whatís happening.

```
nmake $(wkspName)PS.mk
regsvr32 -s $(WkspName)PS.dll
```

Actually, you only need to build the proxy/stub DLL when the MIDL code changes. This is true because the Proxy/Stub is only concerned with marshaling data between the client and server, and doesn't deal with the implementation of either. This post-build processing will cause some unnecessary processing, although NMAKE is pretty good about rebuilding only what it needs.

THE PLG FILE

The PLG file is a build log. It contains the output of your build command, and can be useful when trying to diagnose build problems. The PLG file will show you the commands executed, with all their copious switches and results.

Summary

Thus far we've looked at the origins and capabilities of the MIDL compiler. This tool offers a very powerful way to define COM classes, interfaces, and methods. MIDL also generates a great deal of the code required to quickly build servers. MIDL is also a code generator. It automatically produces headers, proxy/stub DLL, and type libraries.

In the next chapter, we'll look in detail at some of the syntax of IDL. We'll also give examples of using the most common IDL data types.

E I G H T

Defining and Using Interfaces

There are lots of decisions to make when you're setting up COM interfaces. Not only do you have to set up the usual function parameters, but you also have to decide how COM will transfer the data.

Transferring data is what COM is all about. When you're working with standard C++ programs and DLL's, you take data transfer for granted. The only real decision for a C++ programmer is whether to pass parameters by value or to use pointers. When you're working with remote servers, the efficiency of data transfer can be of critical importance to an application. You've got to make a lot of decisions about data types, transfer methods, and interface design. MIDL gives you the tools to work with these issues.

MIDL gives you a number of data types. These include the basic data types, OLE Automation types, structures, arrays and enums. Most of these map easily to the C++ types your application uses. The unfamiliar part of this process is the definition of parameter attributes. The are two basic attributes, [in] and [out], which define how COM will marshal the data.

Base Types

MIDL's base types, such as long and byte, should be immediately familiar to a C++ programmer. Base types are the fundamental data types for IDL; most other types are based on them.

Type	Bits	Description
boolean	8	Can have a value of TRUE or FALSE. These states are represented by 1 and 0 respectively. Note that the C++ type BOOL is 32 bits. MIDL handles it as an unsigned char.
byte	8	May have unsigned values of 0-255. Bytes are considered "opaque" which means MIDL doesn't make any assumptions about content.
char	8	An unsigned 8 bit character. Range 0-255.
int	32 or 16	An integer value. The size depends on the platform. On a 32 bit platform such as windows 95, int's are 32 bit long words. On 16 bit platforms they are 16 bit words.
long	32	A long word. Can be signed or unsigned. The default is signed.
short	16	A signed or unsigned number.
hyper	64	A special 64 bit integer.
float	32	A low precision floating point number. A float in C++. A 32 bit IEEE floating-point number.
double	64	Equivalent to C++ double. A 64 bit IEEE floating-point number.
wchar_t	16	Wide characters (Unicode). This type maps to an unsigned short.

Table 8.1 MIDL's base types

Most integer and char types can have a **signed** or **unsigned** attribute. The **signed** keyword indicates that the most significant bit of an integer variable holds the sign rather than data. If this bit is set, the number represents a negative number. The **signed** and **unsigned** attribute can apply to **char**, **wchar_t**, **long**, **short**, and **small**.

Attributes

A basic IDL method definition looks something like this:

```
HRESULT InCount( [in] long lCount );
```

All COM methods must be qualified with "in" or "out", or both. These are called "Directional" attributes, and they tell MIDL how it should marshal the data.

Attribute	Usage
in	The parameter is only passed to the server. It is not returned. All parameters default to [in] if no directional attribute is specified.
out	The parameter is only returned from the client. It must be a pointer type.
in,out	The parameter is passed to and returned from the server. It must be a pointer.

Table 8.2 COM Directional Attributes

The "[in]" attribute directs MIDL to generate code which will make a copy of the parameter and send it to the server. This is similar to "call by value" in C++, except the data is not copied to the stack. Instead, the marshaling code will pass the parameter to the proxy/stub DLL, which will package it to be copied to a server. If the server modifies the data, COM will not make any attempt to copy it back to the client.

Out parameters are also "one-way" data. Here is an IDL method with an '[out]" parameter:

```
HRESULT OutCount( [out] long *plCount );
```

Out parameters always have to be pointers. In this case, the server will fill in the pointer with the value of the count. This function isn't as simple as it seems - there are several important ambiguities here. Where is the pointer allocated? Can it be NULL?

When working locally in C++, pointers are a very efficient way to pass data. By using pointers, C++ avoids passing large amounts of data. If you're using an in-proc server, this is still true - because an in-proc server is a DLL. If you're working with a local or remote server, the picture is entirely different.

For remote objects, the use of pointers doesn't save very much. Both **in** and **out** parameters must be marshaled across the network. Passing pointers has just as high a cost in overhead. The client and server don't share the same address space, so all the data referenced by the pointer is copied.

To allow more efficiency in transferring data, COM gives you several pointer attributes. There are three different types of COM pointer: **ref**, **unique**, and **ptr**.

Attribute	Usage
ref	• The parameter is a reference pointer. • A reference pointer can always be dereferenced. • The pointer (address) will never be changed during the call. • The pointer must always point to a valid location, and can never be NULL. • You cannot make a copy of this pointer (no aliasing). • The most efficient type of pointer.
unique	• The parameter is a unique pointer. • It can be NULL or changed to NULL. • The server can allocate or change the pointer (address). • Pointed-to object cannot change size. • You cannot make a copy of this pointer (no aliasing).
ptr	• The parameter is a full pointer. • It can be NULL or changed to NULL. • The server can allocate or change the pointer (address). • Aliasing is allowed. • Similar to a C++ pointer. • The least efficient type of pointer.
retval	• The parameter receives the return value of the method. • The parameter must have the [out] attribute and be a pointer. • Used in oleautomation interfaces.

Table 8.3 COM Pointer Values

In the previous examples we didn't specify which type of pointer to use. The default pointer type for in parameters is "ref". This means the pointer references memory allocated on the client. This is the most restrictive type of pointer. The ref pointer is allocated on the client, and its address is passed to the server. Here's the full-blown version of the LongTest() method:

```
STDMETHODIMP LongTest(long lIn, long * plOut)
{
  if (plOut == NULL) return E_POINTER;
  *plOut = lIn + 10;
  return S_OK;
}
```

This method does not allocate any memory, which is consistent with a ref pointer. In this case the server method will add 10 to the in parameter and assign it to the out parameter.

Since we're conscientious programmers, we added in some pointer checking. This isn't strictly necessary for this interface. If you did call LongTest with a NULL pointer, you'll get an error. The stub code would reject the call with an error of RPC_X_NULL_REF_POINTER - our method would never be called. If the pointer had been declared was unique or ptr, the NULL check would be important.

Let's take a quick review of the code for this interface.

MIDL DEFINITION

```
HRESULT LongTest([in] long l, [out] long *pl);
```

SERVER CODE

```
STDMETHODIMP CBasicTypes::LongTest(long l, long * pl)
{
   if (pl == NULL) return E_POINTER;
   *pl = l + 10;
   return S_OK;
}
```

CLIENT USAGE

```
HRESULT hr;
long l1=1;
long l2=0;
hr = pI->LongTest( l1, &l2 );
```

The client is then calling the LongTest() method. The out parameter "l2" is allocated on the local program's stack, and its address is passed to the server. When the call returns we would expect an HRESULT value of S_OK, and the value of pointer l2 to be eleven.

Double Parameters

What if we're using a double parameter? The code is almost identical for all the base types. Here's the IDL code for that interface.

MIDL DEFINITION

```
HRESULT DoubleTest([in] double d, [out] double *pd);
```

SERVER CODE

```
STDMETHODIMP CBasicTypes::DoubleTest(double d, double
  * pd)
{
   if (pd == NULL) return E_POINTER;
   *pd = d + 10.0;
   return S_OK;
}
```

CLIENT USAGE

```
HRESULT hr;
double d1=1;
double d2=0;
hr = pI->DoubleTest( d1, &d2 );
```

Boolean Parameters

The IDL boolean type can be a little confusing. In IDL, boolean is a single byte. This makes sense in the context of COM, which needs to optimize communications by passing the smallest amount of data possible. Unfortunately, C++ defines a BOOL as a 32-bit longword. You have to take this size differential into account, either by using an IDL long, or by casting the parameter from BOOL to a single byte.

Working with Strings

There are three basic types of string allowed in MIDL.

- char arrays.
- Wide character strings.
- BSTR. Visual BASIC style strings.

The default string type for C++ is of course an array of 8-bit **char**'s with a NULL terminator at the end. This type maps to the MIDL "unsigned char" type. There's little difference between arrays of signed and unsigned characters, and arrays of bytes. Any of these choices will work.

If you're working with international software, you have to use "wide" 16 bit Unicode characters. There are several ways to define these in MIDL. The most common types used are "unsigned short" and "wchar_t". The MIDL wchar_t type has an identical type in C++.

There are two attributes that are commonly used in association with MIDL strings. These are the **string** and **size_is** attribute.

Character and Wide Character strings are arrays. The only thing that distinguishes an array from a character string is the NULL terminator at the end. The string attribute tells MIDL that it can determine the length of the string by searching for the NULL terminator. The generated code will automatically call the appropriate strlen(), lstrlen(), or wcslen() function. The string attribute

is not required for passing strings, you can use the size_is attribute to accomplish the same thing.

Attribute	Usage
string	An array of char, wchar_t, or byte. The array must be terminated by a null value.
size_is()	Specifies the number of elements in an array. This attribute is used to determine string array size at run time.

Table 8.4 COM String Attributes

Here are two examples of interfaces that use the string attribute to pass data to the server:

MIDL DEFINITION

```
HRESULT SzSend([in, string] unsigned char * s1);
HRESULT WcharSend([in,string] wchar_t *s1);
```

SERVER CODE

```
STDMETHODIMP CStringTypes::SzSend(unsigned char * s1)
{
  m_String = s1;
  return S_OK;
}
STDMETHODIMP CStringTypes::WcharSend(wchar_t *s1)
{
  long len = wcslen(s1) * sizeof(wchar_t);
  m_pWide = new wchar_t[len+1]; // add 1 char for null
  wcscpy( m_pWide, s1 );
  return S_OK;
}
```

CLIENT CODE

```
char s1[] = "Null Terminated C String";
pI->SzSend( s1 );
wchar_t w1[] = L"This is Wide String";
pI->WcharSend( w1 );
```

These interfaces offer a simple way to pass character strings to a server. This is one of the few cases where I would recommend using the string attribute. For [in,out] parameters, the string attribute can be dangerous. MIDL calculates the size of the transmission buffer based on the size of the input string. If the server passes back a larger string, the transmission buffer will be overwritten. This type of interface can be extremely buggy.

Sometimes we can't use NULL terminated strings. In these cases, you can explicitly specify the length of the string array. There are two methods of doing this. You can use a fixed length string, or you can use the size_is attribute.

A fixed length array would look like this:

MIDL DEFINITION

```
HRESULT SzFixed([in] unsigned char s1[18]);
```

SERVER CODE

```
STDMETHODIMP CStringTypes::SzFixed(unsigned char
  s1[18])
{
  m_String = s1;
  return S_OK;
}
```

CLIENT CODE

```
unsigned char sf[18]= "An 18 byte String";
pI->SzFixed( sf );
```

The Server definition would also require a fixed length declaration in the parameter. Unless you're passing a buffer that is always a fixed size, this is an inefficient way to design an interface. It's also quite easy to inadvertently overrun the boundaries of a fixed array (the classic mistake is to forget space for the NULL terminator.) A better way would be to specify the number of bytes at run-time. The size_is attribute tells the marshaling code exactly how many bytes are actually being passed.

IDL will generate marshaling code to pass exactly the number of bytes required. If you are dealing with NULL terminated [in] strings, this offers no advantages over using the "string" attribute. We'll see similar syntax when we examine IDL array types. Here's the server code to handle a string with an explicit size:

MIDL DEFINITION

```
HRESULT SzSized ([in] long llen,
  [in,size_is(llen)] unsigned char * s1);
```

SERVER CODE

```
HRESULT SzSized(long llen, unsigned char *s1)
{
  char *buf = new char[llen+1];// temp buffer
  strncpy( buf, (char *)s1, llen );// copy string
  buf[llen]=NULL;// add null to end
  m_String = buf;// copy into CString
  delete[] buf;// delete temp
  return S_OK;
}
```

CLIENT CODE

```
  char s1[] = "Null Terminated C String";
  pI->SzSized( strlen(s1), (unsigned char*)s1 );
```

The size_is attribute is often used when returning data in a string. The string pointer is given an [out] attribute, and its maximum size is specified in the first parameter. In this example, the variable "len" specifies the size of the string being sent by the client.

MIDL DEFINITION

```
HRESULT SzRead([in] long len, [out,size_is(len)] char
  *s1);
```

SERVER CODE

```
STDMETHODIMP SzRead(long len, unsigned char * s1)
{
  strncpy( (char*)s1, m_String, len );
  return S_OK;
}
```

CLIENT USAGE

```
char s2[64];
pI->SzRead( sizeof(s2), s2 );
```

The server knows the maximum size of the return buffer, so it can ensure that it isn't overwritten. The problem here is that we're passing around a number of unused bytes. The len parameter specifies how many bytes will be transferred, not how many are actually used. Even if the string were empty, all 64 bytes would still be copied. A better way to define this interface would be to allocate the string memory on the server side and pass it back to the client.

To accomplish this we pass a NULL pointer to the server and allocate the buffer using CoTaskmemAlloc. This function allows the allocated memory to be marshaled back from the server to the client, and deleted by the client. The client will call CoTaskMemFree when it is finished with the pointer. Together these two functions are the COM equivalent of new and delete.

MIDL DEFINITION

```
HRESULT SzGetMessage([out,string] char **s1);
```

SERVER CODE

```
STDMETHODIMP CStringTypes::SzGetMessage(unsigned char
  ** s1)
{
  char message[] = "Returned ABC 123";
  long len = sizeof(message);
```

```
    *s1 = (unsigned char*)CoTaskMemAlloc( len );
    if (*s1 == NULL) return E_OUTOFMEMORY;
    strcpy( (char*)*s1, message );
    return S_OK;
  }
```

CLIENT USEAGE

```
char *ps=NULL;
pI->SzGetMessage(&ps);
// use the pointer
CoTaskMemFree(ps);
```

In this example we used the [string] attribute to determine the length of the string buffer. We could just as easily have used [size_is] and explicitly determined the buffer size.

BSTR is a data structure that contains an array of characters, preceded by the string length. BSTR's are NULL terminated, but they aren't simple arrays. There can be several null terminators in a BSTR, but the last counted element in the array will always have a NULL terminator. If you're using dual or oleautomation interfaces, you will need to work with BSTR's.

BSTR's are a difficult type to use in C++. You shouldn't try to manipulate them directly. Fortunately there are several helper classes and functions available.

- SysAllocString, SysFreeString create and destroy BSTR's.
- CString::AllocSysString and CString::SetSysString.
- bstr_t encapsulates the BSTR class in C++
- The ATL CComBstr wrapper class.

MIDL DEFINITION

```
HRESULT BsSend([in] BSTR bs);
HRESULT BsRead([out] BSTR *pbs);
```

SERVER CODE

```
BSTR m_BSTR;

STDMETHODIMP CStringTypes::BsSend(BSTR bs)
```

```
{
    m_BSTR = bs;// save value in local
    return S_OK;
}

STDMETHODIMP CStringTypes::BsRead( BSTR *pbs)
{
    CComBSTR temp;
    temp = m_BSTR;
    temp.Append( " Returned" );
    *pbs = temp;
    return S_OK;
}
```

CLIENT USEAGE

```
wchar_t tempw[] = L"This is a BSTR";
BSTR bs1 = SysAllocString(tempw);
BSTR bs2;
pI->BsSend( bs1 );
pI->BsRead( &bs2 );
```

Note that the string was first created as a wide character string (wchar_t), and then it was copied into the BSTR using SysAllocString(). This extra step is required to properly initialize the character count in the BSTR. You free the string with SysFreeString.

MIDL strings are extremely simple to define. That's not to say they are easy to use. Passing strings and arrays across a COM interface can be frustrating. COM needs a lot of information about parameter lengths before they can be transmitted. When working with strings you need to pay particular attention to attributes.

Arrays

In many ways our discussion of strings covers the important issues concerning arrays; strings are a specialization of arrays. COM allows four basic types of arrays: Fixed, Conformant, vary-

ing, and open. Arrays can be multi-dimensional, and have a set of special attributes associated with them.

Attribute	Usage
Size_is(n)	Specifies the number of elements in an array. This attribute is used to determine array size at run time.
Max_is(n)	Specifies the maximum index size of an array. The maximum index value for the parameter is n-1.
Length_is(n)	Determines the number of array elements to be transmitted. This is not necessarily the same as the array size. Cannot be used with last_is.
First_is(n)	Determines the first array element to be transmitted.
Last_is(n)	Determines the last array element to be transmitted.

Table 8.5 COM Array Attributes

MIDL DEFINITION

```
HRESULT TestFixed([in,out] long lGrid[10]);
HRESULT TestConf([in] long lSize,
                 [in,out,size_is(lSize)] long *lGrid);
```

SERVER CODE

```
STDMETHODIMP CArrayTypes::TestFixed(long lGrid[10])
{
  for( int i=0; i<10; i++)
    lGrid[i] = lGrid[i] + 10;
  return S_OK;
}

STDMETHODIMP CArrayTypes::TestConf(long lSize, long *
  lGrid)
{
  for( int i=0; i<LSIZE;
```

CLIENT CODE

```
long arr[10]={0,1,2,3,4,5,6,7,8,9};
  pI->TestFixed( arr );
  pI->TestConf( 10, arr );
```

Varying arrays, and their close cousin open arrays allow the interface designer to have even more control over how data is transferred. By using the first_is, length_is, and last_is attributes, you can design the interface so that only modified data is marshaled and transmitted between the client and server. For large arrays, this makes it possible to transmit only those elements of an array that have been changed.

Structures and Enumerations

MIDL structures are almost identical to "C" language structures. Unlike the C++ struct, the MIDL type cannot contain methods - it's limited to data. A MIDL struct is similar to "C" language structs, which don't allow methods either. Here's a typical MIDL structure definition:

```
typedef struct
{
  long lval;
  double dval;
  short sval;
  BYTE bval;
} TestStruct;
```

The typedef and struct declarations work very similarly to their "C" counterpart. This is not to say it is an exact analog; like most of MIDL, it uses a limited subset of data types. You also can't use a struct in oleautomation or dual interfaces.

You can define enumerated data types with a typedef statement:

```
typedef enum { Red, Blue, Green } RGB_ENUM;
```

Values start at 0 and are incremented from there. You can also explicitly assign values to enum's. MIDL handles enum's as unsigned shorts. This is incompatible with C++ which uses signed int's. If you want to be compatible with C++ use the v1_enum attribute.

Here's an example using the standard MIDL 8 bit enumeration type. Note that we're giving the enumeration's explicit values - that's not required.

MIDL DEFINITION

```
typedef enum {Red = 0,Green = 1,Blue = 2} RGB_ENUM;

HRESULT EnumTest([in]  RGB_ENUM e, [out]  RGB_ENUM
  *pe);
```

SERVER CODE

```
STDMETHODIMP CBasicTypes::EnumTest(RGB_ENUM e,
  RGB_ENUM *pe)
{
  if (pe == NULL) return E_POINTER;
  *pe = e;
  return S_OK;
}
```

CLIENT USAGE

```
RGB_ENUM r1, r2;
r1 = Blue;
pI->EnumTest( r1, &r2 );
```

Note that the client knew about the definition of RGB_ENUM. MIDL will generate a definition of the enumeration in the project header, as well as the type library.

Summary

Much of the real "COM" programming you do will deal with calling methods and passing parameters. This task is a lot trickier than it first appears. In C++ this task can be almost seem trivial, but for components there is a lot you need to know. Fortunately, MIDL gives us tremendous control over this process. We've tried to give examples of all the most common data types and attributes.

N I N E

OLE Automation and Dual Interfaces

There's been a debate going on for years now about the merits of Visual Basic (VB) and C++. The C++ programmers insist that their tool offers the most powerful and efficient method of developing Windows software. VB developers insist their tool is an easier and quicker way to develop applications. It's undeniable that Visual Basic is becoming ubiquitous - it's integrated in spreadsheets, word processors, databases, and even Microsoft Developer Studio.

Add to this mix developers writing applications with Java, Delphi and scripting languages. Developers using all these tools need to access your COM components.

To accomplish this cross-platform integration we're going to have to deal with IDispatch and the topic formerly known as "OLE Automation". Microsoft now prefers to call it ActiveX and COM, but OLE is lurking just under the surface. OLE itself is a big topic, and there is a plethora of books on the subject. To keep things manageable, we'll concentrate on the basic COM methods required to communicate with automation clients.

Because you're reading this book, I can assume that you already know a lot about C++ programming. C++ is the native language of Windows, and offers an efficient and powerful way to access the Windows API. Regardless of its other merits, C++ is

an excellent tool for developing powerful server applications. To access these servers, COM is the communication method of choice.

Visual Basic, Java, and scripting languages all share one design limitation. They don't offer native support of pointers. This limitation presents a fundamental incompatibility; COM is built around the concept of a VTABLE, which is really an array of function pointers. A client also needs to know the exact VTABLE layout of the server to call its methods. In C++ we use a header file, but it's only useful to other C++ programs. Obviously, we're going to have to do things a little differently to make COM objects available to a language like Visual Basic.

There are two ways to use COM through a non-pointer language. The traditional method involves accessing COM through a special well-known interface called IDispatch. If the language itself knows how to use this interface, it automatically converts its syntax into calls to IDispatch. The more recent (and efficient) alternative is through a facility called type libraries. Type libraries provide detailed information on the COM interface, allowing the language to handle the details of calling the interface.

Both of these techniques have advantages and disadvantages. In this chapter I will describe IDispatch and an alternative called dual interfaces. I'm going to present the client code in this chapter in Visual Basic. These examples were developed using Visual Basic version 6.0.

IDL Definitions

In MIDL, IDispatch interfaces are referred to with the [dual] and [oleautomation] attributes. These interfaces must inherit from the standard interface IDispatch.

```
[
  object,
  uuid(F7ADBF5B-8BCA-11D1-8155-000000000000),
  dual
]
interface IVbTest : IDispatch
```

This is a dual interface, which implements IDispatch through interface inheritance.

The IDispatch Interface

IDispatch is a special interface that is used to locate and call COM methods without knowing their definition. The way COM accomplishes this feat is quite complex, but extremely flexible. There are two ways to use IDispatch from a VB program: late binding and early binding. First let's look at late binding.

IDispatch has four methods. Because the interface inherits from IUnknown (as all COM interfaces must), it also gets the required QueryInterface, AddRef, and Release. Remember that COM interfaces are immutable. This means all IDispatch interfaces MUST define these methods.

WHAT IS BINDING?.....

Binding refers to how a compiler or interpreter resolves references to COM objects.

Late Binding • The client (Visual Basic) interpreter waits until the COM method is called before it checks the method. First, the object is queried for the ID of the method. Once this is determined, it calls the method through Invoke(). This behavior is automatic in Visual Basic and most interpreted (and macro) languages. If you want to do this in C++, it will take some programming. Late binding is very slow, but very flexible.

Early Binding • The interpreter or compiler checks the method before the program runs. All the Dispatch ID's are determined beforehand. The method is called through Invoke(). Early binding is much faster than late binding. It is also catches errors sooner.

Very Early Binding • This is also known as Virtual Function Table, or VTABLE, binding. The client calls the COM methods directly without using Invoke(). A type library or header file is required. This is how almost all C++ programs access COM objects. This is the fastest type of binding.

ID Binding • This can be thought as "manual" early binding. The programmer hard-codes, or caches, all the DISPID's and calls Invoke() directly.

IDispatch method	Description
GetTypeInfoCount	Returns the availability of type information. Returns 0 if none, 1 if type information is available.
GetTypeInfo	Retrieves type information about the interface, if available.
GetIDsOfNames	Takes a method or property name, and returns a DISPID.
Invoke	Calls methods on the interface using the DISPID as an identifier.

Table 9.1 Methods of Dispatch

COM-compatible languages have built-in support for the IDispatch interface. Here's an example from VB:

```
Dim Testobj As Object
Set Testobj = CreateObject("VbTest.VbTest.1")
Testobj.Beep (1000)
Set testobj = Nothing
```

This example isn't really very different from the way we do it in C++ using the #import directive. What is going on behind the scenes is quite a bit different.

The first statement creates a generic object. A VB "Object" is generic and can contain any Visual Basic type, including a COM object. This is accomplished by using the VARIANT type, which we'll discuss later in this chapter. We use the CreateObject function to look up the object in the registry and attach to the server. The string passed into CreateObject is the ProgID of the server. In this case, “VbTest.VbTest.1” is a name we chose to give our test server (more details on naming below). If the server wasn't properly registered or we typed in the wrong name, the CreateObject call will fail with a run-time error.

The call to the Beep method is straightforward to write, but the Basic interpreter has to do a lot of processing to make it happen. Here's a summary of the steps:

1. Get the DISPID of the COM method named Beep
2. Build a parameter list.
3. Call Invoke, passing the DISPID and parameter list.

VB calls the GetIDsOfNames function, passing in the name of the function. In this case, the function name is the string "Beep". GetIDsOfNames will look up the name and return the dispatch ID. Where does this ID come from? Let's look at the IDL definition of the method:

```
[id(7), helpstring("method Beep")]
  HRESULT Beep([in] long lDuration);
```

GetIDsOfNames will look up the string "Beep" internally and return a DISPID of "7". How does it find the DISPID? That depends on how the IDispatch interface is created. In our examples we are using an interface created with ATL. ATL uses the class IDispatchImpl, which looks it up in the type library. When the MIDL compiler created the type library it included a map of all the function names and their DISPID's.

In the OLE world there is a whole set of pre-defined DISPID's. These ID's have been mapped to standard properties of objects, such as fill color, text, and font. These pre-defined DISPID's all have negative numbers, and are defined in OLECTL.H.

Once VB has the DISPID, it needs to build a list of parameters to send to Beep. In this case, there is only one parameter: an [in]long parameter. The problem here is that VB doesn't know how many parameters Beep takes, or what type they are. Building parameter lists is actually a pretty complex operation. (If we use a type library, VB can get this information directly. See the section on early binding).

A parameter list is contained in a Dispatch Parameter, or DISPPARAMS structure. This structure is defined in OAIDL.IDL as follows:

```
typedef struct tagDISPPARAMS {
    [size_is(cArgs)] VARIANTARG * rgvarg;
    [size_is(cNamedArgs)] DISPID * rgdispidNamedArgs;
    UINT cArgs;
    UINT cNamedArgs;
} DISPPARAMS;
```

If you would rather see the C++ header, it's in OAIDL.H and looks like this:

```
typedef struct  tagDISPPARAMS
    {
    VARIANTARG  *rgvarg;
    DISPID  *rgdispidNamedArgs;
    UINT cArgs;
    UINT cNamedArgs;
    }DISPPARAMS;
```

The parameter list is packed up in a DISPPARAMS structure, each argument added to the VARIANTARG structure. As each argument is added to the structure, the counter (cArgs) is incremented. The member "*rgvarg" is essentially an array of VARIANTARG structures.

OLE allows the use of what are called "Named" arguments. These arguments are identified by a nametag, and may be passed in any order. The handling of named arguments is taken care of by the implementation of Invoke. We're not going to be using named arguments here, but be aware of the possibility.

A VARIANTARG is a VARIANT, which is a gigantic union of different data types. Here is an edited version of the definition from OAIDL.IDL. If you look in OAIDL.H, you'll see that it is created by running OAIDL.IDL through the MIDL compiler.

```
//VARIANT STRUCTURE
typedef VARIANT VARIANTARG;

struct tagVARIANT {
  union {
    struct __tagVARIANT {
      VARTYPE vt;
```

```
  WORD    wReserved1;
WORD    wReserved2;
WORD    wReserved3;
union {
    LONG          lVal;       /*VT_I4*/
    BYTE          bVal;       /*VT_UI1*/
    SHORT         iVal;       /*VT_I2*/
    FLOAT         fltVal;     /*VT_R4*/
    DOUBLE        dblVal;     /*VT_R8*/
    VARIANT_BOOL  boolVal;    /*VT_BOOL*/
   _VARIANT_BOOL bool;       /*(obsolete)*/
    SCODE         scode;      /*VT_ERROR */
    CY            cyVal;      /*VT_CY*/
    DATE          date;       /*VT_DATE*/
    BSTR          bstrVal;    /*VT_BSTR*/
    IUnknown *    punkVal;    /*VT_UNKNOWN*/
    IDispatch *   pdispVal;   /*VT_DISPATCH*/
    SAFEARRAY *   parray;     /*VT_ARRAY*/
    BYTE *        pbVal;      /*VT_BYREF|VT_UI1*/
    SHORT *       piVal;      /*VT_BYREF|VT_I2*/
    LONG *        plVal;      /*VT_BYREF|VT_I4*/
    FLOAT *       pfltVal;    /*VT_BYREF|VT_R4*/
    DOUBLE *      pdblVal;    /*VT_BYREF|VT_R8*/
    VARIANT_BOOL *pboolVal;   /*VT_BYREF|VT_BOOL*/
    _VARIANT_BOOL *pbool;     /*(obsolete)*/
    SCODE *       pscode;     /*VT_BYREF|VT_ERROR*/
    CY *          pcyVal;     /*VT_BYREF|VT_CY*/
    DATE *        pdate;      /*VT_BYREF|VT_DATE*/
    BSTR *        pbstrVal;   /*VT_BYREF|VT_BSTR*/
   IUnknown **   ppunkVal;   /*VT_BYREF|VT_UNKNOWN*/
    IDispatch **  ppdispVal;  /
*VT_BYREF|VT_DISPATCH*/
    SAFEARRAY **  pparray;    /*VT_BYREF|VT_ARRAY*/
   VARIANT *     pvarVal;    /*VT_BYREF|VT_VARIANT*/
    PVOID         byref;      /*Generic ByRef*/
    CHAR          cVal;       /*VT_I1*/
    USHORT        uiVal;      /*VT_UI2*/
    ULONG         ulVal;      /*VT_UI4*/
    INT           intVal;     /*VT_INT*/
    UINT          uintVal;    /*VT_UINT*/
   DECIMAL *     pdecVal;    /*VT_BYREF|VT_DECIMAL*/
```

```
        CHAR *          pcVal;      /*VT_BYREF|VT_I1*/
        USHORT *        puiVal;     /*VT_BYREF|VT_UI2*/
        ULONG *         pulVal;     /*VT_BYREF|VT_UI4*/
        INT *           pintVal;    /*VT_BYREF|VT_INT*/
        UINT *          puintVal;   /*VT_BYREF|VT_UINT */
        } __VARIANT_NAME_3;
      } __VARIANT_NAME_2;
      DECIMAL decVal;
  } __VARIANT_NAME_1;
  };
```

The actual variable stored will will correspond with its data type. The actual type of the variable is stored in the VT member. Each of the VT types is #defined to a number, for example VT_I4 has a value of 3. Because it's a union, the size of a VARIANT is at least the size of its largest member. The types allowed in a VARIANT are the only types you can pass to and from an IDispatch interface.

Here's how you would put a long value into a VARIANT. You should be able to generalize this code to any of the types defined in the structure.

```
VARIANT v;
VariantInit(&v);
v.vt = VT_I4;
vt.lVal = 100;
```

Variants are somewhat ungainly structures to work with in C++. Variants have their origin in Visual Basic with its notion of changeable data types and automatic type conversion. Traditionally Basic hasn't been a typed language, and Variants were used to store all variables. One of the strengths of C++ is strong type checking, so Variants are antithetical to good C++ programming practice.

All parameters passed to Invoke will be packaged in this VARIANTARG structure. The next step in calling a method through IDispatch is the Invoke function.

Using Invoke

An IDispatch interface calls its functions through the Invoke() method. Generally, the client programmer doesn't call any of the IDispatch methods directly. Visual Basic hides all its methods, including Invoke. If you need to call IDispatch methods from a C++ client, you're a lot better off going directly through a VTABLE. If you don't have a dual interface, building the parameter lists and calling Invoke is going to be a laborious task. We'll look at the Invoke method, even though I hope you won't have to use it directly.

Here is the definition of Invoke from OAIDL.IDL.

```
HRESULT Invoke(
            [in] DISPID dispIdMember,
            [in] REFIID riid,
            [in] LCID lcid,
            [in] WORD wFlags,
            [in, out] DISPPARAMS * pDispParams,
            [out] VARIANT * pVarResult,
            [out] EXCEPINFO * pExcepInfo,
            [out] UINT * puArgErr
        );
```

The DISPID of the requested function is given in the first parameter. This tells Invoke which method to call. The requested method's parameter list is passed in the "pDispParams" argument.

What happens if you call a method with an invalid parameter? The Basic interpreter won't catch your error, because it doesn't know enough to do so. The error is caught at run-time by the COM server itself. One of the functions of Invoke() is to check parameter lists. Invoke will try to convert the incorrect parameter if possible, and if not, it will return an error status. For example, if you called the Beep() method in Visual Basic with a string like this:

```
Testobj.Beep ("1000") ' ok
Testobj.Beep ("Hello" ) ' run-time error
```

Invoke() would convert the string "1000" into a number, and everything would work fine. When, however, you use a non-numeric string like "Hello", Invoke() doesn't know how to make the conversion. The function will fail with a VB run-time error 13, for "type mismatch". This ability to convert numbers can be dangerous. If you accidentally reverse the order of parameters, Invoke may be able to convert them anyway - giving unexpected results.

The status of a function is returned in three ways: 1) through the HRESULT, 2) the pExcepInfo structure, and 3) in the pVarResult argument. Like all COM methods, severe failure will be returned as an HRESULT. Visual Basic doesn't use the return codes - the closest equivalent is the Err object. Invoke returns its error information in an EXECPINFO structure.

```
typedef struct tagEXCEPINFO {
WORD  wCode;           /*An error code*/
WORD  wReserved;
BSTR  bstrSource;      /*A source of the exception */
BSTR  bstrDescription; /*A description of the error */
BSTR  bstrHelpFile;
    /*Fully qualified drive, path, and file name*/
DWORD dwHelpContext;
    /*help context of topic within the help file */
ULONG pvReserved;
ULONG pfnDeferredFillIn;
SCODE scode;
} EXCEPINFO;
```

This structure looks amazingly like the Visual Basic "Err" object. Table 9.2 shows the properties of that object.

The third type of returned data is in the pVarResult parameter of Invoke(). This data contains a user defined return value. What gets placed in here is determined by the [retval] attribute in the IDL code of the interface. Any parameter marked with [retval] is stuffed into a VARIANT and returned here. We'll see more of this when we look at property "get" functions.

Err Property	Description
Number	An error code for the error. This is the default property for the object.
Source	Name of the current Visual Basic project.
Description	A string corresponding to the return of the Error function for the specified Number, if this string exists. If the string doesn't exist, Description contains "Application defined or object defined error."
HelpFile	The fully qualified drive, path, and file name of the Visual Basic Help file.
HelpContext	The Visual Basic Help file context ID for the error corresponding to the Number property.
LastDLLError	On 32-bit Microsoft Windows operating systems only, contains the system error code for the last call to a dynamic link library (DLL). The LastDLLError property is read only.

Table 9.2 Error Properties in VB

If you need to work directly with Variants in C++ you should use the ATL CComVariant or “__variant_t” classes. There are also API-level functions, all starting with "Variant". Variants are very good at converting between data types. You can even make use of variants as a quick-and-dirty method of converting data types.

A 'pure' IDispatch interface is only required to implement IUnknown, GetTypeInfoCount, GetTypeInfo, GetIDsOfNames, and Invoke. Note that the methods called by Invoke don't have to be COM methods. They can be implemented any way the programmer wants because they aren't called by COM directly. Using these four methods, you can write a sever to do almost anything.

As you can see, there is a lot of processing required to call a method through IDispatch. It all happens behind the scenes in VB so you are not aware of it, but it does take time. All that pro-

cessing time means one thing: slow. There has to be a better way, and there is - type libraries and Early binding.

Using Type Libraries for Early Binding

Most clients don't need to use a pure IDispatch interface. In VB for example, you have extensive access to type information through type libraries. The type library defines a number of very important items:

- Interfaces exposed by a server
- Methods of an interface
- Dispatch ID's of methods
- Parameter list for methods
- GUIDs
- Data structures
- Marshaling information

The type library provides a complete description of the server's methods and capabilities. Using this information, the VB interpreter can provide more efficient access to the server.

Here's how we would write our VB interface using the type library. First, you have to turn on object browsing for the VbTest object. Do this in the Tools/References... menu item of the VB editor. Find the server object in the list and check it on the list. This causes Visual Basic to open the type library for browsing.

Warning - you won't be able to build the C++ server while VB has an open reference to it. If you need to make any changes, turn off browsing or close the project and rebuild your C++ project.

We will only make two changes in the VB code. Instead of creating a generic object, we create a specific COM Object. VB will find the definition among its open references.

First, we create the object with a specific object type. In the VB world this is also known as "hard typing." As objects become more common in the VB world, the practice is becoming more common.

```
Dim Testobj As VbTest
```

"VbTest" is a name we assigned to our test server (more details on naming below). When we create the object we can specify it more concisely, in a format C++ programmers will find reassuringly familiar:

```
Set Testobj = New VbTest 'early binding
```

Syntactically early binding isn't very different. All the other references to the object will remain unchanged. (In our four-line program this isn't remarkable).

The real difference is in how VB calls the methods of the Testobj object. It no longer needs to call GetIDsOfNames() to get the DISPID of the method. Visual Basic is using the Type library to get information about the Beep() method. It can find out about the required parameters and build a parameter list. It can also do all this before it calls the object, and without the overhead of communicating remotely with the object.

Here's what's most interesting: VB isn't using Invoke to call Beep() anymore. The call goes directly through the COM VTABLE! VB is figuring out the pointers of the VTABLE without the VB programmer even knowing they're using pointers.

Dual Interfaces

Dual interfaces can be called both through VTABLE pointers and from interpreted languages via IDispatch. Actually, I've been using dual interfaces in my examples all along.

If you aren't familiar with old-style OLE IDispatch implementations, dual interfaces may seem trivial. If you're working with ATL, it's extremely easy to create a dual interface. Back in the old days with MFC-based COM, this wasn't the case - it took a considerable programming effort to build a dual interface. This approach would have required, for example, that we implement all the object's methods inside Invoke.

When an interface is called through Invoke(), it only needs to implement the four methods of IDispatch. If you specify the interface as dual, it implements these four methods, plus your custom methods. Here's what the VTABLE looks like for a dual interface:

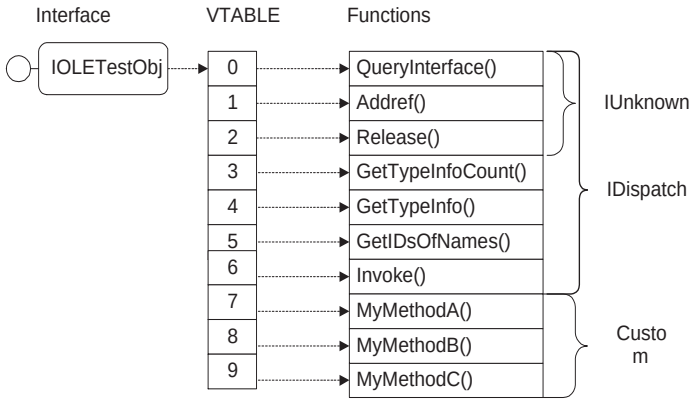

Figure 9–1 The VTABLE of a dual interface

When a dual interface method is called through Invoke, it looks up the DISPID and finds the location of the method in the VTABLE. It can then call the method through the VTABLE pointer. In this case, if the user requested MyMethodA, the client would call GetIdsOfNames and get the DISPID of that method - five. It would map the call to the VTABLE, and call MyMethodA(). An early binding client can skip all this and call MyMethodA directly, without using GetIDsOfNames or Invoke.

When you create a new COM object using the ATL Object Wizard, you can specify the dual attribute. This will cause the wizard to add the ATL IDispatch implementation to the object.

Figure 9–2 Specifying the dual attribute

ATL makes IDispatch available by using the template class IDispatchImpl<>.

```
class ATL_NO_VTABLE CVbTest :
  public CComObjectRootEx,
  public CComCoClass<CVBTEST, &CLSID_VbTest,
  public ISupportErrorInfo,
  public IDispatchImpl<IVBTEST, &LIBID_CH6Lib
  &IID_IVbTest,
...
```

Note that one of the parameters to IDispatchImpl is the GUID of a type library. This is the id of the type library defined in the library statement of the IDL file. IDispatch interfaces and type libraries are very closely tied together. The following line is added to the COM MAP and, voila! Instant dual interface.

```
COM_INTERFACE_ENTRY(IDispatch)
```

For the amount of functionality this adds, it's remarkably easy. The IDispatchImpl class will handle all the calls to GetID-

sOfNames, Invoke, and the other IDispatch methods. The only cost of the dual interface is the limitation of using variant compatible types.

There is no Proxy/Stub DLL for Dispatch Interfaces

One of the nice things about IDispatch-based interfaces is that they have built-in marshaling. The type library is an inherent part of working with dispatch interfaces, and the type library contains all the information necessary to transfer data between processes. This is possible because dispatch interfaces only allow a very limited subset of data types - those that can be stored in a VARIANT. Standard marshaling knows how to handle all of these types.

Properties

One of the conventions of the Visual Basic programming model is to describe objects and controls with properties. The VB browser presents property tabs for almost every type of control. Here's a typical property put and get implementation in Visual Basic. The property name is "LongValue".

```
Dim lval As Long

testobj.LongValue = lval‘ property put
lval = testobj.LongValue‘ property get
```

The closest equivalent in the C++ would be the public member variables in a class. Although member variables can be used like properties, it is considered bad Object Oriented form to do so.

COM interfaces that do not have public data members, just methods. It wouldn't make sense to expose data members for remote clients - there is no way for a client to directly manipulate the data on a server. In-process severs are DLL's, which can

export data members. COM however, does not allow this - COM interfaces need to be compatible with both DLL and remote use.

This means our COM objects are going to have to simulate properties via methods. For oleautomation and dual interfaces, there are four attributes used to describe properties:

Attribute	Usage
Propget	A property-get function. Used by IDispatch clients to get a single value from the COM object. The method must have as it's last parameter an [out] pointer type. E.G. ([out, retval] long *pdata)
Propput	The propput attribute specifies a property-set function. Used by dispatch clients to set a single value of a COM object. The last parameter of the methods must have the [in] attribute. E.G. ([in] long data)
Propputref	Similar to a propput method, except the parameter is a reference with the [in] attribute. E.G. ([in] long *pdata)
Retval	Indicates that the parameter receives a return value for the method. The parameter must be an out parameter, and be a pointer. For propget the last parameter must be a retval.

Table 9.3 Property Attributes of Methods

Here's the definition of a property interface in IDL:

```
[propget, id(1), helpstring("property LongValue")]
  HRESULT LongValue([out, retval] long *pVal);
[propput, id(1), helpstring("property LongValue")]
  HRESULT LongValue([in] long newVal);
```

Notice that properties are methods. Both methods have the same name and the same dispatch ID. MIDL resolves these ambiguities using the propget and propput attributes. MIDL will generate the following function prototypes for this interface.

```
STDMETHOD(get_LongValue)(/*[out, retval]*/ long
  *pVal);
STDMETHOD(put_LongValue)(/*[in]*/ long newVal);
```

Notice that it took the member name and appended it to a "get_" or "put_". The function definitions will be as follows:

```
STDMETHODIMP CVbTest::get_LongValue(long * pVal)
{
  *pVal = m_Long ;
  return S_OK;
}

STDMETHODIMP CVbTest::put_LongValue(long newVal)
{
  m_Long = newVal;
  return S_OK;
}
```

In this example I'm using the methods to access the class member variable named m_Long. As COM methods go, these are pretty simple. There's nothing magic about these property functions. If you were calling them from a C++ client, you would use the full method name.

```
// C++ client example
long l;
hr = pI->get_LongVal(&l);
```

Adding Properties with the Class Wizard

The ATL wizards make adding properties to an interface extremely easy. If you use the "Add Properties" wizard, it's a no-brainer. First, the interface must be a **dual** or **oleautomation** derived interface. Next go to the ClassView tab and press the right mouse button. Choose the "Add Property..." selection.

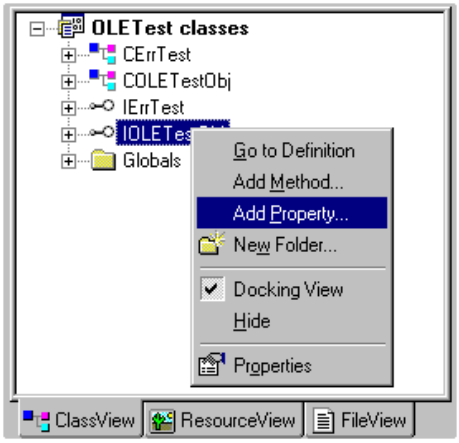

Figure 9–3 Adding properties

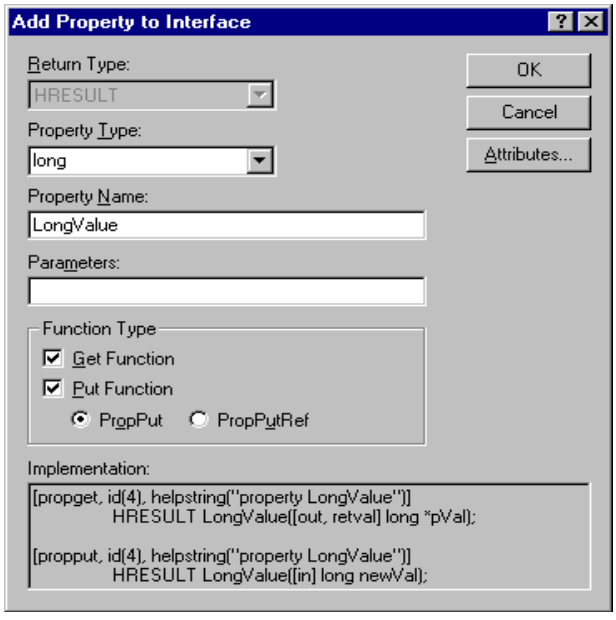

Figure 9–4 Specifying properties

The Add Properties dialog will be displayed. This dialog automatically creates property members for the interface. Note that we didn't enter any parameters for the property interface. This property interface has just one parameter, although you are allowed to have more. If you're going to have more parameters, pay attention to the rules for the **propget** and **propput** attributes.

Methods

You can call methods in an IDispatch interface just like any other COM interface. In the previous example we used the Beep() method. Here's the full example code:

MIDL DEFINITION

```
[id(7), helpstring("method Beep")]
  HRESULT Beep([in] long lDuration);
```

SERVER CODE

```
STDMETHODIMP CVbTest::Beep(long lDuration)
{
  ::Beep( 440, lDuration );// don't forget the ::
  return S_OK;
}
```

CLIENT USAGE

```
testobj.Beep (1000)          ' Visual Basic client
```

The ISupportErrorInfo Interface

HRESULTS provide the basis for all COM error handling, but automation clients are often able to get more detailed information through the ERR object. This extended capability is built into IDispatch, so it makes sense to build it into your server objects. You can add this functionality to Dispatch and Dual interfaces

(or Custom, for that matter) by checking the SupportErrorInfo box when you create your ATL object.

Figure 9–5 Adding error support

What this option does is add several interfaces to your ATL object. Here's the header code of an object that supports the ErrorInfo interface.

```
class ATL_NO_VTABLE CErrTest :
  public CComObjectRootEx,
  public CComCoClass<CERRTEST, &CLSID_ErrTest,
  public ISupportErrorInfo,
  public IDispatchImpl<IERRTEST, &LIBID_OLETESTLib
  &IID_IErrTest,
```

This extended error capability comes through both the ISupportErrorInfo interface and the CComCoClass template. The ISupportErrorInfo interface is added to the supported interfaces of the coclass through the COM map.

```
BEGIN_COM_MAP(CErrTest)
  COM_INTERFACE_ENTRY(IErrTest)
  COM_INTERFACE_ENTRY(IDispatch)
  COM_INTERFACE_ENTRY(ISupportErrorInfo)
END_COM_MAP()
```

The ATL wizard also adds source code to your CPP module. ISupportErrorInfo supports a single method called InterfaceSupportsErrorInfo. The method provides an array of interfaces that support extended errors. The first interface is automatically (IID_IErrTest) added to this list. If you add multiple interfaces to your coclass you'll need to add the IID's to this static array. Note that this code was entirely generated by the ATL wizard.

```
STDMETHODIMP CErrTest::InterfaceSupportsError-
  Info(REFIID riid)
{
  static const IID* arr[] =
  {
    &IID_IErrTest
  };
  for (int i=0; i < sizeof(arr) / sizeof(arr[0]); i++)
  {
    if (InlineIsEqualGUID(*arr[i],riid))
      return S_OK;
  }
  return S_FALSE;
}
```

To populate the ErrorInfo object, you can call the Error method of the CComCoClass template class. This method has a number of overloads, which allow you to set different types of error information. Here's one of the simplest ways to use it.

```
STDMETHODIMP CErrTest::Div(double d1, double d2, dou-
  ble *dresult)
{
  HRESULT hr = S_OK;
  if (d2 == 0.0)
  {
```

```
    wchar_t str[128] = L"Divide By Zero" ;
    Error( str, IID_IErrTest ); // member of CComCo-
  Class
    hr = E_FAIL;
  }
  else
    *dresult = d1 / d2;

  return hr;
}
```

Here's the IDL code that defines this method:

```
[id(1), helpstring("method Div")] HRESULT Div(
                    [in] double d1,
                    [in] double d2,
                    [out,retval] double *dresult);
```

The automation client can extract this information the usual way, using the ERR object. Even if your coclass doesn't implement ISupportErrorInfo, the VB ERR object does a pretty good job of filling itself with usable information. Here's a Visual Basic sample:

```
Private Sub DoCalc_Click()
    Dim ETObj As Object
    Dim v1 As Double, v2 As Double, v3 As Double

    Set ETObj = CreateObject("OLETest.ErrTest.1")
    v1 = Me.D1
    v2 = Me.D2

    On Error GoTo ShowProb
    v3 = ETObj.Div(v1, v2)
    Me.Result = v3

    Set ETObj = Nothing
    Exit Sub

ShowProb:
```

```
Dim msg As String
    msg = "Description:" + Err.Description + Chr(13)+_
          "Source:" + Err.Source + Chr(13) + _
          "Number:" + Hex(Err.Number)
    MsgBox msg, vbOKOnly, "Server Error"
    Set ETObj = Nothing

End Sub
```

The Visual Basic "ERR" object has a number of useful properties - these include "Description", "HelpContext", "HelpFile", "Number", and "Source". All these properties can be set with Error method.

This error information is getting stored in a structure called EXCEPINFO. Here's the layout of this structure from OAIDL.IDL. You can immediately see the similarities between this and the "ERR" object.

```
typedef struct tagEXCEPINFO {
    WORD  wCode;           /* An error code describing
  the error. */
    WORD  wReserved;
    BSTR  bstrSource;       /* A source of the excep-
  tion */
    BSTR  bstrDescription;  /* A description of the
  error */
    BSTR  bstrHelpFile;     /* Fully qualified drive,
  path, and file name */
    DWORD dwHelpContext;    /* help context of topic
  within the help file */
    ULONG pvReserved;
    ULONG pfnDeferredFillIn;
    SCODE scode;
} EXCEPINFO;
```

Obviously, this structure is getting filled by CComCoClass::Error. The information in the structure is passed back through an interface called IErrorInfo. There's also an interface called ICreateErrorInfo that sets the EXECPINFO structure. If

you're accessing the coclass through C++, you can use these two interfaces directly.

All of this error handling comes standard with an IDispatch interface. The EXECPINFO structure is one of the parameters to IDispatch::Invoke(). The extended error interfaces provide a good way to pass detailed information back to a client program.

Summary

Here are a few important points from the discussion above:

- Your components are going to need to communicate with Visual Basic, Java, Scripting languages, and a whole slew of other applications. These applications will probably support IDispatch based OLE interfaces.
- With ATL it's easy to implement IDispatch and dual interfaces.
- Dual and IDispatch interfaces can only use data types allowed in variants.
- The earlier the binding, the faster the interface. Use hard typing on the client for maximum performance.
- Type libraries provide extensive information to the client application. IDispatch interfaces can use the type library to marshal data.

T E N

COM Threading Models

One of the more esoteric aspects of COM is the naming and usage of the different threading models. The naming alone is designed to bewilder: a COM object can be “single threaded”, “apartment threaded”, “free threaded”, and “both.” Each of these models deals with a different set of synchronization and threading issues.

The default "apartment" threading model works quite well for almost all applications. Apartment threading allows COM programmers to effectively ignore threading issues in most cases. There are however, some cases where design and performance problems conspire to require a more precise level of control. In this chapter we will explore the different options so that you can make intelligent threading decisions.

Synchronization and Marshaling

If you've worked with multi-threaded applications before, then you're well aware of the complexities involved. The difficulty is synchronization - ensuring that things happen in the correct order. When COM objects communicate with clients and each other, they face a variety of synchronization issues. COM objects

can run in in-process, out-of-process, and remote servers - each of these has its own unique set of constraints.

COM defines a set of “models” for dealing with threading issues. By following these models, we can ensure synchronized communication. By the use of the word “model”, we can assume that things aren't going to be completely automatic. The proper implementation of threading, especially free threading, is going to take some knowledge on the part of the developer.

Marshaling is the process of packaging data and sending it from one place to another. COM handles all data transfer through method calls. COM marshaling involves the packaging of data in parameter lists. Marshaling can take place at many levels. At its simplest level, it can mean the transfer of data between individual threads. More complex marshaling may involve sending data between different processes, or even across a network. Marshaling is one of the important factors in synchronization.

SOME QUICK DEFINITIONS.....

Process • An application or program running on the system. A process has its own separate address space, program code, and (heap) allocated data. A process can contain one or more threads of execution.

Thread • A piece of code that can be run by a single CPU. Each thread has its own stack and program registers and is managed by the operating system. All threads in an application share the same address space and global data, as well as other resources. Multiple threads can be run independently of each other and the operating system schedules their execution. On multi-CPU computers, more than one thread may be running simultaneously.

Fibers • A type of 'lightweight' thread. Similar to threads except that they have to be manually scheduled for execution. Not commonly used. Currently there is no COM analog to fibers.

Thread local storage • In general, threads share memory with the rest of their process. There is a special type of memory called thread local storage (TLS), that is only available to the thread that creates it. TLS is accessed through the TLS API, which includes the functions TlsAlloc, TlsGetValue, TlsSetValue, and TlsFree

Threading Models

COM servers have two general threading models. This includes *Apartment Threaded* and *Free Threaded* objects. To understand what these threading models mean, let's first look at the way Windows handles threads.

In the world of Windows programming, we customarily deal with two types of threads - *user interface* threads and *worker threads*. The distinction between the two deals primarily with the existence of a message loop.

Worker threads are the simplest type of thread. A worker thread does not have to handle user input. It starts at the beginning and continues until it's finished processing. A typical worker thread would be used to compute a value, or to perform database access, or to do anything that takes a substantial amount of time to complete.

If worker threads do communicate with other threads, it is through mutexes, events, semaphores, and other synchronization or IPC methods. Worker threads are very useful for long computational tasks which don't need any user interaction. A typical program would create a worker thread by calling CreateThread. The worker thread has a main function, called a THREADPROC, and can have subsidiary functions as well called by the THREADPROC. The THREADPROC executes as if it were an independent program. The thread is given a block of data to work on, and it churns until it's done. When the worker thread is complete, it usually signals its caller and exits.

By contrast, all interactive Windows programs run as 'user interface threads'. A user interface thread has the special constraint that it must always be responsive to user input. Whenever the user resizes a window, or clicks the minimize button, the program must respond. Regardless of what it's doing, it must be able to redraw itself when it's window is uncovered or moved. This means that the program must break it's processing up into small manageable chunks which complete quickly and allow the program to respond.

User Interface threads have a message loop to process user and system events. All windows programs are based on the con-

cept of the message loop. Each window has a user interface thread that monitors the message queue and processes the messages it receives. A typical message loop looks like this:

```
MSG msg;
while(GetMessage(&msg,NULL,0,0))
{
    TranslateMessage( &msg );
    DispatchMessage( &msg );
}
```

Windows uses this method to ensure that a program processes its input in a sequential manner. Messages are always processed in the order they are received. Each window has only a single thread, and this thread must handle all user-input events. When the program receives a message, such as a "Close Window" (WM_CLOSE) message, it dispatches the message to a specific function.

The message loop continues until it receives a WM_QUIT message. Notice that the 'while' statement in the message loop tests the return value of GetMessage. The WM_QUIT message causes GetMessage to return FALSE, which exits the loop. After the message loop is finished, the thread typically shuts itself down and closes all its windows.

The Windows operating system handles the overall routing of messages. It handles the hardware generation of events such as mouse moves, and ensures that each window receives the appropriate messages.

The big advantage of a User Interface thread is that it breaks up its processing into a series of compact functions, each of which handles a specific message. For example, the message might be WM_CLOSE, and it would be sent to the function called OnClose(). When the application is finished processing a message, it returns to the message loop and responds to the next message. Because messages are queued up as they arrive, the UI thread is never processing more than one message at a time. This is a very safe, but somewhat inefficient, method of processing requests while still remaining responsive to input.

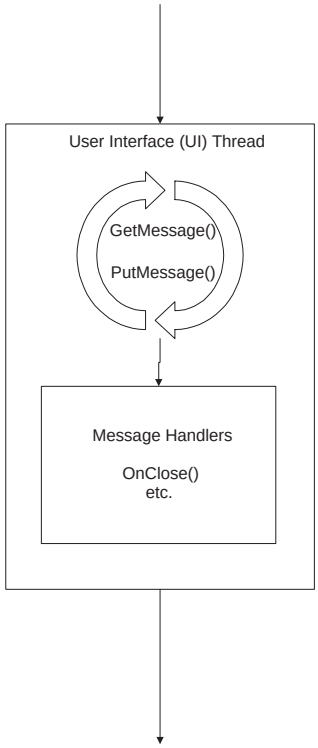

Figure 10–1 User Interface thread model

Apartment, Free, and Single Threads

In the COM world we use a different terminology to describe threads: User Interface threads are called "Apartment" threads. Worker threads are called "Free" threads. Although not identical to their Win32 counterparts, there are quite a few similarities. The third type of thread is a 'single' thread. Single threads are really just a special type of apartment threads.

Actually, the terminology is somewhat more confusing. You'll often see apartment threads called 'single threaded apartments', or STAs. The 'free' threaded counterpart is commonly called a 'multi-threaded apartment', or MTA. While this it may be

technically accurate, I avoid the STA and MTA nomenclature because it is more confusing. I'll use Apartment, single, and free threading in my examples.

The term "apartment" is purely conceptual. The apartment is the environment that the thread "lives in" - it separates the thread from the rest of a process. In many ways, the threading model is really a set of rules describing how the thread will behave.

The ATL Wizard and Threading Models

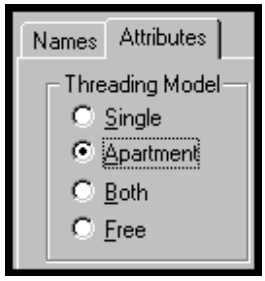

Figure 10–2 ATL Threading Models

The ATL object wizard makes it extremely easy to define an object's threading model. In a Win32 environment, the two options **Single** and **Apartment** behave similarly. Although each of these options represents real concepts, they don't generate different code under ATL. The designation is really just a flag for the COM subsystem. The COM subsystem will use these values to determine how to marshal calls between threads. This behavior is not reflected anywhere in the source code of the coclass or server. The **Both** and **Free** options are similarly identical.

COM determines threading model in two different ways, depending on the type of server. For out-of process (EXE) servers, the threading model is set when you initialize COM. You specify the model by the call to **CoInitialize** and **CoInitial-**

izeEx. Let's look at the extended version of the initialization routine.

```
HRESULT CoInitializeEx(
  void * pvReserved, //Reserved, always 0
  DWORD dwCoInit );//COINIT value - threading model
```

The second parameter specifies the threading model. The dwCoInit argument is a COINIT enumeration, which is described in the <objbase.h> header. The COINIT enumeration determines the threading model. There are several other values of the enumeration, but they aren't commonly used.

COINIT enumeration	Value	Description
COINIT_APARTMENTTHREADED	2	Initializes the thread for *apartment-threaded* object concurrency.
COINIT_MULTITHREADED	0	Initializes the thread for *multithreaded* object concurrency.

Table 10.1 COINIT enumerations used by CoInitializeEx

When you call the default version of CoInitialize(0), it is the same of specifying COINIT_APARTMENTTHREADED. CoInitialize remains for compatibility reasons, but the extended version (CoInitializeEx) is the recommended form.

The behavior of a threading model is often determined by server implementation. A remote (EXE) server behaves differently from an In-process server (DLL). COM looks at the threading modes of the client and server, and determines how to access the object.

In-process servers don't always call CoInitialize() for each COM object. COM needs a way to determine the threading requirements of the object. This is accomplished by using a registry key that describes the COM object. This registry key determines the threading model of the in-process object.

Under HKEY_CLASSES_ROOT\CLSID, each in-process server can have a key named InprocServer32. Under this key

there is a named valued called "ThreadingModel". The threading model can be "Single", "Apartment", "Free", or "Both".

Apartment Threads

In any threading diagram, an “apartment” is the box around the message loop and the COM object’s methods. The apartment is a logical entity that does not map directly to either threads or memory. The apartment defines the context of the executing COM object and how it handles multi-threading issues.

An apartment thread has a message loop and a hidden window. Whenever COM needs to communicate with an apartment threaded object, it posts a message to the object’s hidden window. The message gets placed in the object’s message queue. This is done with PostThreadMessage().

The message queue ensures that all COM requests are processed sequentially. This means that if you are programming an apartment threaded server, you don't have to worry about threading issues. Each method call waits until the previous call is 100% completed with its processing. The COM subsystem automatically takes care of posting messages in the correct thread.

Regardless of what thread a client uses to access an apartment threaded object, the processing will be done on a single designated thread. This is true even if the client and server are running in the same process (as in an In-Process server).

When you create the server, you specify "Threading Model: Apartment" in the ATL Object Wizard. The apartment threaded object is implemented through the ATL template class <CComSingleThreadModel>. Here is the header generated by the ATL object wizard.

```
class ATL_NO_VTABLE CMyObject :
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<CMyObject, &CLSID_MyObject>,
  public IDispatchImpl<IMyObject, &IID_IMyObject,
  &LIBID_MYLib>
...
```

COM does all the work of setting up the object and its apartment and message loop. You can then write your COM object's code without worrying about thread safety. A COM server can have multiple apartment threaded objects running simultaneously. The limitation is that each object is always accessed through the same apartment thread. Always. If the server creates a new object, it creates a new apartment for that individual object.

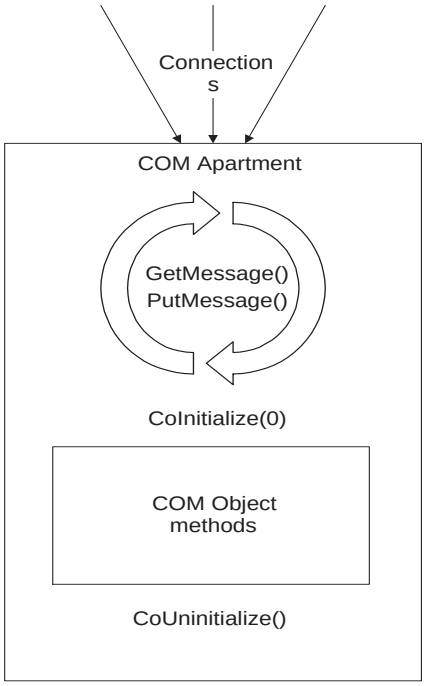

Figure 10–3 An Appartment Threaded COM Object

Single Threads

Single threaded servers have a specialized type of apartment model. A single threaded server has only a single apartment thread. That means that all COM objects are processed by the

same thread. When the server creates a new COM object, it reuses the one-and-only apartment thread to execute methods on that object. You can imagine the traffic jam this will cause on a busy server. Back in the early days of COM and OLE, this was the only model available.

Free Threaded Servers

Free threaded (or Multi-Threaded Apartment) servers in some cases offer advantages over Apartment threaded servers. Because they are not thread-safe, they can be much more complicated to code.

A free threaded server is similar to a worker thread. It has no message loop, and does nothing to ensure thread safety. We say the COM object has an 'Apartment', but in this case, the apartment does nothing to limit thread access.

When the client application makes a call to the object in a free thread, COM processes that request on the thread that called it. Remember that COM calls are often made through a Proxy/Stub. On the server side, the COM call arrives on the Stub's thread, and it will be processed on the same thread. If two clients call the same free threaded object, both threads will access the object at the same time. The potential for data corruption, or even server crashes, is very real if you do not manage synchronization properly. However, the problems encountered in a free threaded COM server are no different from those found in any multi-threaded application.

ATL implements a free threaded object with the ATL template class <CComMultiThreadModel>.

```
class ATL_NO_VTABLE CMyObject :
  public CComObjectRootEx<CComMultiThreadModel>,
  public CComCoClass<CMyObject, &CLSID_MyObject>,
  public IDispatchImpl<IMyObject, &IID_IMyObject,
  &LIBID_MYLib>
```

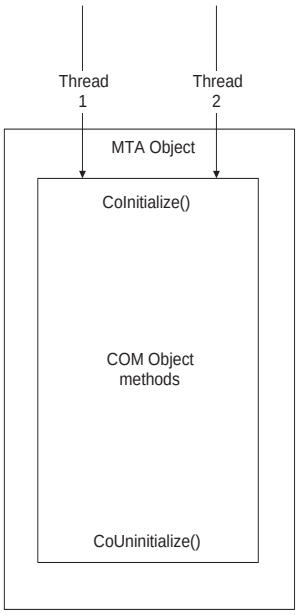

Figure 10–4 Free Threaded Server

Both

A value of 'both' for in-process servers means the object can be loaded in both free and apartment threads. This option is stored in the registry under CLSID/InprocServer32 as ThreadingModel="Both".

This object is free-threaded, and must handle all the thread-safety issues of a free threaded model. By marking itself ThreadingModel=Both, the object can be loaded directly into an in-process apartment thread without the overhead of creating a proxy object.

Normally, when an apartment thread loads a free threaded object, it automatically creates a proxy to marshal the object's interface. The "Both" model allows the loading apartment somewhat faster access without this proxy.

Marshaling Between Threads

Now that you understand the three COM threading models, you need to know more about marshaling. The two topics are closely tied together. As you recall, marshaling is the process of making calls and copying data between COM clients and servers. The whole concept of an apartment is based on not only the execution and synchronization of COM objects, but on the marshaling of data between clients, servers, and threads.

One of the features of MIDL is that it automatically builds all the marshaling code for an object and its client. This marshaling code is doing some complex things to ensure that data and method calls are working properly. COM's marshaling methods need to take into account many factors, especially threading models.

The other factor that we've discussed is synchronization. All calls in an apartment threaded environment are synchronized. Calls in a free threaded environment aren't. Here are some rules that describe how COM will behave when communicating between threads.

COM Access	Synchronization	Marshaling
To the same thread.	None required.	None required. All calls are made directly. Pointers can be passed.
Any thread to an Apartment thread.	All calls automatically synchronized through a message loop.	COM marshals all calls into an apartment thread. (This is done by posting messages.)
Any free thread to a free thread.	Not synchronized. The programmer ensures synchronized access.	No marshaling within the same process. Between processes, all calls are marshaled.
Apartment to Free	Not synchronized.	Marshaled.

Table 10.2 COM communication between threads

Using Apartment Threads

Apartment threads are by far the easiest type to work with. You can effectively ignore all the complexities of multi-threaded access to your COM object. The message queue ensures that methods are called serially. For most applications, the performance is quite good. In fact, you'll probably not get any advantage from free threading.

Commonly, the client thread creates a single object that it always accesses. In other words, there is a one-to-one correspondence between client thread and server thread. In this case, there is never a wait for access to the object. In this case, there would be no advantage to free threading.

There are some disadvantages to apartment threads. The foremost of these is that your COM object will be unresponsive when it is executing a method. If you have multiple clients or multiple client threads accessing the same object, they will have to wait for each other. If your object has lengthy processing to perform, performance will suffer. This is not as common as you might think.

You'll only have performance problems when multiple clients are accessing the exact same object. This is rare because most clients will create their own instance of the object. Multiple instances of the object run in separate apartments, and therefore work independently of one another. Singleton objects are one case where a single object is accessed by multiple threads - in singleton objects apartment threading might be problematic.

Another disadvantage is that apartment threaded objects can't utilize multiple CPU's. Admittedly, this is very rarely a significant performance concern. If you have a singleton class, you're going to have to look carefully at performance issues. Singleton objects share many of the performance concerns of single threaded objects.

Performance for single threaded servers can be a problem. Even if a server has multiple objects, they all run from the same message loop. That means all processing is going through the same pipe, which will cause performance bottlenecks.

Free Threading Model

Free threading is not a panacea for performance problems. It has a lot of disadvantages, and in many cases little benefit. However, for some specific cases it can be very powerful.

Programming free threaded objects is complex. You have to assume that your object is going to be accessed simultaneously by random threads. That means that all member variables are wide open to be changed at any time. Stack based local variables are specific to the calling thread, and will be safe. There are two ways to handle this problem: explicitly write code to serialize access to data (using standard synchronization techniques such as mutexes), or ensure that the object is stateless.

"Stateful" and "stateless" refers to how an object stores data. An object is stateful if it retains information between method calls. Using global, static or even member variables may make your object stateful. Stateful objects are a problem because multiple threads can change their data unexpectedly, causing erratic and failure-prone behavior.

A stateless object doesn't retain information, and isn't prone to being unexpectedly changed. Here's an example of a stateless method call in a COM object.

```
STDMETHODIMP CBeepObj::Beep(LONG lDuration)
{
  ::Beep( 550, lDuration );
  return S_OK;
}
```

The only data used by this method is a local stack variable (lDuration). There's not really much that could go wrong here. This method would work safely in any threading environment.

If you write an object that needs to retain globally accessible data, it will have to use the Win32 synchronization methods to protect the data. If you've done much of this you'll realize what a truly exacting task it can be.

If you're willing to spend the design and programming time to ensure thread safety, you can get some performance advan-

tages. Each free threaded method call executes as soon as it gets CPU time. It doesn't have to wait for other objects to complete. COM maintains (through RPC) a pool of threads that are ready to service any incoming calls.

There is very little marshaling overhead on free threaded objects. For in-process severs, there is no need for a proxy and stub between the object and its client. (Out-of-process servers will almost always require marshaling.) Data can be safely transferred between in-process free threads without any serialization overhead. Because free threaded objects don't have a message pump, there is no need for a busy message loop to rob CPU cycles. Data doesn't need to be placed on, and removed from, the message queue.

Perhaps the biggest gotcha about free threads is marshaling. Even if you create a free threaded server object, it may incur significant marshaling overhead. COM is very conservative about how it marshals data. If the client is apartment threaded, and if the server is free threaded, COM will marshal all access to the object. This marshaling will impact the performance of the object.

Testing the Different Models

You can easily experiment with and understand the three threading models using the beep server presented at the beginning of the book. Create three versions of the server with the Wizard, one with single threads, one with apartment threads and one with free threads. Modify the server so that it beeps for 10 or 15 seconds (or beeps and then sleeps for 10 seconds). Now run multiple clients in separate windows and watch what happens.

In the single threaded case, all of the beep requests from all of the separate clients will be serialized. You will hear each beep for the 10 second duration, followed by the next beep. In the free threaded case you will find that beeps can occur simultaneously because they are being created by multiple threads. If you multi-thread the client so it can simultaneously make multi-

ple calls to Beep, you will be able to see the difference between apartment and single threaded objects.

Summary

COM has four threading models: single, apartment, free, and both. Single threads are a subset of apartment threads. Apartment threads offer good performance for most applications while eliminating most thread synchronization concerns. The apartment model synchronizes access to the COM object, and Marshals data to and from it.

Free threaded objects don't have any synchronization mechanism - the programmer has to ensure thread safety. If you've done much multi-threaded programming, you know how complex and difficult to debug it can be. Writing thread-safe code isn't an easy task.

If you're working with the Single or Apartment model, you don't have to worry about thread safety, but your application may take a performance hit. Free threaded servers are potentially more efficient because they can take advantage of the multi-threading and multi-CPU architecture of windows.

E L E V E N

The COM Registry

This chapter describes how the registry is used to store information about COM servers.

One of the important features of COM is that it allows a client program to use a component without locating, starting, and manually connecting to a server. This greatly simplifies the client program. However, this does mean the information required for server activation must be stored somewhere. On Windows, this type of configuration data is stored in the Registry.

The Registry is a single, well-organized location that stores all system, application, and user configuration information. Normally the user does not manipulate the registry directly, although that is possible using the Registry Editor.

In one sense, COM applications aren't really stand-alone programs. In order to run a COM application, a complex interaction between the operating system and the application takes place. The Service Control Manager (SCM) does the work of locating, starting, and shutting down COM servers. In one sense, the server is just part of a complex interaction between the client, Windows, and the COM components. For a COM server to operate it must have registered itself and its capabilities with Windows.

COM stores three types of information in the registry:

- Human readable information about COM classes
- The mapping of CLSIDs to their servers
- Information about server capabilities

Another name for the registry is the "Class Store". On Windows 95 and NT 4.0, the Class Store is synonymous with the Registry. In future versions, the Class Store will evolve into a centralized storage location for COM information.

There are several ways in which COM information is written to the Registry. Most commonly, COM servers have the ability to store this registration information themselves - also known as self-registration. This capability fits in very nicely with the component model because it allows objects to be responsible for their own configuration. The alternative would be to include an external registration component for each object (such as an REG script).

Self-registration can be implemented in both remote and In-process servers. There are several ways to implement this capability. We'll take a closer look at how ATL servers handle registration. We will also be taking a look at the registry structures that hold COM information.

The COM Registry Structure

The registry is organized in a tree structure. The top level of the tree consists of a number of "Hives" or HKEY's. Exactly which hives you have depends on the operating system. The two hives that are of interest to COM are HKEY_LOCAL_MACHINE and HKEY_CLASSES_ROOT. You'll find these two keys on both Windows 95 and NT.

The key HKEY_CLASSES_ROOT is where all COM's registry information is kept. Actually, if you look carefully, you'll find that HKEY_CLASSES_ROOT is a subdirectory under HKEY_LOCAL_MACHINE. The HKEY_CLASSES_ROOT key is just a shortcut.

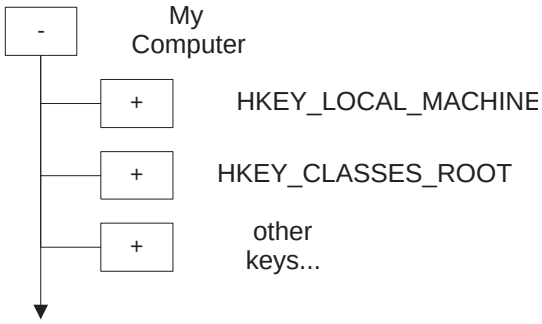

Figure 11–1 The registry is arranged in a series of "hives". The two hives important to a COM programmer are HKEY_LOCAL_MACHINE and HKEY_CLASSES_ROOT

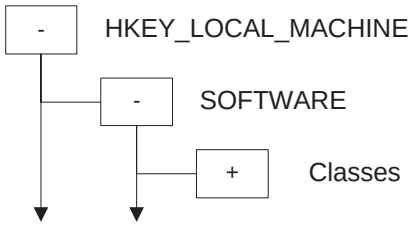

Figure 11–2 HKEY_CLASSES_ROOT is simply an alias for the Classes key found in HKEY_LOCAL_MACHINE/Software.

There are numerous keys and branches under the HKEY_CLASSES_ROOT branch, but only a few basic types. These keys are mappings used to locate servers, classes, and server information. Here's the tree structure:

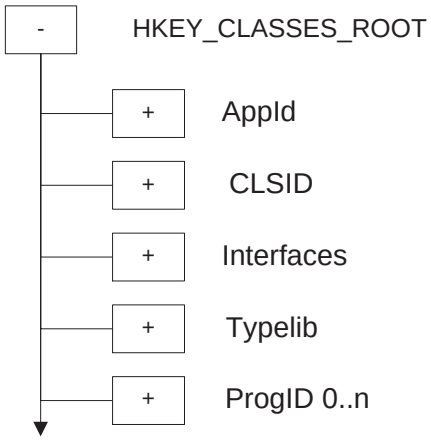

Figure 11–3 Standard keys in HKEY_CLASSES_ROOT

Each of these keys stores a specific type of information.

KEY	Description
AppID	Application ID. Each AppID represents a COM server, which supports a grouping of one or more COM classes.
CLSID	Class ID. The ID of a COM class. The CLSID key is used to map a class to its server. Each entry under this key will be a GUID describing a COM object and it's server.
Interfaces	Information about the interface Proxy/Stub.
ProgID	Program ID. Used to map a readable class name to its CLSID. There are numerous ProgID's under HKCR.
Typelib	Type library information.

Table 11.1 Standard keys in HKEY_CLASSES_ROOT

The most significant of these types is the CLSID, which is a COM class identifier.

Registration of CLSIDs

Class information is stored in the HKEY_CLASSES_ROOT/CLSID key. This branch of the registry has a separate entry for the GUID of each registered COM class in the system. A registry key will be set up for each GUID - surrounded by curly braces. For example, the BaseTypes class has a key of {FF23CF23-89D5-11D1-8149-000000000000}.

When COM needs to connect to the server, it uses the GUID of the Class ID to find server information. CLSID is the only required key for a COM component. At a bare minimum, all COM components should have a valid entry under this registry key.

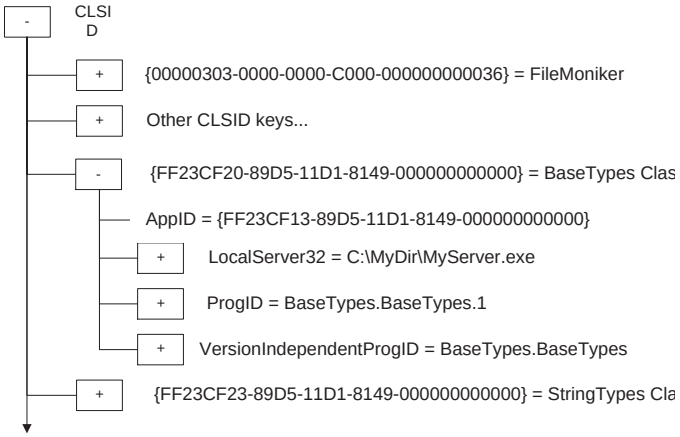

Figure 11–4 The CLSID registry entry for a typical COM server

The CLSID key may have several sub-keys and data entries under it. The exact set of registry entries depends on the specifics of the COM class. In the example above, BaseTypes has a remote (EXE) COM server. This dictates that it has a key called LocalServer32, which points to the name of the executable program.

Following is a table of common values found under the CLSID branch:

Registry Entry	Description
AppID	Associates the AppID with the CLSID. The AppID can be looked up under the \AppID key in the registry.
InprocServer32	The filename and path of a DLL that supports the CLSID.
LocalServer32	The filename and path of a server application that supports the CLSID.
ProgID	The ProgID of the class with a version number.
ThreadingModel	Specifies the threading model for the CLSID if it is not specified. Values can be Apartment, Both, and Free. This key is used with InprocServer32 for in-process servers.
VersionIndependant-ProgID	The ProgID of the class, without a version number.

Table 11.2 Standard CLSID values in the registry.

This is by no means an exhaustive list of values. You will often find other values under this key. Many of these are keys for specific OLE components.

Registration of ProgIDs

ProgID stands for programmatic identifier. The ProgID is a human readable name for the COM class. This is usually the component name used with imported Type libraries (using smart pointers), or when creating Visual Basic objects. Ultimately, the CLSID is the unique identifier for each COM object. The CLSID is always unique, ProgID's are not. ProgID's are just a convenience for locating the CLSID.

Here's the smart pointer example:

```
IBasicTypesPtr pI( _T("BasicTypes.BasicTypes.1") );
```

And the Visual Basic line:

```
Set Testobj = CreateObject("BasicTypes.BasicTypes.1")
```

The COM subsystem will take this name and look it up in the Registry under ProgID. There are two main ProgID entries under HKEY_CLASSES_ROOT. Here's what they look like:

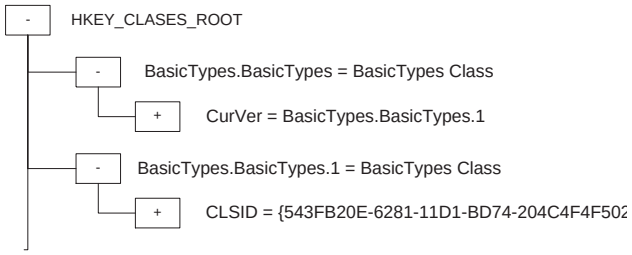

Figure 11–5 The CLSID and ProgID stored in the registry

As you can see, COM can look up the ProgID either with or without a version number. Once the ProgID is found, it is used to determine the CLSID, which is required to create the object.

Most of the entries you'll find under the HKEY_CLASSES_ROOT branch are ProgID's. These typically have the format of <*Vendor*>.<*Component*>.<*Version*>. You'll quickly see that there is a huge variation on this standard. Typically the ATL wizard generates ProgID's that don't follow the standard. ATL names have the format <*Name*>.<*Name*>.<*Version*>. If you want to follow the standard you'll have to modify the code that the wizard generated files.

The ProgID shows up in several other places. You'll often find a copy of the ProgID under the key of a specific CLSID.

Registration of AppIDs

AppID stands for Application Identifier. The AppID key is found under HKEY_CLASSES_ROOT in the registry. You'll also find an AppID string under the CLSID registry key for a COM Class.

AppIDs are used by DCOM to group information about the COM applications. Many COM servers support more than one COM object. The AppID may contain information about how to run the server, if it runs on a remote computer, and access permissions.

Here are some common values found under the AppID key:

Registry Entry	Description
RemoteServerName	The name of a server on a remote computer. This is required if the client program doesn't specify the server in the COSERVERINFO structure when calling CoCreateInstnaceEx().
LocalService	Used to specify that a server runs as a Windows NT service. Used in conjunction with ServiceParameters.
ServiceParameters	This is the command line passed to the server when it is started as a Windows NT service. Value = "-Service"
RunAs	Specifies that the server be run as a specific user. This is often used to give the server network privileges of a particular user.

Table 11.3 Common AppID values

Self-Registration in ATL Servers

There are several ways the server can write entries into the registry. The most direct method is to write a program that writes its values directly into the registry using the Registry API calls. This is conceptually simple, but can be very frustrating in practice.

COM provides a standard interface (IRegister) for registration. In this section, we'll look at how ATL and the ATL wizard handles object registration.

The RGS File

If you look at the resources for an ATL wizard generated server, you'll see a section that contains registry entries.

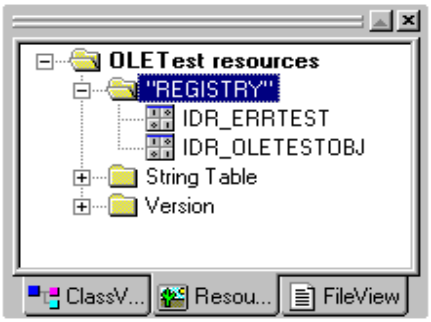

Figure 11–6 Registry resources produced by ATL

These resources are used to identify a new type of registry script. If you look in the file list of the project you'll see a file with the extension "RGS" for each of these entries. Double click on these "REGISTRY" resources and you'll see that they are text file containing the server's registration commands.

This file will be used to automatically update the registry entries for the server. You may be familiar with "REG" scripts used with the REGEDIT application -- the RGS scripts are used by a completely different application. The server's ATL classes implement a special COM interface called IRegister. This interface executes the scripts. IRegister has a limited ability to add, delete, and make simple text substitutions. Here's an example of one of the RGS files.

```
HKCR
{
    BasicTypes.BasicTypes.1 = s 'BasicTypes Class'
  {
        CLSID = s '{543FB20E-6281-11D1-BD74-
  204C4F4F5020}'
    }
    BasicTypes.BasicTypes = s 'BasicTypes Class'
    {
        CurVer = s 'BasicTypes.BasicTypes.1'
    }
    NoRemove CLSID
    {
        ForceRemove
          {543FB20E-6281-11D1-BD74-204C4F4F5020} =
                    s 'BasicTypes Class'
        {
            ProgID = s 'BasicTypes.BasicTypes.1'
            VersionIndependentProgID =
              s 'BasicTypes.BasicTypes'
            LocalServer32 = s '%MODULE%'
            val AppID =
            s '{543FB201-6281-11D1-BD74-204C4F4F5020}'
        }
    }
}
```

The syntax here is straightforward. HKCR stands for HKEY_CLASSES_ROOT. It immediately creates two entries for BasicTypes.BasicTypes, and BasicTypes.BasicTypes.1. If you look under HKEY_CLASSES_ROOT, you'll see these entries.

The script also writes information into the CLSID key of the registry. Under CLSID, the script will write a key for the GUID, and several significant sub-keys such as LocalServer32. Remember that this script works for both registration and unregistration. The "NoRemove" keyword tells it not to delete the CLSID branch when the server unregisters.

Automatic Registration of Remote Servers

If the server runs as an EXE or service, the registration is accomplished with a special startup command:

```
MyServer - RegServer
```

Let's look at the code ATL generates. The following was taken from the WinMain function of the IDLTEST server from chapter 6, IdlTest.CPP.

```
if (lstrcmpi(lpszToken, _T("RegServer"))==0)
{
  _Module.UpdateRegistryFromResource(IDR_IdlTest,
      TRUE);
  nRet = _Module.RegisterServer(TRUE);
  bRun = FALSE;
  break;
}
```

When the server is run from the command line, it checks for the "RegServer" command. This command tells the server to write its settings into the registry and exit immediately. In this example object **_Module** is an ATL class of type CComModule.

The first function called is UpdateRegistryFromResource(). If you step into this module you'll see some familiar COM behavior. This CComModule class calls CoCreateInstance on the IRegister interface, then calls a method named ResourceRegister, passing in the ID of the RGS file's resource.

The unregistration is simply a mirror image of registration. The server is invoked with a command line of "UnRegserver". Note the boolean FALSE passed into UpdateRegistryFromResource(). Here's the source from the server main routine:

```
if (lstrcmpi(lpszToken, _T("UnregServer"))==0)
{
  _Module.UpdateRegistryFromResource(IDR_IdlTest,
      FALSE);
  nRet = _Module.UnregisterServer();
```

```
    bRun = FALSE;
    break;
  }
```

In-Process Servers

Servers implemented as DLL's have a different registration scheme. Each COM DLL must contain two exported functions for server registration. These are **DllRegisterServer** and **DllUnregisterServer**. These functions implement the same registration functions as a remote COM server.

Because you can't directly run a DLL, registration is handled somewhat differently. Windows provides a utility called REGSVR32, which can register a DLL. The way REGSVR32 works is that it finds and loads the DLL containing the In-Process server, then calls the DllRegisterServer function. This is the same utility that we have used to register Proxy/Stub DLL's. It is executed automatically as part of the build process, or you can run it manually.

Using the Registry API

How you accomplish the registration of components is your own business. If you like doing things the old-fashioned way, you can skip the RGS files and directly call the registry API functions.

These consist of functions like RegCreateKey() and RegDeleteValue(). In the old days of COM this is how all server registration was accomplished. If you're not familiar with these functions they can be somewhat counterintuitive. The help files describe how to use these functions.

Summary

COM uses the registry as an storage area for all information releated to COM servers and interfaces. When a COM client

wants to access a COM server, the operating system uses the information in the registry to find, start and control the server. By becoming familiar with the information in the registry, you improve your ability to understand and debug COM applications.

The registry is also one of the areas responsible for many COM errors. For example, if a server does not properly self-register, then the client will not be able to activate it. See the error-handling appendix, which discusses many of the problems that can occur in the registry.

T W E L V E

Callback Interfaces

So far, all the interfaces we've seen are strictly one directional - a client program connects to a COM server and calls its methods. Most COM interfaces are driven entirely by the client. The client makes the connection, uses the connection, and shuts it down when finished. This works great for simple methods that complete relatively quickly.

For more complex server applications, this client driven design can break down. Often the server needs to send responses to the client. For example, a server may need to notify the client whenever some asynchronous event takes place.

An example of this would be a server that generates reports. These reports are created from a database, require extensive searches, and make lengthy calculations. The client GUI program would call a method on the server called DoReport. The DoReport method might take several minutes to complete. Meanwhile, the client GUI would be stalled, waiting for the report to complete. Obviously, this is a poor design.

A better solution would be for the client GUI to call a method named StartReport which causes the server to spawn a worker thread that handles the lengthy report generation. StartReport would start the worker thread and return as soon as it was started. The client could then do other work, such as dis-

playing progress. After several minutes, the server would tell the client GUI that it was finished. The client GUI would call a method named GetReportData, and display the complete report.

The simplest way to do this is for the client program to constantly poll the server.

```
BOOL IsReady;

// Start a worker thread on the server
pI->StartReport();

// Check if report is done
pI->CheckReport( &IsReady );

while(IsReady == FALSE)
{
  Sleep( 60000 );// wait 1 minute
  pI-> CheckReport ( &IsReady ) )// poll the server
}

// get the report
pI->GetReport( &data )
```

There are three problems with this code. First, there is potentially a minute delay before the event is processed (because of the duration of the Sleep statement). The second problem is efficiency. You can shorten the Sleep delay at the expense of efficiency. For remote network connections this could mean expensive and unnecessary network traffic. The fundamental trade off is between responsiveness and efficiency. If you can afford the waiting, this is a simple way to design an interface.

A more efficient way to design this program is for the server to make a COM connection back to the client. When the server finishes processing, it immediately notifies the client. There are two ways to do this - custom callback interfaces and connection points. In essence, we will use COM to create an asynchronous link from the server to the client.

While conceptually simple, the implementation of this bi-directional design can be complex. After debugging a bi-directional client and server, you may reconsider the polling interface shown above.

This chapter covers the simpler of the two - custom callbacks. Connection Points are more flexible, but considerably more complex and are discussed in the following chapter. Both of these techniques have advantages and disadvantages in specific situations.

Client and Server Confusion

Before we embark on further explanation, here are a few words of caution. This subject can be quite confusing. It is difficult to keep track of clients and servers. The concepts aren't all that complicated, but they are hard to track mentally.

While the diagrams are relatively simple, the description can be difficult. The basic problem is this: each object is both a COM client and a COM server. The labels "client" and "server" have little meaning in this context.

Another point of distraction is the implementation of callbacks. To demonstrate this concept, we will need both a server and client application. Because the two are closely tied together, we can't explain one without explaining both. The actual callback interface is extremely simple, but the interaction is complex.

Custom Callback Interfaces

A callback is simply a function on the client that is called by the server. In COM, it's perfectly OK for a client application to also expose COM objects. The server can connect back to a client object. This does blur the distinction between client and server.

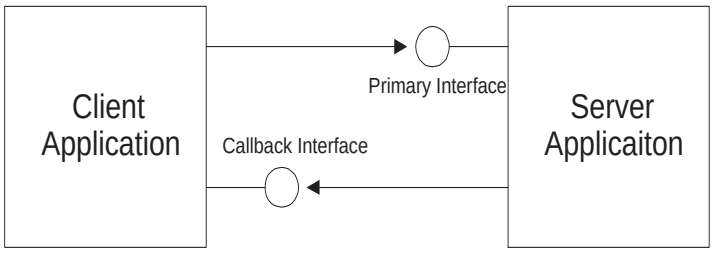

Figure 12–1 With a callback, the server can talk back to the client

The COM/OLE world uses the terms 'source' and 'sink' to describe bi-directional interfaces. In the above diagram, the client application has a sink interface. This interface is used by the server application to notify its caller. The source is a source of events. In other words, it's an object that makes the connection back to the client application. The source connects to the sink.

We are going to build a dialog-based client program called 'CallbackClient' that implements a COM interface. We'll also design a server which implements an interface allowing the client to 'register' itself. Once the client is registered, the server has a way to connect back to it.

In the COM vocabulary, registering a callback interface with the server is often called an "Advise". Basically, the Advise() method makes a copy of the client's callback interface, and stores it for later use. When the client disconnects, it "un-advises" its callback.

In Figure 2, Notice that the CCallBack object, and its interface, are *inside* the callback Client Application box. They also have dashed lines. It was drawn this way to show that ICallBack interface is not exposed to the outside world. Unlike most COM objects, this Object cannot be connected through a normal call to CoCreateInstance(). The only way for the Callback Server to get this object pointer is when the client explicitly passes it to the server. We'll see how this is done in the example.

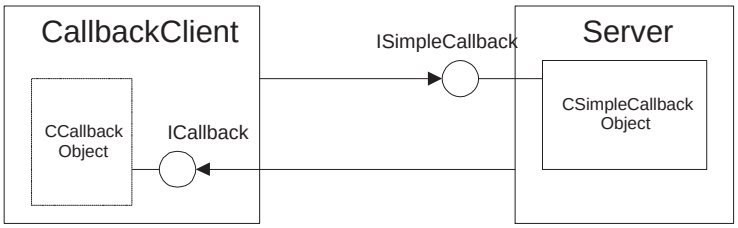

Figure 12–2 The relationship between server and client when a callback is used. Note that a COM server to handle the callback is embedded within the client.

A Callback Example

Here is an outline of the steps we'll follow to implement the server object. We'll describe each step in more detail below.

1. Create a COM Server using the ATL Wizard. Name the server CallbackServer.
2. Add a COM object to the server. Name the object SimpleCallback. Use the ATL object wizard.
3. Add the definition of a COM interface named ICallBack to the IDL code for the server. Note that we won't be implementing the ICallBack interface in the server, we're just adding a definition.
4. Add four methods to the ISimpleCallback interface on the server: Advise(), UnAdvise(), Now() and Later()

Create the Server

First we'll define the CCallbackServer server program. There's nothing special about this server. Use the wizard to create an ATL COM AppWizard project. You could implement the server either as an in-process server, or an EXE based server. Note that an EXE server is slightly more complex to build, and also harder to debug.

The sample code was built as an EXE server. These techniques work with any type of server - it will work just as well as a service or a DLL.

Add a COM Object to the Server

Add a COM object to the server using the ATL Object Wizard. Select a simple COM object. Give it the name SimpleCallback. On the attributes page select the following:

- Apartment threading model.
- Custom Interface. (Dual would also work)
- Either yes or no for aggregation.

Note that the "Support Connection Points" option is completely unnecessary for a custom callback method. We'll use this option in a later section when we add a Connection Point in the next chapter.

Next we'll add four methods: Advise(), UnAdvise(), Now(), and Later(). Look in the file CallbackServer.IDL. The wizard generated MIDL definitions have an interface that looks like the following code. I've stripped out some extraneous material, and of course, the GUID's will be different.

```
 [
    object,
    uuid(B426A80D-50E9-11D2-85DA-004095424D9A),
    helpstring("ISimpleCallback Interface"),
    pointer_default(unique)
]
interface ISimpleCallback : Iunknown
{
    HRESULT Advise([in] ICallBack *pICallback,
      [out] long *lCookie);
    HRESULT UnAdvise([in] long lCookie);
    HRESULT Now([in] long lCode);
    HRESULT Later([in] long lSeconds);
};
```

Don't laugh about the cookies. We'll explain how they are used later.

Adding the ICallBack Interface to IDL

Next, we'll add the callback interface definition to the IDL code of the server. The callback interface will not be implemented by this server. We will write the callback interface when we write the Client application. We are just going to add the IDL code. Although we're not implementing this interface, the server needs its definition.

We can't use the ATL Object Wizard, because it will also add the CPP and H files. We're using MIDL as a convenient way to generate header definitions. We'll include the headers in the client program.

Type in the following definition to the CallbackServer.IDL file. Put it near the top of the file where it's easy to find, just above the definition of the ISimpleCallback interface.

```
// implemented on the client only
[
  object,
  uuid(B426A80D-50EA-11D2-85DA-004095424D9A),
  helpstring("ICallBack Interface"),
]
interface ICallBack : Iunknown
{
  HRESULT Awake( long lVal );
};
```

Modify the Header

Now we'll modify the CSimpleCallback object and add two member variables. We'll add a cookie and an ICallBack interface pointer to the class definition. Find CSimpleCallback in the header file SIMPLECALLBACK.H. Add these two variables to the public part of the class definition.

```
long m_lCookie;
ICallBack *m_ICallBack;
```

Adding the Advise Method to the Server

Now we'll add code to SimpleCallback.CPP. The first method we'll add is Advise(). The purpose of this method is to save a pointer to the client's callback interface. The client program will call this method, passing in a pointer to its callback interface. Note that the client is responsible for creating the ICallBack interface pointer - we're NOT going to call CoCreateInstance.

```
// Register the callback
STDMETHODIMP CSimpleCallback::Advise(ICallBack
  *pICallback, long *lCookie)
{
  // Save the pointer
  m_ICallBack = pICallback;
  // keep the interface alive by calling AddRef
  m_ICallBack->AddRef();
  // Make up a cookie with a semi-unique number
  *lCookie = (long)this ;
  m_lCookie = *lCookie;
  return S_OK;
}
```

The client passes in a pointer to its own COM interface. This method will do an AddRef() on the callback interface, and save the ICallBack COM pointer. All AddRef is going to do is increment the ICallBack interface reference count.

Notice that the client passed us an ICallBack interface. There is no ambiguity here - this method only accepts ICallBack interfaces. When we use connection points later, we'll see that they are more flexible.

Finally, we get to the cookie. You may already be familiar with Internet cookies - COM cookies are somewhat different. The cookie is a unique ID that the server retains to keep track of connected clients. The server will use this value later, when it needs to close down the connection. We've used the **this** pointer for a semi-unique number. The cookie only has to be unique within the context of our server.

The only purpose of the cookie is to ensure that the client un-advises the same interface it advised. This is an unnecessary check in this example program, but more complex servers may require it.

Adding the UnAdvise Method

Let's take a look at the UnAdvise() method. It is going to close the connection made by the previous call to Advise().

```
// Remove the callback object
STDMETHODIMP CSimpleCallback::UnAdvise(long lCookie)
{
  // Compare the cookie. Be sure this is same client
  if (lCookie != m_lCookie) return E_NOINTERFACE;

  // Release the clients interface
  m_ICallBack->Release();

  m_ICallBack = NULL;

  return S_OK;
}
```

We check the cookie to ensure this is the right client. In this example there's not much to do if the cookie is wrong, so we return a generic COM error. Finally, we Release the interface pointer we've been saving. Remember, we did an AddRef() in the Advise method. This will allow the client to shut down with no outstanding connections.

Calling the Client from the Server

Now we get to the heart of our callback interface - the callback! All that this function calls is a method on the client (sink) interface. Since we already have the pointer, this method works just like any COM interface. The Awake method is quite simple, but there's nothing to prevent methods that are more complex. In

fact, our Later() method will be used later to demonstrate a multi-threading example.

```
// Callback the client immediately
STDMETHODIMP CSimpleCallback::Now( long lCode)
{
  HRESULT hr = E_FAIL;
  if (m_ICallBack != NULL)
  {
    // Call method synchronously.
    // This will not return until
    // the client presses OK on the MessageBox.
    hr = m_ICallBack->Awake( lCode );
  }
  return hr;
}
```

This isn't a very realistic use for a callback interface. Normally, the server (source) object would execute the callback with some important notification for its client. With a little imagination, you can come up with useful implementations of a callback.

Notice that the call to Awake() is synchronous. That means the client's call to Now() won't complete until the server's Awake() callback completes. This means the client is waiting for itself! This doesn't solve our original problem of waiting for the server. Don't worry, we will provide a usable threading solution later with the Later() method. For now, the Now() method will demonstrate basic concepts.

If this discussion seems a little hypothetical, it's because we haven't seen the client application yet.

Note: Build the server, and don't forget to build the Proxy/Stub DLL. Use the BuildMe.bat file to automate this task. We'll add the Now and Later implementation later, but the MIDL code won't change. The test client requires these MIDL generated headers.

The Client Application

With the client application we're going to do something a little different. Up to this point, we've implemented all our COM objects using the ATL wizards. For this program we're going to write the ATL code ourselves. It's really quite easy.

The client is a standard MFC dialog-based application. This will be a plain-vanilla MFC dialog application, which we will modify to implement a COM object. Next we'll write the callback (sink) COM object. Finally, these two elements will be hooked together. Here's the sequence of events.

1. Create an MFC Dialog Based Application. Call it CallbackClient.
2. Modify the client's main program to support the Callback Interface (ICallBack). The definition of the ICallBack object comes from the MIDL code of the CallbackServer server. We'll set everything up in the application InitInstance and ExitInstance methods.
3. Attach code to the button that tests the callback.

Create the Client Dialog Application

Go to the File/New tab and create a new MFC AppWizard project. I've named the project CallbackClient. Take all the standard dialog application options.

The AppWizard will create a dialog named IDD_CALLBACKCLIENT_DIALOG. Edit this dialog and delete the OK button. Leave the Cancel button.

Next we'll add a new button, with the ID as IDD_BUTTON1. Change the text of the button to read "Now". Add IDD_BUTTON2 for the "Later" button. The dialog should now look similar to Figure 12-3.

Next, edit the STDAFX.H file to include ATL. If you leave out this line, the compiler won't recognize ATL templates such as CComModule. Add the following line near the end of STDAFX.H:

```
#include <atlBase.h>
```

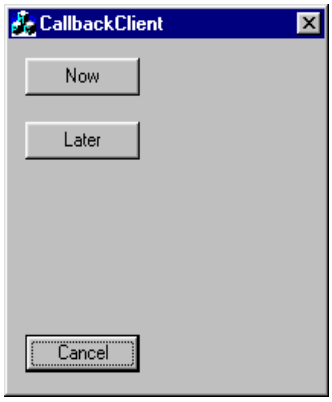

Figure 12–3 The sample application is a simple dialog

Our dialog should now compile, but it will just be an empty shell. In the examples that follow we will be leaving out the part of the application we haven't modified. We will only present enough of the application framework to give a context. For a holistic look at this application you'll need to look at the example source code on the CD.

Adding the Callback COM Object

Next we're going to modify the CallbackClient application to support our ICallBack interface. To do this, we'll have to manually add a COM object to the application. This is the first time we're not going to use the ATL wizards, but it's really easy to do.

Edit the main application source module, CallbackClient.CPP, and add the following class definition at the top, just after the #include section.

```
CComModule _Module;    // Define main COM module.
                       // Required for <atlcom.h>
#include <atlcom.h>    // definition of CcomObjectRoot
```

```
// Callback interface to be implemented on client
class CCallBack :
    public ICallBack,           // Use this interface
                                // (server.idl)
    public CComObjectRoot       // Use ATL
{
public:
    CCallBack() {}              // Default constructor

    // Define COM object map
    BEGIN_COM_MAP(CCallBack)
    COM_INTERFACE_ENTRY(ICallBack)
    END_COM_MAP()

// Icallback
public:
    // The callback method
    STDMETHOD(Awake)(long lVal);
};

// Create object map for callback interfaces
BEGIN_OBJECT_MAP(ObjectMap)
END_OBJECT_MAP()
```

There's a lot going on here so we'll break it up into smaller bites.

First, we need to include ATL in the source code. This is done by including <atlcom.h>. If you just plop this definition down anywhere in the code, you'll get lots of errors. This is because this ATL header is assuming a main module named "_Module" is already defined. "_Module" is a magic name, and you'll have to declare it in every ATL module.

The ATL class CComModule handles all the plumbing of starting, stopping, and registering a COM server. Needless to say, there's a lot going on in CComModule. Fortunately, we don't have to understand all this behind-the-scenes magic to use it.

```
CComModule _Module;
```

Once this symbol is defined, we can proceed with our class definition. We're going to inherit from ICallback, and the root of all ATL objects, CComObjectRoot. This class is required for all ATL objects, it provides the required implementation of IUnknown - in other words, QueryInterface, AddRef, and Release.

```
// Callback interface to be implemented on client
class CCallBack :
    public ICallBack,           // Use this interface
                            // (server.idl)
    public CComObjectRoot       // Use ATL
```

Linking to the Server Headers

The definition of ICallBack is a bit of a mystery here. Remember, this Interface is defined in the CallbackServer server. We put a definition of the interface in the IDL code for the server. When MIDL was executed, it emitted two very useful files: CallBackServer_i.c and CallBackServer.h. The "i.c" file includes GUID definitions, and the ".h" file defines the ICallBack interface in C++.

To get these definitions, include the following statements in the header file CallbackClient.h:

```
#include "..\CallbackServer\CallBackServer_i.c"
#include "..\CallbackServer\CallBackServer.h"
```

You may need to change the path to these files, depending on where the client and server project was created.

COM Maps

Meanwhile, back at the CallbackClient.CPP module, we need to add some more code. The next part of the class is the interface map. This interface map sets up an array of interface IID's . The COM object will use these IID's when it calls QueryInterface. These macros hide a lot of code, and if you mistype any of the entries you may get some unusual, and apparently unrelated, error messages. Luckily, our COM object only has one interface.

```
BEGIN_COM_MAP(CCallBack)
 COM_INTERFACE_ENTRY(ICallBack)
END_COM_MAP()
```

Implementing the Callback Method

This COM object has only a single method called Awake. All that our implementation does is display a message box.

```
STDMETHODIMP CCallBack::Awake(long lVal)
{
    CString msg;
    msg.Format( "Message %d Received", lVal );
    AfxMessageBox( msg );
    return S_OK;
}
```

In a real-life implementation of a callback, this method might be considerably more complex. The purpose of this method is to notify the client application of a server event, such as report completion. Obviously, there isn't any code in this example to do this, so we simulate it with a Message Box.

There are also some significant threading issues here. We need to be aware that the dialog box and CCallBack objects are running in the same apartment (i.e. thread). You've got to be careful about the callback blocking execution for the dialog.

At the end of the CallbackClient.CPP file, we have the normal AppWizard generated MFC application class. In this case, the application is called CCallbackClientApp, and inherits from CWinApp - a standard dialog based application. We're going to add a few lines of code to set-up our server connection.

Adding the Object Map

The object map is located just after the class definition. The purpose of this structure is to maintain an array of ATL objects that will be supported. We're not exposing any ATL objects to the outside world, so we don't need any entries. Any ATL objects that are put in this structure will be registered when _Module.Init() is called.

```
// Create object map for callback interfaces
BEGIN_OBJECT_MAP(ObjectMap)
END_OBJECT_MAP()
```

That points out another difference between our ICallBack interface and a normal COM interface. We aren't allowing other programs to instantiate a CCallBack object and interface by calling CoCreateInstance. It isn't necessary because we'll be creating the object internally, and explicitly passing it to any outside objects that need it.

Connecting to the Server

In COM, everything starts with the client. By defining our CCallBack class first, we're getting ahead of ourselves. Before anything can happen, we need to initialize COM and connect to the server. We're doing this in the application's InitInstance method.

InitInstance is a standard method of CWinApp, and gets called to display the application's main dialog.

```
// Initialize the application
BOOL CCallbackClientApp::InitInstance()
{
  AfxEnableControlContainer();

  // Initialize COM interfaces
  InitCOM();
etc...
```

We've added the InitCOM method, which we're about to implement. Add this method to the header and enter the following code.

```
BOOL CCallbackClientApp::InitCOM()
{
  HRESULT hr;

  CoInitialize(0); // Initialize COM
```

```
    // Initialize the main ATL object
    _Module.Init( ObjectMap, 0 );

    // Create a server object
    m_pSimple = NULL;
    hr = CoCreateInstance( CLSID_SimpleCallback,

    0,CLSCTX_SERVER,IID_ISimpleCallback(void**)&m_pSimpl
    e );
    if (SUCCEEDED(hr))
    {
      // Create a callback object
      CComObject<CCallBack>* pCallBack = NULL;
      CComObject<CCallBack>::CreateInstance( &pCallBack
    );
      pCallBack->AddRef();

      // Set up the callback connection
      hr = m_pSimple->Advise( pCallBack, &m_lCookie );

      // Done with our ref count. Server did an AddRef
      pCallBack->Release();
    }
    return SUCCEEDED(hr);
  }
```

We must, of course, start by initializing the COM subsystem. We're using apartment threading, so the old-fashioned CoInitialize works fine. Next, we initialize the ATL main module. This is done by calling the Init() method on the _Module object. This gets ATL going and ready to serve the CCallBack COM object.

```
    hr = CoCreateInstance( CLSID_SimpleCallback,
                           0, CLSCTX_SERVER,
                           IID_ISimpleCallback,
                           (void**)&m_pSimple );
```

We connect to the server in the usual way, with CoCreateInstance. This starts up the server and delivers a server-side COM interface. If you haven't done all of your server registration

correctly, you'll probably get an error like "Class Not Registered" here. If you did everything perfectly on the server, we're ready to call Advise and register our callback interface with the server.

```
CComObject<CCallBack>* pCallBack = NULL;
CComObject<CCallBack>::CreateInstance( &pCallBack );
pCallBack->AddRef();
```

This code looks a lot like ATL templates - for good reason. We use the CComObject template to define a pointer to our client-side CCallBack class. We instantiate this class using its CreateInstance method.

CreateInstance is a static method that provides an efficient way to create a local COM object. There's more going on here than first meets the eye. Notice that we're not calling CoCreateInstance, the usual way of getting a COM interface. We're cheating a little because CCallBack is implemented locally.

Normally, COM restricts access to COM objects to their interfaces only. That doesn't make sense here because we're actually **implementing** the object. The object creation process is normally hidden by CoCreateInstance, but in this case, we can see it. Because everything's local, we skip all CLSID's and registration entirely. We do an AddRef on the object, to ensure that it stays around for a while.

```
hr = m_pSimple->Advise( pCallBack, &m_lCookie );

// Done with our ref count. Server did an AddRef
pCallBack->Release();
```

We created this COM object so we could pass it into the server. This is done in the Advise method. If you remember the server side, the interface is copied and AddRef'ed by the server. This leaves us free to release the object, and let normal COM lifetime management take its course. In the implementation of UnAdvise, we'll see where the CCallBack object is finally released and can shut itself down

Cleaning Up

Eventually, the user is going to press the cancel button and shut down the application. At this point, we need to close the server connection and UnAdvise our callback. We put this code in ExitInstance, which is called right before the application shuts down. ExitInstance is a virtual method of CWinApp. We'll add ExitInstance to the CCallbackClientApp header, and enter the following code.

```
int CCallbackClientApp::ExitInstance()
{
  // If we have a server object, release it
  if (m_pSimple != NULL)
  {
    // Remove server's callback connection
    m_pSimple->UnAdvise(m_lCookie);
    // Release the server object
    m_pSimple->Release();
  }
  // Shut down this COM apartment
  CoUninitialize();
  return 0;
}
```

This is very straightforward code. We UnAdvise our callback and release the server. Finally we shut down the COM apartment with CoUninitialize.

This concludes the application portion of our server. What's left is almost trivial - we add the button methods for Now and Later.

Adding the OnButton Code

Now that we have wired-in our callback sink into the main application, it's time to build a test method. We're going to hook this test method up into the "Now" button on the main dialog. Use the class wizard to add a method to the dialog called OnButton1. You can also add the method for the "Later" button.

The class wizard will generate all the usual message maps for the two buttons. We're creating these methods on the actual dialog class, not the main application. The end result is two OnButton methods on the CCallbackClientDlg dialog. These two OnButton methods will act as our test platform.

```
void CCallbackClientDlg::OnButton1()
{
  HRESULT hr;
  CCallbackClientApp *pApp =
      (CCallbackClientApp*)AfxGetApp();
  hr = pApp->m_m_pSimple->Now(1);
  if (!SUCCEEDED(hr)) AfxMessageBox( "Call Failed" );
}

void CCallbackClientDlg::OnButton2()
{
  HRESULT hr;
  CCallbackClientApp *pApp =
      (CCallbackClientApp*)AfxGetApp();
  hr = pApp->m_pSimple->Later(5);
  if (!SUCCEEDED(hr)) AfxMessageBox( "Call Failed" );
}
```

Since we already connected to the server in InitCOM, we don't have to do much here. We just get a pointer to the main application and use its COM pointer. We call AfxGetApp() to get a pointer back to our main application. The COM interface pointer is called "m_pSimple", and we use it to call a method.

Note that we haven't implemented the Later method on the server. It won't do anything. At this point, the code is complete. We have presented a large block of source - it was unavoidable. This example covers a complex interaction between a client and server. Build the client and press the "Now" button.

A Chronology of Events

The purpose of the following list is to follow the sequence of events required to make the callback. Since a callback involves the close interaction of a client and server application, we've included both sequences here.

CLIENT DIALOG	COM SERVER
The client application is started. It calls InitInstance() on the application class. This will initialize the objects required for the application.	
The InitCOM() method is called. This is a custom method we wrote to initialize all COM objects. CoInitialize is called to initialize COM.	
InitCOM creates a CCallBack object using CComObject::CreateInstance. This object will remain in existence during the lifetime of the client. It is an ATL COM object.	
InitCOM() instantiates an ISimpleCallback interface on the server application by using CoCreateInstance().	
	The server will be automatically started, and the CSimpleCallback object is created. This object will remain until the client releases it.
InitCOM() passes a pointer to a CCallBack object to the server's Advise() method	.
	The Advise method makes a copy of the ICallBack interface. It calls AddRef to lock the object. It creates a cookie and returns control to the client.
InitCallback() releases the CCallBack object created with CreateInstance.	

CLIENT DIALOG	COM SERVER
...	...
The user presses the "NOW" button, calling OnButton1. The client program calls Now() on the server	
	The Now method immediately calls Awake on the client. It uses the saved ICallBack interface it received in Advise.
Awake displays a message box. The user presses OK to clear the box. Awake completes.	
	The call to Awake returns. The Now method completes.
The call to OnButton1 method completes.	
...	...
The user presses the "CANCEL" button. The main dialog closes and is destroyed. The main application calls ExitInstance.	
ExitInstance calls UnAdvise, passing in the cookie.	
	UnAdvise releases the ICallBack interface and returns.
The CCallBack objects reference count goes to 0. ATL automatically shuts down and deletes the CCallBack object.	
The client calls Release on the ISimpleCallback interface on the server.	

CLIENT DIALOG	COM SERVER
	The reference count to CSimpleCallback goes to 0. The server shuts down.
ExitInstance calls CoUninitialize. The client application closes.	

Table 12.1 Interaction between client and server when using a callback

In the preceding example, we built a client and server application. These two applications work together to demonstrate all the basic points of a bi-directional callback interface. Although informative, this isn't a realistic example of how callbacks are used.

The whole point of this exercise was to demonstrate how a COM server can notify a client program that asynchronous events occur. Unfortunately, when the client calls the Now() method everything is blocked until it completes. We can solve this problem with multi-threading.

A Multi-Threaded Server

Now that everything works, we're going to add a worker thread to the COM server. This worker thread allows the server to accomplish lengthy processing without locking the client. When the client application calls the COM server, it will kick-off a processing thread and return immediately. This thread will run for awhile, then notify the client that it's finished.

Here's the interaction:

CLIENT DIALOG	COM SERVER
User presses "LATER" button. Client calls the Later() method on the ISimpleCallback interface.	
	Later method starts a worker thread. It returns as soon as the thread starts.
The Later method finishes. The client dialog waits for the next command.	
	Several seconds elapse...
	The worker thread finishes processing. It calls the Now() method on itself (using the ISimpleCallback interface.)
	The Now() method calls the Awake() method on the client application.
Awake displays a message box. It returns when the user presses OK.	
	The worker thread completes, and shuts itself down.
	The server waits for it's next call.

Table 12.2 Multithreaded interaction with a callback

If you've done much multi-threaded programming, you know what you're in for. Creating a worker threads in Win32 is quite easy - doing it right is not! Multi-threaded programming can cause problems in thousands of ways that you never imagined. Nevertheless, multi-threading provides some tremendous benefits.

If you're an experienced multi-threaded programmer, much of the following material is obvious. I've described some of the basics of threading for the benefit of those readers who need some review. The only thing unique about this code is the inter-thread marshaling used to pass a COM pointer.

Starting the Worker Thread

The Later() method is going to launch a worker thread, then return to caller. We're going to use AfxBeginThread to start the worker thread, and pass it a C++ object. This C++ object will start COM, do some processing, and call a method back on the main thread. Later() is called directly by the client, after the call-back is registered. Here's the code:

```
STDMETHODIMP CSimpleCallback::Later(long lSeconds)
{
  HRESULT hr;
  CWinThread *pt = NULL;// ID of created thread
  IStream *pStream;// OLE Stream interface
  ISimpleCallback *pSimple = NULL ;// Copy of this
                              // interface

  // Query ourselves
  hr = QueryInterface( IID_ISimpleCallback,
     (void**)&pSimple);
  if (!SUCCEEDED(hr)) return hr;

  // Marshall an interface pointer in the stream
  hr = CoMarshalInterThreadInterfaceInStream(
      IID_ISimpleCallback,
      pSimple,
      &pStream );
  if (!SUCCEEDED(hr)) return hr;

  // Create a processing thread object
  CWorkerThread *pObj = new CWorkerThread();

  // Set object variables
  pObj->m_pStream = pStream;
  pObj->m_lWait = lSeconds;

  // Create and start a thread to do
  // some processing. Pass in a
  // pointer to the thread object.
```

```
    pt = AfxBeginThread( CWorkerThread::StartProc, pObj
    );
    if (pt == NULL) hr = E_FAIL;

    // Release our reference to the interface.
    pSimple->Release();

    // Return to the calling client
    return hr;
}
```

The first thing we're going to do is get an interface pointer to our ISimpleCallback object. We'll use QueryInterface to get a pointer to the interface. This interface pointer is going to get passed to the worker thread so it can communicate back to us.

```
ISimpleCallback *pSimple = NULL ;

// Query ourselves
hr = QueryInterface( IID_ISimpleCallback,
   (void**)&pSimple);
```

Marshaling the Interface Between Threads

When we start the worker thread, we're immediately going to have some tricky threading issues. This is an apartment-threaded server, so the COM object and its worker thread are going to be running in different apartments (i.e. threads).

One of the rules of COM is that interfaces must be marshaled when used between threads. This means we can't just use a pointer to the COM interface, we've got to set up marshaling code. This is something we haven't done yet. Fortunately, there's a simple way to marshal interfaces. We'll use the CoMarshalInterThreadInterfaceInStream method.

```
hr = CoMarshalInterThreadInterfaceInStream(
    IID_ISimpleCallback,
    pSimple,
    &pStream );
```

We're using IStream for inter-thread marshaling. The IStream interface will be used to pass a COM pointer between the main server thread, and our worker thread. IStream is one of those ubiquitous OLE interfaces that you often see used in COM code. The receiving end of this call will be CoGetInterfaceAndReleaseStream, which will be called on the worker thread.

The end result of this process is an IStream object, that is used to marshal the ISimpleCallback interface. Later on, we're going give a pointer to the IStream to our worker thread object. If you want more information on streams, see any of the numerous OLE books and articles.

Starting the Worker Thread: Part 2

First we're going to instantiate our worker thread object. We'll show the definition of CWorkerThread in the next section. The CWorkerThread class has two member variables. The IStream pointer stores the IStream we created with CoMarshalInterThreadInterface-InStream.

The m_lWait member is used to set the timeout period of the worker thread. The worker thread will basically sleep this amount of time before it notifies the client that it's finished.

```
CWorkerThread *pObj = new CWorkerThread();

// Set object variables
pObj->m_pStream = pStream;
pObj->m_lWait = lSeconds;

// Create and start a thread to do
// some processing. Pass in a
// pointer to the thread object.
pt = AfxBeginThread( CWorkerThread::StartProc, pObj
);
if (pt == NULL) hr = E_FAIL;
```

One of the standard ways to start a thread in MFC is AfxBeginThread. We'll pass it a pointer to a static ThreadProc, and a pointer to our worker thread object.

The main routine of a worker thread is called a "ThreadProc". A ThreadProc is analogous to the "main" function of a "C" program, or the "WinMain" of a Windows application. This is the starting address of the newly created thread. We'll name our ThreadProc "StartProc". Notice that the ThreadProc is a static member of the CWorkerThread class. Being static is a requirement - AfxBeginThread will be given the address of this method.

AfxBeginThread starts a worker thread, and transfers control to the ThreadProc. AfxBeginThread always passes in a single parameter to the worker thread, a pointer. In this case, we're going to give the worker thread a pointer to our CWorkerThread object. Let's look at the definition of that object.

A Simple Worker Thread Class

We're going to define a class that encapsulates the threading behavior we need. This class is going to run as a worker thread, which means it doesn't have a window or a message loop. This class will do its processing, then exit.

```
class CWorkerThread  : public CwinThread
{
public:
  // Thread start function. Must be static.
  static UINT StartProc( LPVOID pParam );

  // pointer to stream interface used in marshaling
  pointer
  IStream *m_pStream;

  // number of seconds to wait
  long m_lWait;
};
```

As you can see, this is a simple class definition. We're going to put all the thread's processing logic into the one and only

method - the ThreadProc. For more sophisticated processing, you'll need a more sophisticated thread class.

Implementing the Worker Thread

The worker thread only has a single method. This method will do all the required calculations, then send a message back to the client when it's done. Here's the one and only worker thread method:

```
UINT CWorkerThread::StartProc( LPVOID pParam)
{
  HRESULT hr;

  // Get the object pointer
  //we passed in to AfxBeginThread.
  CWorkerThread *pThis = (CWorkerThread*)pParam;

  // Pointer to parent COM object
  ISimpleCallback *pSimple;

  // init apartment model for this thread
  hr = CoInitialize(0);

  // Get marshaled interface from stream
  hr = CoGetInterfaceAndReleaseStream(
         pThis->m_pStream,
         IID_ISimpleCallback,
         (void**)&pSimple);

  // DO SOME REAL PROCESSING HERE!
  // Spoof processing with a sleep
  Sleep( pThis->m_lWait * 1000);

  // Signal client that processing is done.
  hr = pSimple->Now( pThis->m_lWait );

  // Note: This pointer will be
  // marshaled from this worker thread
  // back to the main server thread.
```

```
    // The actual Now() method
    // gets called from the main server thread.
    // Shutdown com on this thread
    CoUninitialize();

    // Delete CWorkerThread object
    delete pThis;

    // ThreadProcs usually return 0
    return 0;
}
```

The first thing the thread does is extract a pointer from the startup parameter. Remember, this is a static method, and it doesn't have a "this" pointer. To work around this, we've passed in a pointer to a CWorkerThread object that was previously instantiated (on the other thread.) This gives a working context.

```
// Get the object pointer we passed
// in to AfxBeginThread.
CWorkerThread *pThis = (CWorkerThread*)pParam;
```

Next, we need to extract information from that object. The first thing we're going to use is the IStream interface that will marshal our callback COM interface. CoGetInterfaceAndReleaseStream does exactly what its name implies: it extracts the ISimpleCallback interface from the stream, and cleans up the stream. The end result of this call is a usable ISimpleCallback interface pointer.

```
hr = CoInitialize(0);
// Get marshaled interface from stream
hr = CoGetInterfaceAndReleaseStream(
        pThis->m_pStream,
        IID_ISimpleCallback,
        (void**)&pSimple);
```

The COM interface ISimpleCallback is safely marshaled between threads. We can call its methods without fear of threading problems.

Now, we get to the actual processing step of the worker thread. Because this is an example program, there isn't any real processing. To simulate a time consuming operation, we're going to waste some time with a Sleep.

```
Sleep( pThis->m_lWait * 1000);
```

Once this wait is finished, the worker thread is ready to kill itself. Before we exit, however, we need to tell the client program we're finished. This is done by calling the familiar Now() method.

```
hr = pSimple->Now( pThis->m_lWait );
```

The ISimpleCallback interface was marshaled to the original thread, so it will be executed on the server's original thread. We need to do this, because that main thread owns the client's ICallBack interface. If we tried to call the Awake method directly, bad things might happen. Instead of dealing with Awake directly, we're letting the Now() method handle it on the original server object.

All Good Threads Eventually Die

What remains is just cleanup code. We close COM, delete the worker thread object and exit the thread. At this point we've finished implementing our worker thread.

```
CoUninitialize();
delete pThis;
return 0;
```

Summary

Normally, COM interfaces are one-directional and synchronous. More sophisticated programs are going to have to move beyond this model. If you're going to use COM to establish two-way communication between client and server, you're going to have

to deal with callbacks. The other alternative, Connection Points, is really just a specialization of callbacks.

Implementing callbacks may seem unnecessarily complicated - and it probably is. To effectively implement callbacks, you have to have a basic understanding of threading models and marshaling. Most of us are interested in building applications, not the minutia of marshaling.

T H I R T E E N

Connection Points

In the previous chapter, we built a client and server program that demonstrated bi-directional, or callback, interfaces. Connection points are really just a special type of callback interface. Actually, for many applications, callbacks are the preferred type of bi-directional interface.

If you skipped the chapter on callbacks, consider going back and reading it. Most of the background on connection points is covered in the callback chapter.

What makes connection points special is the fact that they offer a standardized technique and a set of interfaces for two-way communications. Connection points aren't so much a single entity as they are a set of interlocking interfaces.

The main advantages of connection points over callbacks are standardization and flexibility. In the OLE world, many types of objects expect an implementation of connection points. An example of this is IQuickActivate interface, which requires IPropertyNotifySink as a sink interface. These objects need to communicate back to their clients.

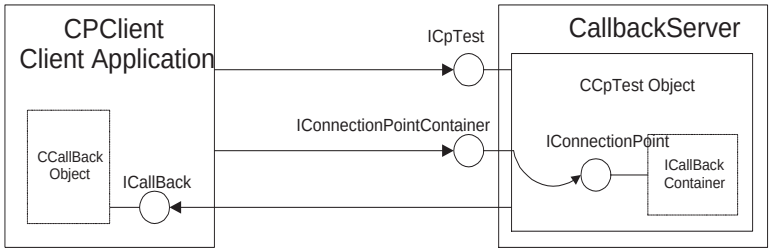

Figure 13–1 Configuration of a server and client using connection points

Connection points offer flexibility in their implementation. A server can have numerous client (sinks) attached, or a single client can have numerous servers. Connection points work well with either configuration. If your server design needs this flexibility, connection points may be a good choice.

Here's a list of connection point classes and interfaces we'll be using in the example.

Interface or class	Where	Description
IConnectionPointContainerImpl	Server	ATL class to manage a collection of connection points. The client will use this interface to find the connection point it needs.
IConnectionPointImpl	Server	ATL class to implement the connectable object on the server. This class allows the client to register (Advise) and un-register (UnAdvise) its sink objects. A COM object may use this template to implement multiple Connection Points.

CCallBack	Client	The callback object implemented by the client. This is a user-defined interface that the sever can call to notify it of important events.
ICallBack	Client	The callback interface.
CCpTest	Server	Our user-defined ATL object on the server. This object implements connection points.
ICpTest	Server	The interface of the CCpTest object.
_ICpTestEvents	Server	The connection points class created by the ATL object wizard, but not used. We used ICallBack instead.

Table 13.1 Connection point classes and interfaces

Description here

You'll notice that the ATL classes are named like interfaces. Normally we would expect anything starting with an "I" to be an interface, which is just a definition. The ATL code for these interfaces will provide a full implementation of the interface.

Modifying the Callback Server

Rather than writing a separate project to demonstrate connection points, we're going to modify the example programs from the previous chapter. Connection points and callbacks are so similar that we can re-use most of this example, while adding only those parts necessary for connection points.

We're going to use the same server we used for the callback example. Open the CallbackServer project and do the following:

1. Insert a new ATL object using the "Insert/New ATL Object" menu
2. Name the new object CpTest
3. Select Apartment Threading

4. Select Custom Interface
5. Aggregation doesn't matter
6. Check the "Support Connection Points" check box
7. Press the OK button and add the object

Selecting the "Support Connection Points" box added several additional lines of code to the object definition. The server class is defined in the file CpTest.h. -- look in this file for the definitions added by the wizard. Here's the class definition:

```
class ATL_NO_VTABLE CCpTest :
public CComObjectRootEx<CComSingleThreadModel>,
public CComCoClass<CCpTest, &CLSID_CpTest>,
public IConnectionPointContainerImpl<CCpTest>,
public IConnectionPoinImpl<CCpTest,&IID_ICallBack>,
public ICpTest
...
```

The ATL template class IConnectionPointContainerImpl was included in the multiple inheritance of CCpTest. This class is a container to manage a list of connection points. You can use IConnectionPointContainerImpl to find a specific connection point attached to the server.

The wizard also added the container object to the COM map of CCpTest. The other interface in this map is, of course, the ICpTest interface.

```
BEGIN_COM_MAP(CCpTest)
COM_INTERFACE_ENTRY(ICpTest)
COM_INTERFACE_ENTRY(IConnectionPointContainer)
END_COM_MAP()
```

The ATL Object wizard added a Connection Point Map to the object. Initially, the map is empty. We will add entries to it later. A server object can support numerous different connection point types. This means a single server object can support connection points to many different types of client sink objects. These will be listed in the connection point map.

The wizard also added the actual connection point to the class inheritance. Each connection point object is explicitly tied to a sink interface on the client. In this case, we're going to use the ICallBack interface. This is exactly the same interface we used for the callback example, has already been implemented by the client.

The wizard doesn't add everything we need. We're going to add the individual connection points to the object. Much of this code is just boilerplate. We will explain it briefly, but the only way to understand it is to see how it all fits together.

The actual connection point class is an ATL template IConnectionPointImpl.

```
public IConnectionPointImpl<CCpTest,&IID_ICallBack>,
```

The client sink interface we just added must also be put in the object's connection point map. This allows the container (IConnectionPointContainer) object to use the callback. The map needs the GUID of the interface on the client.

```
BEGIN_CONNECTION_POINT_MAP(CCpTest)
CONNECTION_POINT_ENTRY( IID_ICallBack )
END_CONNECTION_POINT_MAP()
```

The last thing we need to add is the test methods. This isn't part of the actual connection point set up, but we'll need it for the demonstration. We will add them as two standard COM methods to CCpTest. We will add the MIDL definition, and the definition to the header file.

Thefollowing lines go in the definition of ICpTest interface (in the file CallbackServer.IDL). You can use either the "Add Method" from the class view, or type it directly into the IDL:

```
HRESULT Now2([in] long lCode);
HRESULT Later2([in] long lSeconds);
```

Each method has one parameter - it will be called by the client to exercise the connection points we are implementing. The

last step is to put the matching definition the C++ header (CpTest.H).

```
public:
STDMETHOD(Later2)(/*[in]*/ long lSeconds);
STDMETHOD(Now2)(/*[in]*/ long lCode);
```

Here is the completed listing, with the required objects inserted. The new code for connection points is in bold.

```
////////////////////////////////////////////
class ATL_NO_VTABLE CCpTest :
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<CCpTest, &CLSID_CpTest>,
  public IConnectionPointContainerImpl<CCpTest>,
  public IConnectionPointImpl<CCpTest,&IID_ICallBack>,
  public ICpTest
{
public:
  CCpTest()
  {
  }
DECLARE_REGISTRY_RESOURCEID(IDR_CPTEST)

DECLARE_PROTECT_FINAL_CONSTRUCT()

BEGIN_COM_MAP(CCpTest)
  COM_INTERFACE_ENTRY(ICpTest)
  COM_INTERFACE_ENTRY(IConnectionPointContainer)
END_COM_MAP()

BEGIN_CONNECTION_POINT_MAP(CCpTest)
  // Client callback (sink) object
  CONNECTION_POINT_ENTRY( IID_ICallBack )
END_CONNECTION_POINT_MAP()

// ICpTest
public:
  STDMETHOD(Later2)(/*[in]*/ long lSeconds);
  STDMETHOD(Now2)(/*[in]*/ long lCode);
};
```

The implementation of Now2() will be covered a little later. This method is going to be quite different from its equivalent in the callback test. The Later2() method will be functionally identical to the callback example. It will only be necessary to change the name of the interface from ISimpleCallback to ICpTest. After the client has been explained, we will cover this code.

Now we have the infrastructure for the connection points on the server. Most of it was added by clicking the "Support Connection Points" option in the ATL wizard. Note that the wizard also added the following interface to the IDL code:

```
dispinterface _ICpTestEvents
{
  properties:
  methods:
};
```

We're not going to use this interface in our example. This is the suggested name for the callback interface that the connection points will support. We are going to substitute our ICallBack interface. The wizard also added the following code to the definition of the CpTest object in the IDL code:

```
[default, source] interface _ICpTestEvents;
```

Replace _ICpTestEvents with the ICallBack interface. The code should now look like this:

```
[
  uuid(A47ED662-5531-11D2-85DA-004095424D9A),
  helpstring("CpTest Class")
]
coclass CpTest
{
  [default] interface ICpTest;
  [default, source] interface ICallBack;
};
```

The "[source]" attribute in the IDL code tells COM that the CpTest coclass is the source of ICallBack events. In other words, this object will be calling the client's ICallBack interface. The source keyword doesn't seem to have any actual effect on the behavior of the interface.

Note: Build the server, and don't forget to build the Proxy/Stub DLL. Use the BuildMe.bat file to automate this task. We'll add the Now2 and Later2 implementation later, but the MIDL code won't change. The test client requires these MIDL generated headers.

Adding Connection Points to the Client Program

The connection point client program is going to be very similar to the callback client. You can either modify the existing CallbackClient project, or create a new project named CpClient. I've added a new project, and cloned much of the code from the callback example.

1. Create a new MFC project.
2. Choose a Dialog Application

Now, edit the dialog to look like this:

Figure 13–2 The sample application is an extremely simple dialog

Add the following controls and events through the Class-Wizard:

1. Name the two buttons IDC_BUTTON_NOW, and IDC_BUTTON_LATER.
2. Attach the methods OnButtonNow() and OnButtonLater().

This dialog is functionally identical to the callback example.

Add the Callback Object to the Client

Add the callback object to this project. This object is identical to the callback object in the CallbackServer project. Cut and paste the definition of CCallBack into the CPClient.cpp source file. Also remember to include the "CallbackServer_i.c", and "h" file from the server.

Note that there is absolutely no difference between the callback object used for callbacks, and for connection points. This sink object will behave identically, and will be called by the server (source) in an identical way.

Modifying the CpClient Application

We're now going to add initialization and shutdown code to the main application class, CCpClientApp.

```
BOOL CCPClientApp::InitInstance()
{
  AfxEnableControlContainer();

  InitCOM();
  // Standard initialization
...
```

Add the InitCOM function to the class header. Enter the following code. Note that this is now identical to the CallbackClient application. We're adding the InitCP method, instead of calling Advise directly.

```
BOOL CCPClientApp::InitCOM()
{
```

```
    HRESULT hr;
    CoInitialize(0); // Initialize COM

    // Initialize the main ATL object
    _Module.Init( ObjectMap, 0 );

    // Create a server object
    m_pCP = NULL;
    hr = CoCreateInstance( CLSID_CpTest,
        0, CLSCTX_SERVER,IID_ICpTest,
        (void**)&m_pCP );
    ASSERT( SUCCEEDED(hr) );
    if (SUCCEEDED(hr))
    {
      // Create a callback object
      CComObject<CCallBack>* pCallBack = NULL;
      CComObject<CCallBack>::CreateInstance(
          &pCallBack );
      pCallBack->AddRef();

      InitCP( pCallBack );

      // Done with our ref count. Server did an AddRef
      pCallBack->Release();
    }
    return SUCCEEDED(hr);
  }
```

Initializing connection points is going to take some extra code, so we've isolated it in a separate method. I've covered the rest of this code in the previous chapter.

Registering With the Server's Connection Point Interface

We're now going to interrogate the server COM object for information about its connection points implementation. The InitCP method was designed to do double duty. It is able to both register and unregister with the server's connection point interfaces.

This method will be called both from InitCOM, and from ExitInstance; ExitInstance will pass a NULL pCallBack pointer. InitCP is a new method so you must add the definition to the CCpClientApp class (in CpClient.h).

```
HRESULT CCPClientApp::InitCP(IUnknown* pCallBack)
{
  HRESULT hr;
  IConnectionPointContainer *pConnPtCont;
  IConnectionPoint * pConnPt;

  // Get a pointer to the
  // connection point manager object
  hr = m_pCP->QueryInterface(
      IID_IConnectionPointContainer,
      (void**)&pConnPtCont);
  ASSERT( SUCCEEDED(hr) ); // crash if failed
  if (SUCCEEDED(hr))
  {
    // This method is the QueryInterface
    // equivalent for an outgoing
    // interfaces. See if the server supports
    // connection points to our callback interface
    hr = pConnPtCont->FindConnectionPoint(
      IID_ICallBack, &pConnPt);
    ASSERT( SUCCEEDED(hr) ); // crash if failed

    // Release the container object
    pConnPtCont->Release();

    if (SUCCEEDED(hr))
    {
      // Register the Connection Point
      if (pCallBack != NULL)
      {
        // Establish connection between
        // server and callback object
        hr = pConnPt->Advise(pCallBack, &m_lCookie);
      }
      else // Remove the Connection Point
```

```
      {
        // Remove connection
        hr = pConnPt->Unadvise(m_lCookie);
      }

    // Release connection point object
    pConnPt->Release();
    }
  }
return hr;
}
```

We start the function by getting a pointer to the server's IConnectionPointContainer interface. This interface points to the object that the server uses to keep track of its connection points. Since we already have a pointer to the ICpTest interface, we can use QueryInterface().

```
IConnectionPointContainer *pConnPtCont;
// Get a pointer to the connection
// point manager object
hr = m_pCP->QueryInterface(
        IID_IConnectionPointContainer,
        (void**)&pConnPtCont);
```

Now we can ask the connection point container for a specific type of connection point. In this case, we want one that handles the ICallBack interface that our client implements. Calling FindConnectionPoint() on the container will give us the callback interface. Once we have the connection point object, we're done with the container, so it is released. Since we wrote the server object, we can be pretty sure it supports the ICallBack callback interface.

```
hr = pConnPtCont->FindConnectionPoint(
    IID_ICallBack, &pConnPt);

// Release the container object
pConnPtCont->Release();
```

If we call InitCP with an ICallBack interface pointer, we are registering the sink object with the server. If a NULL pointer is passed in, the sink object will be un-registered. Calling Advise() on the server object registers the sink object. Advise is implemented in the ATL class IConnectionPointImpl. It is very similar to the Advise() method we wrote for our custom callback. On the server, Advise makes a copy of the sink interface, and returns a unique cookie to identify it. Once Advise has been called, we can release the sink object we passed it.

```
if (pCallBack != NULL)
{
  // Establish connection between
  // server and callback object
  hr = pConnPt->Advise(pCallBack, &m_lCookie);
}
```

The mirror image method for Advise() is Unadvise(). This method will remove the sink object from the server's list of connection points. Unadvise() checks the cookie, and terminates the connection. This code will be called when InitCP is called from ExitInstance.

```
else // Remove the Connection Point
{
  // Remove connection
  hr = pConnPt->Unadvise(m_lCookie);
}
```

Add the ExitInstance method to the CPClientApp application. This method is called when the application shuts down:

```
int CCPClientApp::ExitInstance()
{
  // If we have a server object, release it
  if (m_pCP != NULL)
  {
    // Remove servers callback connection
    InitCP(NULL);
    // Release the server object
```

```
    m_pCP->Release();
  }
  // Shut down this COM apartment
  CoUninitialize();
  return 0;
}
```

Adding the Now and Later Buttons

Enter the following code for the Now and Later buttons. This code is functionally identical to the CallbackClient program. It is added to the application's main dialog class, CCPClientDlg.

```
void CCPClientDlg::OnButtonNow()
{
  HRESULT hr;
  CCPClientApp *pApp = (CCPClientApp*)AfxGetApp();
  hr = pApp->m_pCP->Now2(1);
  if (!SUCCEEDED(hr)) AfxMessageBox( "Call Failed" );
}
void CCPClientDlg::OnButtonLater()
{
  HRESULT hr;
  CCPClientApp *pApp = (CCPClientApp*)AfxGetApp();
  hr = pApp->m_pCP->Later2(5);
  if (!SUCCEEDED(hr)) AfxMessageBox( "Call Failed" );
}
```

We've now completed the client application. Let's go back and implement the Now2() method on the server.

Using the Connection Point - the Server Side

So far, on the server, we've added a connection point map and a connection point object. Just adding these objects really doesn't give much insight on how to use them.

At some point in the execution of the server, it will need to make a call back to the client. Normally, this will be in response to some triggering event. Here's the implementation code.

```
STDMETHODIMP CCpTest::Now2(long lCode)
{
  HRESULT hr = S_FALSE;

  // Lock the object
  Lock();

  // Get first element in CComDynamicUnkArray.
  // m_vec is a member of IConnectionPointImpl
  IUnknown** pp = m_vec.begin();
  ICallBack* pICp = (ICallBack*)*pp;
  if (pICp)
  {
    // Call method on client
    hr = pICp->Awake( lCode );
  }
  // Unlock the object
  Unlock();

  return hr;
}
```

The first thing this method does is lock the COM object. This takes ownership of the critical section protecting the module. Lock is implemented in the ATL CComObjectRootEx base class. Lock is paired with the Unlock() at the end of the method.

Our COM object inherited from the ATL class IConnectionPointImpl, which contains a list of connection points. The variable m_vec is of type CComDynamicUnkArray, which holds a dynamically allocated array of IUnknown pointers.

Before this method is called, the client has done a considerable amount of set-up. Recall that the client called Advise() on the connection point container. When Advise() was executed, it made a copy of the client sink interface. This is the interface saved in m_vec.

Because we only have one connected client sink, we get the first pointer in the m_vec array. We call Awake() on the client sink object. The result of this is that Awake() gets called on the client process, causing a message box to display. Not a very impressive result for all the work we've had to do.

Adding the Later2 Method

The implementation of the Later2 method is identical to the Later method in the ISimpleCallback object. Just cut-and-paste this code, changing only the name of the interface. The worker thread will behave identically. When the worker thread calls the Now2 method, it will properly navigate the connection point map. If you had registered multiple callback interfaces, you would iterate through the m_vec collection.

Summary

The implementation of our connection point example was mostly a cut-and-paste modification of the callback example. The few lines of code that are different handle the navigation of the ATL connection point container classes.

One reason to implement connection points is the fact that you are working with OLE clients (such IQuickActivate with IPropertyNotifySink). Or, if you are handling multiple sink (callback) objects, connection points may make your life easier. Both callbacks and connection points do approximately the same thing, and implementing one or the other can add a lot of functionality to your servers.

F O U R T E E N

Distributed COM

So far we haven't ventured very far away from our computer. All the COM examples so far have been for clients and servers running on the same machine. In this section we'll discuss how to extend our range into the area of DCOM and distributed computing.

There is some good and bad news here. The good news is that converting from COM to DCOM is easy. The bad news: there are many more things that can go wrong. Foremost among these problems are security issues.

An Overview of Remote Connections

Most of the differences between COM and DCOM are hidden from the developer. For example, local COM uses LPCs (Local Procedure Calls), and DCOM uses RPCs (Remote Procedure Calls). As a programmer you would never notice the difference, except that RPCs are slower. There's also a whole new level of security and remote activation going on. There are only a few things in your program you'll need to change.

Like all COM communication, everything starts when the client requests an interface from a server. In DCOM, the client calls

CoCreateInstanceEx(), passing in a description of the server computer, and requesting a CLSID and Interface.

This request is handled by the Service Control Manager (SCM), which is a part of Windows. The SCM is responsible for the creation and activation of the COM object on the server computer. In the case of DCOM, the SCM will attempt to create the object on the remote computer.

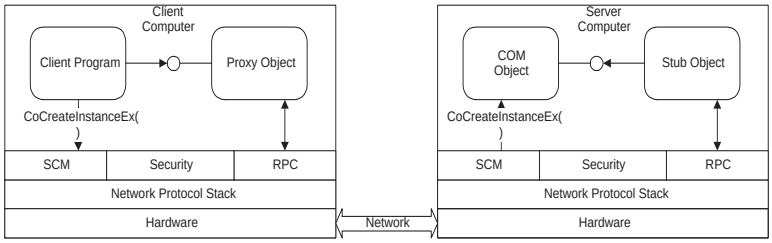

Figure 14–1 Components of clients and servers in when using Distributed COM

Once the remote COM object has been created, all calls will be marshaled through the Proxy and Stub objects. The proxy and stub communicate using RPCs (Remote Procedure Calls) as a mechanism. RPCs will handle all the network interaction. On the server side, marshaling is taken care of by the stub object.

The transmittal of data across the network is taken care of by RPCs. RPCs can run on a number of protocols, including TCP/IP, UDP, NetBEUI, NetBIOS, and named pipes. The standard RPC protocol is UDP (User Datagram Protocol). UDP is a connectionless protocol, which seems like a bad fit for a connection-oriented system like DCOM. This isn't a problem however, because RPCs automatically take care of connections.

At the time of writing, only TCP/IP was available on Windows 95. This can be an annoying limitation, requiring you to install TCP/IP on all Windows 95 systems, even when other network protocols are available.

Perhaps the single most frustrating aspect of DCOM is security. Windows 95/98 doesn't have enough security, while Windows NT seems to have too much. As always, NT security is a complex and specialized field. There are various levels and layers of security. Our examples will only cover the most basic uses. On a large network, it's almost guaranteed that you'll spend time handling security issues for your distributed applications.

Converting a Client for Remote Access

There are two ways to connect to a remote server. You can make slight changes to your program, or you can change the server registration. Of these two, changing the program is the better choice. Once the program is converted to work remotely, it will work locally without any changes.

Changing the server registration is also a possibility. You can put the remote connection in the registry, and COM will automatically make the connection. We'll cover this topic later.

There's very little programming required to make a client work with remote connections. When you create the remote COM object you need to specify a COSERVERINFO structure. You'll notice that CoCreateInstance() doesn't have a place for this structure, so you'll have to use CoCreateInstanceEx() instead.

The COSERVERINFO structure should be set to zero, except for the pwszName member, which points to the server name. This isn't as easy as it may seem. The pwszName member is a wide character (UNICODE) string. If you're not already using wide characters, you'll need to convert a string to wide characters. There are a number of ways to do this:

- Use the mbtowc() function. This string converts a multibyte (char*) string to a wide string.
- Use the CString.AllocSysString() method.
- Use the SysAllocString and SysFreeString API.

Here is one way to accomplish this conversion:

```
CString strServer = "ComputerX";
```

```
// Remote server info
COSERVERINFO cs;
// Init structures to zero
memset(&cs, 0, sizeof(cs));
// Allocate the server name in
// the COSERVERINFO structure
cs.pwszName = strServer.AllocSysString();
```

The server name is usually going to be a standard UNC (universal naming convention) name. This would take the form of "server", or "\\server". You can also use DNS names, with the format of "www.someserver.com", or "server.com". A third option is to specify a TCP/IP address here, e.g. "123.55.5.0". This name will have to be compatible with your network transport.

The CoCreateInstanceEx() function takes different parameters than its precursor, CoCreateInstance(). Specifically, this extended function takes the COSERVERINFO as its 4th argument. You can still use this call for local connections. Just pass in a NULL for the COSERVERINFO pointer.

Perhaps the most interesting difference is the last two parameters.

For remote connections we obtain interface pointers a little differently than we do for local connections. CoCreateInstanceEx() takes an array of MULTI_QI structures instead of a plain IUnknown pointer. The MULTI_QI structure receives an array of interface pointers. This is done to reduce the number of calls to CoCreateInstance() across the network. The designers of DCOM did this in recognition of the fact that network performance can be slow.

The MULTI_QI structure has the following members:

```
typedef struct _MULTI_QI {
const IID* pIID; // Pointer to an interface identifier
IUnknown * pItf; // Returned interface pointer
HRESULT hr; // Result of the operation
} MULTI_QI;
```

You pass in an array of these structures. Each element of the array is given an Interface ID (IID) of an interface. If the function

succeeds, you'll get back a pointer to the interface in **pItf**. If there is an error, the **hr** member will receive the error code.

Here's how to initialize the MULTI_QI structure. You can make the array any size required (often it is just one element long):

```
MULTI_QI qi[2]; // Create an array of 2 structures
memset(&qi, 0, sizeof(qi)); // zero the whole array
qi[0].pIID = &IID_IinterfaceX // add an interface
qi[1].pIID = &IID_IinterfaceY; // add another
```

You pass both these structures along with the usual parameters. CoCreateInstanceEx() also needs the length of the MULTI_QI array, and a pointer to the first element.

```
// Create a server COM object on the server.
HRESULT hr = CoCreateInstanceEx(CLSID_CMyServer, NULL,
            CLSCTX_SERVER, &ServerInfo, 2, qi);
// check the qi codes
if (SUCCEEDED(hr))
{
  // also check qi hresult
  hr = qi[0].hr;
}
if (SUCCEEDED(hr))
{
  // extract interface pointers from
  // MULTI_QI structure
  m_pComServer = (ICpServer*)qi[0].pItf;
}
```

We have more than one COM status to check. CoCreateInstanceEx() returns a status like every COM library call. We also need to check the status of each element in the MULTI_QI array. The server may return different statuses, depending on weather the requested interface is supported. You'll have to check the hr member of each MULTI_QI element.

If the status is OK, the interface pointer can be extracted from the array. The pItf member will contain a valid interface pointer. This interface can now be used normally.

Once the connection has been established, there are no differences between COM and DCOM. All the DCOM extensions work equally well for local connections. You'll hear this referred as Local/Remote Transparency. This is one of the most powerful features of COM.

Adding Security

Once you start connecting to the outside world, you will quickly run into a multitude of security issues. Security is an area where there are significant differences between Windows NT and Windows 95/98. In general, NT provides a rich and bewildering set of security options. Window 95/98 on the other hand, provides the bare minimum required to exist on a network. Many of the concepts that follow apply primarily to Window NT. Window 95 is inherently insecure.

Generally COM has reasonable defaults for most security settings. By using DCOMCNFG and some basic settings, you can get most client/server systems to run. If you need a high level of security, you'll need to delve into the many levels of COM security. Here we will cover the basic tools used to control COM security to help get you started.See the error-handling appendix for further details.

Security Concepts

DCOM has numerous levels of security. Many of DCOM's security features are borrowed from other subsystems. RPCs provide the basis for COM security, and many of the concepts used come directly from RPCs.

The most basic security level of DCOM is provided by the network. Most networks provide some level of login security. If the local area network is a Windows NT domain, for example, network logins are managed and restricted by the domain controller. Having a secure network environment goes a long way towards making the DCOM environment secure.

Of course, some networks must provide relatively open access to users. If you provide access to guest accounts, other domains or a large user community, things are going to be wide open. It's only a matter of time before someone starts hacking your systems.

There is also some basic network security. Most networks check to ensure that all data packets are legitimate. This means the network may filter out altered or damaged network traffic. This also adds a significant level of security to DCOM.

Access Permissions

DCOM runs on top of RCPs, and inherits much of its security from the RPC mechanism. Fortunately, RPCs have been around for quite awhile and have developed a good set of security tools. Much of what follows is actually RPC-based security that has been piggybacked into DCOM.

Access permission security determines if a particular user has access to your COM application. Access security checking is done for the entire application (or process). Depending on the object, you may allow only certain users to have access, or you can deny access to particular users. For a Windows NT domain, the administrator has probably set up special users and groups, otherwise you'll get the default groups.

DCOM gets a security descriptor from the registry. This value is stored as a binary array under the AppID key. DCOM checks this value against that of the caller.

```
[HKEY_CLASSES_ROOT \AppID{<AppID>}]
"AccessPermission" = hex: Security ID
```

DCOM sets up defaults for access security. If an application requires more security, it can drill down into more sophisticated security implementation. Checking can be done for the object, for the method call, and even for the individual parameters of the method call.

What is a Security ID?.....

A security ID is a unique number that identifies a logged-on user. The SID can also represent groups of users, such has Administrators, Backup Operators, and Guests. This SID is unique and is valid on the local system and the network (provided there is a domain controller controlling the network). The system always uses the SID to represent a user instead of a user name.

You can either allow or deny permission to any user, or group of users.

Windows 95/98 has very weak user level security. For Windows 95/98, the user information will be provided by some other system on the network. Usually this would be the domain controller.

If you try to connect to a server without sufficient access permission, you will probably get the "Access Denied" error.

Launch Permissions

Launch Security determines if a caller can create a new COM object in a new process. Once the server has been launched, this permission does not apply.

When a client requests a new COM object, it does not create it directly. COM itself is responsible for the creation of the object. Before it creates the object, it checks to see if the caller has permission to do so.

Because DCOM allows remote activation, any computer on the network can try to start a server. Your COM objects are potentially vulnerable to anyone on your network. Good security practices require that COM doesn't even start the object if the caller doesn't have permission to use it.

Launch permission for an application is defined in the AppID key.

```
[HKEY_CLASSES_ROOT \AppID{<AppID>}]
"LaunchPermission" = hex: Security ID
```

Windows 95/98 doesn't have the user level security features to control the launch of an object. Because of this limitation, Windows 95/98 doesn't even try to launch remote applications. This means that the server must already be running on a Windows 95/98 system.

You can pre-start the server application interactively. This will run the server as the desktop user. The remote object can then connect to the object. Unfortunately, when the remote user disconnects, if it is the only connected client Windows 95/98 will shut down the server. The next time the remote user tries to connect, they will get an error because the server won't re-start itself.

The work-around is pretty simple. You can write a bare-bones client that connects to the server locally from the Windows 95/98 computer. As long as this client is connected, the server will have an active reference count and will remain available. You can put this program in the startup menu, thus making the server available as long as somebody is logged into the desktop. Windows NT has no such restrictions.

Authentication

Authentication means confirming that the client or server are who they claim to be. The subsystem that provides authentication is known as the "authentication-service" provider. There are several authentication services. The default on Windows NT is NT LAN Manager Security Support Provider (NTLMSSP). Another is the DCE authentication service, which is based on the Kerberos standard.

Impersonation

Impersonation occurs when a server assumes the identity of its caller. Although this seems a bit odd at first, it considerably simplifies security issues.

Normally when the server is started, it must log in with a specific username. There are three possibilities:

1. The user who started the server (launching user).
2. The user who is currently logged into the desktop.
3. A specially designated user.

The default is the launching user. For most servers, this makes sense. The server assumes all the privileges of its creator, and thus only has access to what it's supposed to. For servers that have multiple connected users, this approach doesn't work very well. Each of the users may have different security access privileges and needs.

If you specify that the server uses a specific user, this also can cause problems. You must ensure that the server's account has access to everything it needs. More importantly, you must ensure it does not provide access to things it shouldn't.

Impersonation allows the server to temporarily assume the identity of the calling client. This way, it uses the Security ID of the client to access the system. Once the client's operation is complete, it reverts back to its original account. When the next client makes a request, it assumes the Security context of that client also. Impersonation allows the best of both worlds in terms of security.

The benefits for a server are quite clear. The client, however, must be careful about impersonation. By impersonation, the server can gain access to resources that it normally couldn't. A server can impersonate a more privileged client and perform operations from which it would normally be blocked. This is a much more subtle security issue. Most of the time we are concerned about protecting the server from the client.

Identity

DCOM allows you to designate that a server runs as a specific user. Often this is an excellent way to control a server's security access. By running as a user with specific privileges, you can control its access.

Windows NT services default to a special account called "LocalSystem", which has unlimited privileges on the local machine but no network access privileges. If the server does not make use of some form of impersonation, it won't have access to network resources.

Custom Security

Regardless of all the levels of DCOM security, you may want to implement your own. There are numerous ways to implement custom security. Usually this would involve a username and password to access the server.

CoInitializeSecurity

The CoInitializeSecurity() function sets the default security values for a process. It can be used on both the client and server application. This function is invoked once per process; you don't need to call it for each thread. By process, we mean an application program, either a client or COM server. It should be invoked right after CoInitialize(), and before any interfaces are used.

If you don't call CoInitializeSecurity(), it will be automatically invoked by COM. It will be called with the defaults set by DCOMCNFG. The security settings invoked here will override any defined in the registry AppID key.

This function has quite a few parameters. Some apply to COM clients, others to servers. Several of these parameters deserve an entire chapter unto themselves.

```
HRESULT CoInitializeSecurity(
  PSECURITY_DESCRIPTOR pVoid,
  DWORD cAuthSvc,
  SOLE_AUTHENTICATION_SERVICE * asAuthSvc,
  void * pReserved1,
  DWORD dwAuthnLevel,
  DWORD dwImpLevel,
  RPC_AUTH_IDENTITY_HANDLE pAuthInfo,
```

```
DWORD dwCapabilities,
void * pvReserved2);
```

Parameter	Used on	Description
pVoid	both	Points to security descriptor. This parameter is only used on Windows NT. This descriptor is a structure that contains the security information associated with an object. If NULL, no security (ACL) checking is done.
cAuthSvc	server	Count of entries in *asAuthSvc*. A value of -1 tells COM to choose which authentication services to register.
asAuthSvc	server	Array of SOLE_AUTHENTICATION_SERVICE structures.
pReserved1		Not used.
dwAuthnLevel	proxies	The default authentication level.
dwImpLevel	proxies	The default impersonation level.
pAuthInfo		Reserved; must be set to NULL
dwCapabilities	both	Additional client and/or server-side capabilities
pvReserved2		Reserved for future use

Table 14.1 CoInitializeSecurity parameters

Security descriptors are only used on Windows NT. A security descriptor is a structure that contains the security information associated with an object. If NULL is specified, no security (ACL) checking is done.

The next two parameters concern authentication. Authentication is the service used to determine if incoming COM messages are from a known source. There are several authentication packages, including the NT LAN Manager, and Kerberos. These services are automatically handled by RPCs.

The default authentication level is specified for the proxy. The server will reject calls made at a lower authentication level. There hare several possible values, each providing a more comprehensive level of checking. These constants are defined in <RPCDCE.H>. Passing in a value of RPC_C_AUTHN_NONE provides a decent default.

Impersonation allows one process to assume the identity and credentials of another. In this case, the impersonation level determines how much the client trusts the server.

Impersonation Level	Description
RPC_C_IMP_LEVEL_ANONYMOUS	The server object cannot get security information about the client.
RPC_C_IMP_LEVEL_IDENTIFY	The server can get security information, but cannot impersonate the client.
RPC_C_IMP_LEVEL_IMPERSONATE	The server can use the client's security credentials for local operations. Represents a high level of trust.
RPC_C_IMP_LEVEL_DELEGATE	The server can use the client's security credentials for network operations. This level is not supported by many authentication services.

Table 14.2 Impersonation Levels

The *dwCapabilities* flags are used to determine further capabilities of the proxy. These are defined in the EOLE_AUTHENTICATION_CAPABILITIES enumeration in <OBJIDL.IDL>.

If you are somewhat bewildered by all the parameters on the CoInitializeSecurity() call, here are some very perfunctory default values.

```
hr = CoInitializeSecurity(NULL, -1, NULL, NULL,
  RPC_C_AUTHN_LEVEL_NONE,
  RPC_C_IMP_LEVEL_IMPERSONATE,
  NULL,
```

```
    EOAC_NONE,
    NULL);
```

Basically, these settings leave security pretty wide open. If you have real security concerns, you are going to have to research these issues thoroughly and set up acceptable values.

Disconnection

One of the insidious characteristics of networks is that they are fragile. You can expect your client and server to be disconnected for any number of reasons. Perhaps the network had an error, or the server was rebooted, or the client computer crashes. Whatever the cause, your applications have to clean up the results.

Another name for this cleanup is "Garbage Collection." COM implements some simple garbage collection on the COM object level. A COM server is able to detect when clients have been disconnected.

Normally, a client will disconnect gracefully from its server, shutting down its connection in an orderly way. You need to be aware of what happens when it doesn't. Let's examine how a client and server would handle a disconnection.

For the client program, a disconnection is pretty obvious. The client will make a COM call, and the call will return with an error. Chances are that your client will be inclined to crash in one form or another. Whatever the error, the client will have to handle shutting itself down.

Unfortunately, the client won't see the disconnection until it tries a COM call. In many applications, the program may run for some time before it uses COM. One solution to this is to write a "heartbeat" function that checks the server connection periodically.

The server has a different problem: it will never know about the disconnection. Because all COM applications are driven by the client, the server is always waiting for a request. If the client is disconnected, it will stop making requests and the server will remain connected.

If you're ambitious, you can write a server-to-client heartbeat check with a callback. The server would periodically call the client's callback to see if it is alive. Fortunately, in most cases this isn't necessary.

COM implements a type of heartbeat called "Delta Pinging." The "Ping" part of this is obvious. The RPC layer of COM will send out a ping message from client to server every two minutes. A ping is just a small packet of information that indicates the client is connected. If the server fails to get three consecutive ping messages, it disconnects the client and cleans up its outstanding connections. This means it usually takes about seven minutes for a broken client connection to be cleaned up. This is automatic behavior, and you don't have much control over it.

One place for the server to check for a disconnection is in the COM object's destructor. When the COM object is disconnected, its destructor will eventually be called. You can handle custom object cleanup in this code.

Because network operations can be expensive, COM tries to be very efficient about its ping messages. These ping messages are piggybacked onto existing COM calls if possible. This eliminates unnecessary message traffic for active connections.

RPCs also combine all pings from a server before sending them. This means that only one ping message will be sent from one client to its server, even if the client has multiple COM objects. These groups of ping messages are called "ping sets."

Using the Registry for Remote Connections

We've covered some of the programming differences between COM and DCOM. There is another way to connect to remote servers by using registry settings. This is a somewhat crude method, but it is useful when working on legacy applications. For most cases adding remote capabilities to the C++ modules will give more control.

The easiest way to do this is through DCOMCNFG. Select the properties of your COM object and select the "Location" tab.

Using this utility, you can specify the name of a remote computer.

Installing the Server on a Remote Computer

If you want your server to run on the remote computer, you'll need to install it. All you have to do is copy the program (EXE) to the remote computer and register it. Use the -Regserver command. If you have a proxy/stub DLL, you will also have to register that. Use REGSVR32 to register the proxy/stub DLL.

If your server is running Windows 95, be sure DCOM is installed. On NT DCOM installs as part of the operating system, but in Windows 95 it is a separate step.

F I F T E E N

ATL and Compiler Support

COM itself is simple, but for some reason writing COM applications always turns out to be harder than you expected. The demon of that plagues COM is complexity. The only way to tame this complexity is with good programming tools. If you're working with COM, you have three choices:

1. Write C++ SDK programs
2. Use MFC and it's OLE infrastructure
3. Use ATL

C++ SDK Programming

You can write perfectly good COM programs with native C++ and a few of the COM SDK routines. There's just one problem: it takes forever. Most of COM programming is repetitive boilerplate code. In any case, for anything but client programs, it's going to be a lot of work. It's a perfect application for a class or template library. You might as well use MFC or ATL.

MFC COM

MFC offers a viable way to implement COM. Traditionally COM was a substrate of OLE. OLE brings along with it quite a bit of baggage. MFC is designed for writing User Interface programs. MFC offers many powerful features for User Interface programs. Most people writing C++ programs end up using MFC. Unfortunately, the GUI concentration means that MFC isn't a great fit for the server side of COM programming.

When you use the MFC wizards built into Visual C++, you get a great framework on which to base your application. The wizards hide the big problem with MFC, which is also complexity. If you've every tried to do non-standard things with the MFC framework, you quickly find yourself in a morass of unfamiliar and unfriendly code.

The other problem with MFC is size. It's huge. Including MFC in a non-User Interface program adds a lot of overhead. Of course, this isn't always a problem. If your application is already using MFC, linking in the MFC DLL isn't a burden.

Here is a quick summary of the challenges you'll face with MFC

- MFC is large and complex.
- MFC does not easily support dual interfaces.
- MFC does not support free threading. Thread safety is a problem with MFC.

As we saw in Chapter 4, creating a COM client with MFC is straightforward. For COM servers, ATL is the way to go.

ATL - The Choice for Servers

ATL is currently the best choice for developing COM servers. The ATL wizards provided with Visual C++ offer an extremely attractive way to develop server applications. Almost all the server examples in this book use the ATL wizards. Currently there is no tool for COM server development that comes close to ATL.

In addition, ATL supports all threading models. If you want the advantages of free threading you'll probably need to use ATL. Dual Interfaces are another extremely useful feature. With ATL, creating dual interfaces is very easy - it's just a matter of clicking a button in the wizard.

Finally, ATL offers a very small memory footprint. Because ATL is a template library, you aren't linking in a big DLL or library. The ATL templates work with the compiler to generate only the code you need.

That doesn't mean you can't use MFC also. On the simplest level, you can include MFC as a shared DLL, and include the AFX headers in the ATL server. If you want to develop CWinApp-based applications it will take some more work. You'll have to include the standard MFC InitInstance and ExitInstance methods and integrate them with the standard ATL _Module (CComModule).

What's the down side? No question about it - lack of documentation. ATL is a new product, and there's just not that much information published about it. Fortunately this is rapidly changing. Every month, more is being written about this excellent library.

Basic Templates

If you've worked with templates, ATL will make perfect sense. If you've used the standard template library (STL), you'll be right at home. If not, your initial reaction will probably be one of bewilderment. For most C++ programmers templates seem somewhat unnatural,

Templates are a very specialized form of macro with type checking. The 'C' and C++ macro pre-processor allows you do some powerful and sophisticated text substitutions. Unfortunately, macros can be quite cryptic, and worse, they introduce difficult errors into programs. Many of these errors are the result of data type mismatches. Given these difficulties, many C++ programmers cringe whenever they see macros in their source code.

Templates use the same text-substitution technology as macros, but add some extra syntax for type checking. Templates have a more structured environment than traditional pre-processor macros. This eliminates a lot of, but not all of, the problems. Templates can still be extremely cryptic, and debugging them can be difficult.

A Simple Template Example

Take a standard piece of code that swaps two integer values. It's a piece of code that we've all written at one time or another:

```
void swap( int &a, int &b )
{
  int itemp = a;
  a = b;
  b = itemp;
}
```

Here's the call to swap.

```
int a=1;
int b=2;
swap( a, b );
```

This piece of code works only for integers. If we were to pass in a double, we'd get a compiler error. You would have to rewrite the function to take a double & instead of an int &. How would you write this piece of code generically? One easy method would be to use a macro.

```
#define SWAP( t, a, b )  {\
  t temp; \
  temp = a; \
  a = b; \
  b = temp; \
  }
```

Calling this macro would take three parameters; the first one would be the data type.

```
double d1 = 1.1;
double d2 = 2.2;
SWAP( double, d1, d2 );
```

Calling this macro would work with either an int or a double, depending on what we passed in. Actually, this isn't a bad way to write this piece of code. Unfortunately, there's not any type checking going on. That means when you pass in incompatible data types, the compiler will give you very misleading error message, or worse - no error message at all.

Another problem is the ugly syntax. The macro pre-processor wasn't designed to write functions and programs. The #define syntax is difficult to write; it's especially unpleasant to remember all the backslashes.

Templates offer a more type-safe method of doing the same thing. Here's how we'd write the swap routine as a template:

```
template <class T>
void Swap( T & a, T & b )
{
    T temp = a;
    a = b;
    b = temp;
}
```

In templates, the variable "T" usually stands for the substituted data type. We can call Swap with almost any data type:

```
int i1, i2;
CString cs1,cs2;
CmyDataType md1, md2;
Swap( i1, i2 );
Swap( cs1, cs2);
Swap( md1, md2 );
```

The best part of this template is that the compiler will actually give you a meaningful message if you get it wrong. If the

types aren't compatible, the compiler will give you an error message. We should note that the template definition is readable code. The angle brackets do take some adjustment.

Template Classes

Template functions are actually one of the simpler things we can do with templates. The real power of ATL comes in defining classes. ATL makes heavy use of template classes. The syntax of defining a class template is very similar to the function template above.

```
template <class T>
class PrintClass
{
public:
    T m_data;
public:
    void Set( T a ) { m_data = a; };
    void Print() { cout << m_data << "\n"; };
};
```

We can use the class with any data type that is compatible with "cout". This is a trivial example, but you can begin to see the potential of templates.

```
PrintClass<int> x;
x.Set( 101 );
x.Print();
```

One of the characteristics of ATL is multiple inheritance. Most ATL COM classes created by the ATL class wizard are built around multiple inheritance.

Here's one of the headers generated by the ATL wizard.

```
class ATL_NO_VTABLE CBasicTypes :
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<CBasicTypes, &CLSID_BasicTypes>,
  public IBasicTypes
```

Notice that this coclass is implemented by the ATL templates CComObjectRootEx and CComCoClass. The CComObjectRootEx template handles reference counting for the COM object, CComCoClass implements the COM class factory. The third inherited class, IBasicTypes, is an interface, which is a plain C++ base class (with a COM VTABLE layout)

Template code can get extremely ugly. Because of its terseness, template code is hard to follow. Take the following example:

```
typedef CComObject<CComEnum<IEnumString,
&IID_IEnumString,
  LPOLESTR, _Copy<LPOLESTR> > > IEnumObject;
IEnumObject* pNewEnum = NULL;
```

This looks like an entry in the obfuscated C++ code contest - templates nested three deep! This example declares an ATL enumeration interface. (The three template classes here are CComObject<>, CComEnum<>, and _Copy<>.)

There are several interesting things going on here. A peculiar aspect of templates is how they handle **typedef** statements. No code is generated until an actual object is declared. Another oddity is the placement of angle brackets. The critical difference here is between ">>" and "> >". The former is the stream operator, the latter is the end of a nested template definition. If you forget the space between angle brackets, you'll get some interesting compiler errors.

The fundamental ATL classes include:

ATL Class	Template Argument	Description
CComObjectRoot	Your class.	Implements the methods of IUnknown. This class gives you QueryInterface, AddRef, and Release. Works for non-aggregated classes. Uses single threading model.
CComObjectRootEx	ThreadModel. Use one of the threading model classes.	Handles the reference counting for the object. ATL objects must be based on CComObjectRoot or CComObjectRootEx.
CComCoClass	Your Class and a pointer to the CLSID to the object.	Defines the object's default class factory and aggregation model.
CComSingleThreadModel, CComMultiThreadModel		Single and multi-treading models.
IDispatchImpl	Your class, the IID, and LIBID.	IDispatch implementation for dual interfaces.
CComPtr	Interface	Implements a smart pointer to manage an interface.
CComQIPtr	Interface, IID of interface	Implements a smart pointer to manage an interface. Allows querying of interfaces.
CComAggObject	Contained class	Implements IUnknown for an aggregated object.

Table 15.1 Fundamental ATL Classes

Native Compiler Directives

One of the most important recent changes to COM was the addition of native Visual C++ compiler directives. By native, we mean that these commands can be included directly into your C++ source code, and the compiler will recognize them. This native support is oriented towards the client program. This is an interesting step towards making COM programming a lot easier.

The #IMPORT Directive

The import statement allows the compiler and pre-processor to use a type library to resolve certain types of COM references. This information is converted into C++, making it easily available to the application. Type libraries have a tremendous amount of useful COM information in them. This includes Class Identifiers (CLSID), Interface ID's, and especially interface definitions.

Traditionally the only way to get this information was through include files. MIDL generates the C++ headers for these definitions, but you have to locate and include the proper headers. While not especially difficult, this step is tedious and prone to failure when header files are moved or changed.

As useful as these changes are, they aren't really anything new. Many languages that support COM have had this feature for years. Visual Basic has a component browser that does much the same thing.

```
//#import "filename" [attributes]
#import <test.lib> no_namespace
```

The syntax is quite simple. There are however, quite a number of attributes for the import statement. The only one you'll commonly see is "no_namespace". We'll discuss namespaces shortly.

Like the C++ include statement, the #import directive can use either angle brackets "< >" or double quotes. Like the include statement, the choice affects the search order of directories for the type library. Angle brackets will search the "PATH" and "LIB" environment variables first, and lastly check the com-

piler include path option ("/I"). Double quotes will search the current directory first, followed by the paths shown above.

Type libraries aren't the only way to get this type information. EXE and DLL files can also contain type libraries. You can also specify these types of file in the import directive.

Namespace Declarations

Name spaces in C++ are used to prevent name conflicts with variables, types, and functions. By default, the import directive puts all its generated code in a C++ "namespace." The namespace used is based on the type library name. If the typelibrary was "TEST.TLB" the namespace would be "TESTLib". To use data declared in a namespace, you have to prefix everything with the namespace name. Here's an example of a simple namespace.

```
namespace MySpace {
    typedef int SPECIAL;
}
MySpace::SPECIAL x;
```

If you leave off the "no_namespace" attribute for the import statement, you'll have to prefix all the import generated declarations with a namespace. One of the things the import statement does is define "smart pointers" for all of the interfaces in the library. If we create an enumeration in MIDL called RGB_ENUM, we would have access to it through the import statement. Our client program could refer to this enumeration, but it would have to prefix it with the type library namespace.

```
#import "TEST.TLB"

// The compiler will give an error here:
RGB_ENUM BadRgbEnum;

// This works.
TESTLib::RGB_ENUM  RgbVal;
```

If you don't want to mess with namespaces, you import with the "no_namespace" attribute. This allows you to use the MIDL names directly. Of course if the type library you import has name collisions with your program, you'll have to use the name spaces.

Smart Interface Pointers

In the 1956 science fiction classic "Invasion of the body-snatchers", aliens replace everybody in a small town with substitutes that are grown in giant green seed-pods. These replacement people look the same, talk the same, and act the same as the originals, but they are strangely different - they are loveless, emotionless automatons. The plot of this movie reminds me of smart pointers.

Smart pointers are helper classes that manage interfaces automatically. A smart pointer 'takes over' a COM interface, replacing its behavior with some subtle, but different actions. The main advantage of smart pointers is that they handle the COM creation, reference counting, and releasing automatically. With smart pointers, COM interfaces act a lot more like normal C++ pointers.

Before we get much farther, we should look at the downside of smart pointers. The most significant one is that they don't handle remote access. Smart pointers use CoCreateInstance, instead of the more powerful CoCreateInstanceEx method. There is no way to pass a smart pointer the COSERVERINFO structure, which contains the remote computer name. If your going to use a remote system, you'll have to specify the remote computer using DCOMCNFG.

The other common problem with smart pointers is related to program scope. The smart pointer constructor and destructor create and destroy the actual COM interface. This means you've got to be careful about the scope of the pointer.

Another issue is error handling. When a smart pointer gets a COM error, it throws an exception. This means you'll have to program with try/catch blocks. In terms of program overhead,

smart pointers are quite efficient. You shouldn't see a significant performance hit from using them.

Smart Pointer Classes

There are several classes of smart pointers in Visual C++. The first group of classes come from ATL, and are called CComPtr<> and CComQIPtr<>. These classes don't offer big advantages over standard COM interfaces.

If you are using the #import directive, you have access to a more powerful type of smart pointer. These pointers are based on _com_ptr_t. Much of the power of these objects is in their constructor. When you create a _com_ptr_t with either new or a declaration, it can actually connect to a COM object. This means the constructor is calling CoCreateInstance and managing the resultant interface.

Here's a typical use of a _com_ptr_t object. This smart pointer is bound to a specific interface - IBasicTypes. The type IBasicTypesPtr is a smart pointer. It is automatically declared by the #import directive. This is done with a macro _COM_SMARTPTR_TYPEDEF, which creates a typedef with the interface name + "Ptr".

The definition translates into something similar to the following typedef.

```
typedef _com_ptr_t<__uuidof(IBasicTypes) > IBasic-
  TypesPtr;
```

Surprise, _com_ptr_t is a template! The __uuidof() macro retrieves the GUID of the IBasicTypes interface.

Here's how we use the class:

```
IBasicTypesPtr pI( _T("BasicTypes.BasicTypes.1") );

long l1=1;
long l2=0;
pI->LongTest( l1, &l2 );
```

You'll immediately notice the missing steps: there is no CoCreateInstance() and no Release() called, and no HRESULT returned. Actually, all the usual things are going on, but they are hidden in the smart pointer class. First, the **CLSIDFromString()** is called to translate the CLSID (or ProgID) into a GUID. If the CLSID is valid, the smart pointer calls CoCreateInstance and obtains an interface pointer. The returned interface is saved internally and used whenever it is required.

Watch Out for Destructors

Smart pointers are wonderful things, but they also present some problems. You've got to be careful about the scope of smart pointers. When you declare them as we did above, smart pointers are created on the stack. This means that a pointer's lifetime will be only within the braces {} in which it was declared. This might be inside a function, or inside an **if** block.

Remember that the smart pointer destructor will be called when it goes out of scope. By default, the destructor of a smart pointer automatically calls Release(). This can cause several problems. When you destroy the last reference to the COM object, the COM server may shutdown. This means that the next COM call may have to restart the server - this can be quite slow for an out-of-process server.

Here's a piece of code that will cause problems:

```
void main()
{
  CoInitialize(0);
  IBeepPtr pBadPtr( _T("Beep.Beep.1") );
  pBadPtr->Beep();
  CoUninitialize();
  // crash on exit
}
```

The problem here is that CoUninitialize() is called before the destructor to the smart pointer. You'll get an un-handled exception from this code. The smart pointer calls Release() on its

destructor, but COM has already been shut down by CoUninitialize().

There is a relatively simple work-around. Declare the smart pointer inside a set of braces. This will ensure that the pointer is destroyed before CoUninitialize();

```
void main()
{
  CoInitialize(0);
  {
    IBeepPtr pBadPtr( _T("Beep.Beep.1") );
    pBadPtr->Beep();
  }
  CoUninitialize(); // no problem
}
```

This is lousy code for several other reasons. The main problem is that it has no error checking. If the "Beep.Beep.1" interface isn't registered, the declaration of the smart pointer will throw an exception. There is no try/catch block; it will fail with an uncaught exception. The next section describes how to catch errors thrown by a smart pointer.

Smart Pointer Error Handling

Many smart pointer operations don't return an HRESULT. Obviously, they need some sort of error checking. They get around this by throwing an exception whenever they get an error HRESULT. The _com_ptr_t class calls **_com_issue_error** whenever it encounters an error. **_com_issue_error** constructs a **_com_error** object with the HRESULT and throws it. Here's the code of the _com_ptr_t implementation of AddRef().

```
void AddRef()
{
  if (m_pInterface == NULL) {
    _com_issue_error(E_POINTER);
  }
  m_pInterface->AddRef();
}
```

AddRef needs a valid interface pointer. If the member interface pointer is NULL, it calls _com_issue_error with the HRESULT of E_POINTER. You can also see that the implementation of smart pointers isn't especially complicated.

To catch the **_com_error**, you need to include all smart pointer objects with a try-catch block.

```
try
{
  IBasicTypesPtr pI( _T("BadCLSID.BadCLSID.1") );

  PI->SomeMethod();
}
catch (_com_error e)
{
  // handle the error
}
```

All the code you write with smart pointers will need try-catch blocks.

The _com_error class provides a nice encapsulation of HRESULTs and common error handling functions. You can retrieve the raw HRESULT code by calling Error.

```
  catch( _com_error e )
  {
        cout << "HRESULT = " << e.Error() << endl;
       cout << "ErrorMessage() = " << e.ErrorMessage()
            << endl;
}
```

The ErrorMessage method takes the place of the FormatString API. The ErrorMessage method of _com_error handles the creation of the printable error string. It also automatically deletes the message buffer when it's done. FormatString is a very troublesome function. It has numerous complex arguments. The other problem with FormatString is that it allocates a string buffer that must be explicitly released with LocalFree().

How the IMPORT Directive Works

You've probably been wondering how the import directive accomplishes all the things it does. It includes MIDL definitions in the C++ program, creates smart pointers, and it gives us the useful _com_error type. The way all this is accomplished is quite ingenious.

The import directive creates two header files. These files are automatically created by retrieving information from the type library. The contents of these two files are included in the source as headers. Whenever the type library changes, the contents of these two headers is regenerated.

The first type of file is a typelib header, or TLH. It includes the following sections:

- The COMDEF.H header
- MIDL structure, enum, coclass, and interface typedefs.
- Smart pointer definitions for interfaces.
- Interface wrapper declarations.
- Interface raw declarations.
- An #include of the TLI file.

The other generated file is a typelib implementation file, and has the extension TLI. TLI files contain the implementation of smart pointer wrapper methods.

Raw and Wrapper Methods

When you call a method on a smart pointer, you're not directly calling a COM method. The smart pointer has a wrapper method for each of the interfaces methods. When you call the wrapper method, you're calling a local non-COM method of the smart pointer class. The wrapper method will call the raw COM method directly, and check the HRESULT returned.

Here's the definition of an actual wrapper class method. The wrapper class is IBasicTypes. This code comes from a TLI file.

```
inline HRESULT IBasicTypes::LongTest ( long l,
  long * pl )
{
```

```
    HRESULT _hr = raw_LongTest(l, pl);
    if (FAILED(_hr)) _com_issue_errorex(_hr,
        this, __uuidof(this));
    return _hr;
}
```

As you can see, the wrapper class just calls the raw COM interface. In this example, raw_LongTest() is an actual COM method. The preceding "raw" was automatically appended to the method by the compiler when it created the smart pointer. The raw method will return a normal HRESULT code. If the HRESULT is an error, a _com_error object is created, and thrown as an exception. If you debug into a COM method of a client using the #import directive, you'll see a very similar piece of code.

Summary

We've examined some template basics, and looked at how ATL implements COM. Of course the purpose of ATL is to hide all this implementation. Unfortunately, when you start debugging, you'll quickly find yourself trying to understand the ATL code.

The final section of this chapter examined the native compiler directive #import. Import uses the type library to generate two header files that include extensive definitions. One of the most useful parts of the import directive is the use of smart pointers. Using smart pointers, we can simplify much of our client application.

S I X T E E N

Other Topics

COM is full of concepts and techniques that aren't normally seen by programmers. This section attempts to deal with several of these. Most of these items are unrelated, but may be useful when you are working with COM applications.

Errors

We've already briefly discussed HRESULTS. Strangely enough, HRESULTS aren't handles, and they aren't results. An HRESULT is the 32-bit status code returned by almost all COM functions.

Normally in C and C++, we write functions to return values. The atoi() function is typical; it returns an integer from a string.

```
int x = atoi( "100" );
```

As C++ programmers, we're in the habit of returning meaningful values as function results. COM needs to do things a little differently. A COM method always should return an HRESULT. Here's how we would write the COM method for a hypothetical interface called ITest:

```
int x;
HRESULT hr = ITest->AtoI(  &x,  "100" );
```

In COM we can't guarantee that the method call will succeed. Returning an HRESULT allows the client to receive out-of-band information. Typically, a client might receive notification that it has lost communication with the server. If this were the case, the integer result returned by ITest->AtoI() would be meaningless.

The two most common HRESULTS are S_OK and E_FAILED. S_OK is defined as the number zero. When you test an HRESULT, you should use the predefined macros SUCCEEDED() and FAILED(). This is necessary because there are numerous success codes besides S_OK. Following is the standard method of checking COM errors.

```
HRESULT hr;
hr = CoCreateInstance(,,,…);
if (SUCCEEDED(hr))
{
  … // continue processing
}
```

The HRESULT is segmented into several bit fields, each of which defines part of the status.

```
 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+---+-+-+-----------------------+-------------------------------+
|Sev|C|R|     Facility          |               Code            |
+---+-+-+-----------------------+-------------------------------+
```

The meaning of these bits are as follows

Bits	Description
0-15	Information Code. Describes the specific error.
16-27	Facility Code. The subsystem that created the error.
28	Customer code flag. (not commonly used)
29	Reserved bit.
30-31	Severity Code.

Table 16.1 HRESULT bit fields

Information Code

This part of the error status defines the specific message. The code can be extracted with the macro HRESULT_CODE(), which applies a bit-mask to the HRESULT, returning only the code field.

Facility Code

Windows divides its error messages into groups, or facilities. The facility is the subsystem that created the error code. There are a number of standard facilities defined for windows. Each has a #define to identify it:

Facility	Description	Value
FACILITY_WIN32	General Windows error codes. Generally these were returned by the Windows API.	7
FACILITY_RPC	Codes returned by the RPC services. These generally indicate a communications problem.	1
FACILITY_DISPATCH	Errors generated by Idispatch interfaces.	2
FACILITY_STORAGE	Errors from structured storage. Generally the IStorage and IStream intefaces.	3
FACILITY_ITF	Interface dependent error codes. Each interface may define it's own codes.	4
FACILITY_SSPI	Security Support Provider Interface (SSPI). Generally related to authentication and security.	9
FACILITY_WINDOWS	Error codes from Microsoft defined interfaces.	8
FACILITY_NULL	General codes, such as S_OK.	0

Table 16.2 HRESULT Facility Codes

There are several other less common facilities defined in WINERROR.H.

Customer Code Flag and Reserved bits

You probably won't see much of either of these. The customer code is designed to allow interfaces to use their own specific set of errors. The reserved flag is just that, reserved for use by Microsoft.

Severity Code

The most significant two bits of the HRESULT represent the severity of the message. The severity code can be extracted with the HRESULT_SEVERITY() macro. In general, whenever you see an HRESULT with a negative decimal value, it is an error.

Severity	Description
0	Success.
1	Information. Just an informational message,
2	Warning. An error that requires attention.
3	Error. An error occurred.

Table 16.3 HRESULT Severity Codes

Looking Up HRESULTS

Much of the information about HRESULTs can be found in the system header file WINERROR.H. It's worth your time to open and browse this header, it is often a good source of information on error codes. Most of the errors in this file are not COM errors.

In general, HRESULTS are best viewed in hexadecimal. Many of the common error codes have the severity bit set, so they appear as large negative numbers in decimal. For example, the decimal number -2147221164 is much more readable as 0x80040154.

Because the HRESULT is a combination of several fields, you won't always be able to find your specific error code in WINERROR.H. One of the more common errors, RPC_S_SERVER_UNAVAILABLE, isn't in WINERROR.H. If you look it up, you'll find it mapped to the decimal number 1722.

This number is only the information code. The code returned by CoCreateInstance is 0x800706ba. This number is composed of several bit fields, it breaks down into the following:

```
0x10000000 + SEVERITY_WARNING
0x00070000 + FACILITY_RPC
0x000006ba + SERVER_UNAVAILABLE (1722L)
-------------------------------------
0x800706ba = RPC_S_SERVER_UNAVAILABLE
```

SCODES

The SCODE is a holdover from 16-bit windows. On Win32, the SCODE is defined as a 32-bit DWORD value. They are the progenitor of the HRESULT, so there are many similarities. Although interchangeable on Win32, you should use HRESULTS. If you see the term SCODE, you're probably working with code that was ported from Windows 3.1.

Displaying Error Messages

We've shown how to interpret and find error codes using the <WINERROR.h> header. Obviously, there are easier ways to get this information. Perhaps the most accessible method is to run "Error Lookup" application included in the Developers Studio. (Under the TOOLS menu.) This is OK for debugging, but you can also generate the text of the error messages interactively.

Using the _com_error class is by far the easiest way to display error messages. You can construct a _com_error object with your HRESULT and call the ErrorMessage message to get a string.

```
#include <comdef.h>

HRESULT hr = S_OK;
_com_error e(hr);

cout << e.ErrorMessage() << endl;
```

You need to include the file <comdef.h> to get the _com_error definition.

Using FormatMessage

The FormatMessage function can be used to look up the text of the message. Specifying FORMAT_MESSAGE_FROM_SYSTEM tells the function to look up the HRESULT in the system message tables.

```
char *pMsgBuf = NULL;

// build message string
::FormatMessage(
FORMAT_MESSAGE_ALLOCATE_BUFFER |
  FORMAT_MESSAGE_FROM_SYSTEM |
  FORMAT_MESSAGE_IGNORE_INSERTS,
      NULL,
      hr,
      MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
      (LPTSTR) &pMsgBuf,
      0,NULL);

CString MyCopy = pMsgBuf;

// Free the buffer.
LocalFree(pMsgBuf) ;
```

In this example, we are passing in an HRESULT code (hr) as the third argument. The error string will be written to a buffer pointed to by pMsgBuf. Note that this buffer is allocated by FormatMessage. This happened because we passed in the FORMAT_MESSAGE_ALLOCATE_BUFFER flag. FormatMessage will allocate the buffer, fill it with the message text, and return the pointer. This buffer needs to be de-allocated using LocalFree. LocalFree is considered to be obsolete, but I use it because the documentation for FormatMessage says it's required. You should also note that we're making a copy of the string before calling LocalFree.

Aggregation and Containment

COM offers two alternatives for the re-use of components. Containment means that an interface 'contains' another interface, and uses it to accomplish its goals. Aggregation is the act of combining COM objects: one COM object directly exposes another COM object without the client knowing it is dealing with two components.

Containment in COM is very straightforward. The outer, or 'containing' object, creates an instance of the inner object. It creates the 2nd object, and passes calls along to that object.

```
HRESULT ObjectA::BeepMe( long lDuration )
{
  IBeepObj *pInner;

  HRESULT hr = CoCreateInstance( CLSID_ObjectB,
      NULL,
      CLSCTX_INPROC_SERVER,
      IID_IBeepObj,
      (void**)&pInner );
  if (SUCCEEDED(hr))
{
    hr = pInner->BeepMe(lDuration);
    pInner->Release();
}
return hr;
}
```

This is a rather simple example. Normally, we would expect the containing object to create the contained object and keep it around for later use. You have a lot of flexibility in how you implement containment.

As you can see, the calling client will have no idea it is dealing with a second component. The first object completely manages the lifetime of its contained object. This technique is very simple and easily implemented. Aggregation is a special case of containment.

The problem with containment is that the outer object may have to implement every single method of its contained object. If there are a lot of methods, this can be aggravating. An aggregated object does not require these 'shell' methods. Aggregation actually exposes the complete inner object. The client will believe that it is actually dealing with a single component. Hiding the existence of the inner object introduces some very tricky programming into the implementation of IUnknown. Aggreagtable classes need a special version of QueryInterface, and special reference counting. The inner object needs to be specifically coded to handle aggregation.

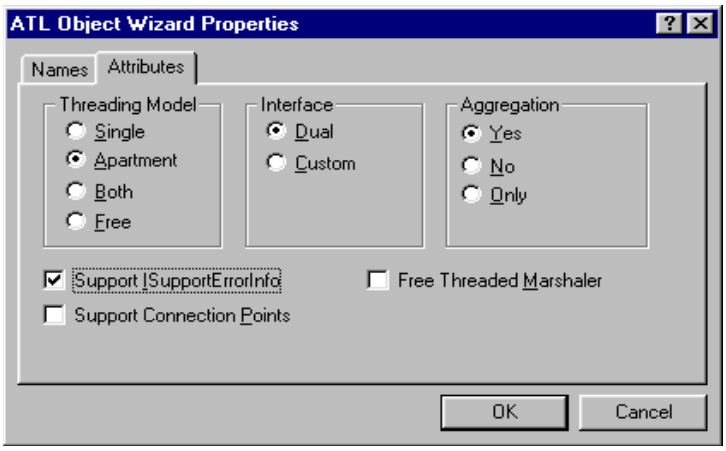

Figure 16–1 Configuring for aggregation

Once again, ATL takes care of much of this complexity. By default, all wizard-generated ATL classes are aggregatable. This is controlled by the radio button on the ATL Object Wizard Properties dialog. By allowing aggregation, you give other programmers the flexibility to aggregate your class.

The class factory of the ATL object has a base class of CComAggObject, which handles the special IUnknown. This comes from the ATL template macro DECLARE_AGGREGATABLE(). By

selecting NO to aggregation, you'll get the DECLARE_NOT_ AGGREGATABLE macro.

Building a COM Object with MFC

You can build perfectly good COM objects using MFC. MFC uses the base class CCmdTarget and a number of macros to implement COM. In this example we'll use MFC to create a COM ready class. This example comes from the MfcClient example program.

CCmdTarget is the base class for MFC message maps. CCmdTarget is fully capable of supporting OLE. This means it supports COM. What you we are doing in this section, therefore, is demonstrating how to use MFC in place of ATL. As discussed in Chapter 15, ATL is the preferred choice for server implementation. However, there are occasions where the use of MFC can have advantages. For example, if you have developed an MFC client and you want to embed a callback object inside of it so that the server can talk back to the client, then you suddenly have a need to implement COM with MFC. This example presented here is based on the Callback client from the chapter on Callbacks. MFC supports COM at a lower level, and we're going to have to write some of the basic COM plumbing to get this example to work.

We will assume you have already defined the COM interface using MIDL. We will be implementing a COM interface named ICallBack that was created in the Callback chapter. In order to get the MIDL-generated definitions, we'll include two files from the server. Note that the path to these include files is relative. You will need to ensure that they point to the correct files where you created the server project.

```
#include "../CallbackServer/CallbackServer _i.c"
#include "../CallbackServer /CallbackServer.h"
```

The CCmdTarget class has several interesting features. This class has a Windows message loop. It also implements the standard methods of IUnknown. As you recall, these are QueryInter-

face(), AddRef(), and Release(). Simply by inheriting from CCmdTarget, we get a fully functional COM object.

We're also going to use an obscure feature of C++ language - nested classes. If you're unfamiliar with nested classes, they are straightforward. A nested class is a class declared within the scope of another class. Whenever you reference a nested class, you need to fully qualify the name with both classes.

You're going to see a lot of macros in the following code. MFC/OLE uses macros quite extensively. This makes sense, because the fundamentals of COM are very standardized. These macros, however, can make MFC-style COM difficult to follow.

We will create our class object as follows:

```
class CMyCallback : public CCmdTarget
{
public:
  CMyCallback(){};
  DECLARE_INTERFACE_MAP()
public:
  BEGIN_INTERFACE_PART( MyEvents, ICallBack )
  STDMETHOD(Awake)(long lDuration);
  END_INTERFACE_PART(MyEvents)
};
```

The BEGIN_INTERFACE_PART and END_INTERFACE_PART create a nested class. This nested class appends an X to the string "MyEvents", and declares a member variable named m_xMyEvents of the same type. The macro expands out into code that is similar to this:

```
// code generated by BEGIN_INTERFACE_PART macros
class XMyEvents: public ICallBack
{
  public:
    ULONG AddRef();
    ULONG Release();
    HRESULT QueryInterface(REFIID iid, LPVOID*
  ppvObj);
    HRESULT Awake(long lDuration);
} m_xMyEvents;
```

```
friend class XMyEvents;
```

The class XMyEvents will have to implement the three methods of IUnknown. Because it is a custom interface, it also has a custom method Awake(). This is the only application-specific method exposed by this COM object. In order to reference AddRef in the nested class, you would use m_xMyClass.AddRef(). If you didn't want to use the MFC macros, you could just type in the required definitions.

Adding Code for the Nested Classes

We will put the implementation of this class in the source file MfcClient.cpp. The standard AppWizard application code is already created, so we're going to add a nested class defined in the BEGIN_INTERFACE_PART macro. This is a fairly basic class, other than the nested definition. Here is the top portion of this file.

```
BEGIN_INTERFACE_MAP( CMyCallback, CCmdTarget )
  INTERFACE_PART(CMyCallback, IID_ICallBack, MyEvents)
END_INTERFACE_MAP()
```

The macros above are implementing an MFC interface map. This code ties in with the DECLARE_INTERFACE_MAP macro in the header. An interface map is a MFC/OLE concept. It is quite similar to a standard MFC message map, which reads and processes Windows messages. The interface map takes care of a lot of the plumbing of a COM object in MFC. This includes reference counting with AddRef() and Release(), as well as handling COM aggregation. This is all functionality that is handled automatically by ATL. Since we aren't using ATL in this client, we have to implement things in the MFC/OLE way.

Let's continue with the class implementation. Here's the standardized part of the CPP implementation.

```
// Standard COM interfaces -- implemented
// in nested class XClientSink
STDMETHODIMP_(ULONG) CMyCallback::XMyEvents::AddRef()
{
```

```
  METHOD_PROLOGUE_EX(CMyCallback, MyEvents)
  return (ULONG)pThis->ExternalAddRef();
}
STDMETHODIMP_(ULONG) CMyCallback::XMyEvents::Release()
{
  METHOD_PROLOGUE_EX(CMyCallback, MyEvents)
  return (ULONG)pThis->ExternalRelease();
}
STDMETHODIMP CMyCallback::XMyEvents::QueryInterface(
  REFIID iid, LPVOID far* ppvObj)
{
  METHOD_PROLOGUE_EX(CMyCallback, MyEvents)
  return (HRESULT)pThis->ExternalQueryInterface(
&iid, ppvObj);
}
```

One unfamiliar part of this code may be the use of the METHOD_PROLOGUE_EX macro. This macro automatically gets the this pointer from the outside class. This is possible because the nested class and its outside class are declared as 'friends'. The METHOD_PROLOGUE_EX macro creates an outside pointer named pThis. (The outside class is CMfcClient).

We use the pThis pointer to delegate the AddRef(), Release(), and QueryInterface functions to the outside CMfcClient class, which knows how to handle them. These functions are implemented in the base class CCmdTarget.

All of what has preceded this point is boilerplate COM code. The only 'custom' aspect of this code is the names of the classes. Finally, we're going to add our one-and-only custom method.

```
// Pop up a message box to announce callback
STDMETHODIMP CMyCallback::XMyEvents::Awake(long lVal)
{
    CString msg;
    msg.Format( "Message %d Received\n", lVal);
    AfxMessageBox( msg );
    return S_OK;
}
```

This code is self-explanatory. It displays a message box.

Accessing the Nested Class

The syntax of accessing the nested class is somewhat unusual. We'll assume you are accessing an instance of the CCmdTarget class called pMyClass. In this example, we'll extract an ICallBack object from the class.

```
ICallBack *pC;
  hr = pMyClass->m_xMyEvents.QueryInterface(
      IID_ICallBack, (void**)&pC);
  if (SUCCEEDED(hr))
  {
    pC->Awake( 1 );
    pC->Release();
  }
```

We have presented this example to show that there are other ways besides ATL to implement COM. Working with MFC COM is a whole different experience from using ATL. There are numerous books on OLE and MFC that cover this area in minute detail - now that you understand the fundamentals of COM, these books become much easier to comprehend!

A P P E N D I X

COM Error Handling

Much of the frustration of using COM arises when things don't work. You create a COM server and its client, run the client but the server never activates. Solving this type of problem can be a time-consuming, maddening activity. This appendix is dedicated to the discovery and elimination of bugs that prevent a COM client from finding and starting a COM server, as well as other bugs that gum up the works between a client and a server.

One of the hardest parts about working with COM is dealing with errors. The debugging cycle with COM programs is more complex then with standard C++ programs. There are a number of reasons for this.

First, much COM error checking is done at run-time. Using the wizards, it's relatively easy to build a client and server application. Everything looks fine, until you run it. A COM program is really a complex combination of different programs, the registry, and operating system components. Any of these parts can go wrong, and they do.

Secondly, when you get COM error messages, they often aren't very specific about the problem. Perhaps the worst example of this is the RPC_S_SERVER_UNAVAILABLE error that you commonly get when working across networks. Even at their best, HRESULT's offer pretty meager information about the prob-

lem. The context of an error can be extremely important in interpreting its cause.

Also, a huge part of the COM system is hidden from the programmer. The COM subsystem is responsible for server location and activation. As an application developer, we hope and pray that all the elements of this connection are correct.

Security just makes matters more difficult. COM is designed to provide inter-process and inter-network connections. Unfortunately, these avenues of communication are highly subject to break-ins. This means that the security layers of the network and operating system are going to insist that access is legitimate. Another characteristic of security subsystems is that they don't give very informative error diagnostics. When a connection fails, the security subsystem probably won't tell you why - to do so would be a security breach.

Finally, we've got to admit that the COM environment is quirky and prone to bugs. Only a few people really understand this complex system enough to diagnose tough problems. When you're just starting with COM, you probably don't have access to these people.

Fortunately, COM is getting easier. Microsoft is exerting considerable effort in on making COM more usable. Each new Developer Studio product has better integration. Also the advent of tools like ATL have made a big difference. As COM grows in prevalence (if not always popularity), there's more information available.

Sources of Information

The primary source for information about COM is the MSDN library. There is quite a lot of COM related material on these CD's, but it is poorly organized and sometimes not clearly written. Nevertheless, the MSDN Libraries are a must-have for serious Visual C++ and COM developers. If you don't get the CD's, the online resources at Microsoft's web site are quite useful as a fallback.

There are several good COM FAQ's available on the Internet. Searching these archives can give help with commonly encountered problems. One example is the "FAQ: COM Security Frequently Asked Questions", Article ID: Q158508

There is also a DCOM list server resource, which archives COM-related mailing lists. If you subscribe to the mailing lists, you'll never have to worry about an empty inbox. The archive holds a huge volume of material about COM. You'll find posts by some of the experts in the COM world, including COM developers. You'll also find thousands of questions about topics that are irrelevant to your application.

Here are two of these resources:

```
http://discuss.microsoft.com/archives/atl.html
http://discuss.microsoft.com/archives/dcom.html.
```

Reading the messages on the list server will also give you some idea about the desperation of COM developers who are debugging problems. You should spend some time searching these archives before posting questions. Almost all the questions have been asked, and answered several times before.

Common Error Messages

In the following section we're going to look at some of the more common errors you will encounter when working with COM. I've encountered most of these, usually when working with remote servers.

These errors are arranged in alphabetical order.

Error	Decimal	Hex
CO_E_BAD_SERVER_NAME	-2147467244	80004014
CO_E_CANT_REMOTE	-2147467245	80004013
CO_E_NOTINITIALIZED	-2147221008	800401f0
CO_E_SERVER_EXEC_FAILURE	-2146959355	80080005
E_ACCESSDENIED	-2147024891	80070005
E_FAIL	-2147467259	80004005

Error	Decimal	Hex
E_NOINTERFACE	-2147467262	80004002
E_OUTOFMEMORY	-2147483646	80000002
E_POINTER	-2147483643	80000005
ERROR_INVALID_PARAMETER	-2147024809	80070057
ERROR_SUCCESS	0	0
REGDB_E_CLASSNOTREG	-2147221164	80040154
RPC_S_SERVER_UNAVAILABLE	-2147023174	800706ba

Table A.1 Typical COM errors

CO_E_BAD_SERVER_NAME • A Remote activation was necessary but the server name provided was invalid.

This is one of the few self-explanatory error messages. Note that this doesn't mean you entered the wrong server name. Unrecognized servers show up with the RPC_S_SERVER_UNAVAILABLE error.

- Check the server name for invalid characters.
- Check the parameters to the COSERVERINFO structure.

CO_E_CANT_REMOTE • A Remote activation was necessary but was not allowed.

This is an uncommon problem. You are trying to improperly start a server.

- Check the CLSCTX in CoCreateInstance. Be sure it matches the type of server.

CO_E_NOTINITIALIZED • CoInitialize has not been called.

This is an easy problem. You probably just forgot to call CoInitialize. It may also indicate that you have a threading problem. CoInitialize should be called on each thread.

- Call CoInitialize or CoInitializeEx before other COM calls.
- Be sure you call CoInitialize for each thread. COM interactions must be marshaled between threads.

CO_E_SERVER_EXEC_FAILURE • Server execution failed.
Often occurs when calling CoCreateInstanceEx.

- Set the "Remote Connect" flag to "Y" on your server. The registry key is HKEY_LOCAL_MACHINE\Software\Microsoft\Ole EnableRemoteConnect='Y'. You are required to reboot after changing this setting.

E_ACCESSDENIED • General access denied error.

This is an error from the security subsystem. The server system rejected a connection. Also known as “the error from hell” because it is often very difficult to resolve. Access Denied problems can be very difficult to diagnose. This error is most likely encountered when using DCOM for remote connections.

- The server must already be started on remote Windows 95/98 computers. You will sometimes get this instead of RPC_S_SERVER_UNAVAILABLE when trying to start a remote server located on a Windos 95/98 machine.
- Check COM security, file protection, and network access. Check launch permissions, etc using DCOMCNFG. There are many levels of security that can be incorrect. Be sure the server identity is not set as "Launching User". See Chapter 14.
- The server program may be registered, but the EXE maymissing from or inproperly located on a remote computer. Try re-installing and re-registering the server.
- Check File and Print Sharing for Microsoft Networks on Windows 95. If you have Novell installed, check NetWare File/Print Sharing.
- The server may not allow remote activation. If so, it cannot start. Change the server.
- Check the parameters of CoInitializeSecurity. See Chapter 14.
- See if the remote server is starting. The server may be starting, but may have a problem with call-level security.
- A server running as an NT service may be running under "SYSTEM" or some other account that does not have permission to access resources. This may mean the system

account is trying to access network resources, such as network disks. Launch the service with a different user name; use the Services applet in the Control Panel.

E_FAIL • Unspecified error.

This error doesn't tell you much. Often servers return this code when they have a general processing failure or exception. In our example code, we often return this code to indicate a domain-specific problem.

- A COM method failed on the server. Check the server implementation.

E_NOINTERFACE • No such interface supported.

You asked a server for an interface it doesn't support. This means your CLSID is probably ok, but the IID is not. This call is returned by QueryInterface (or through CoCreateInstance) when it doesn't recognize an interface. It may also be a Proxy/Stub problem.

- Check the IID or name of the interface you requested. Be sure you typed in the correct CLSID. Be sure the coclass supports the interface.
- The Proxy/Server DLL for the server is not properly registered. You may have forgotten to build and register the "ps.mk" file.
- The interface was not properly registered in the registry. Re-register your server application. Look for the interface with OLEVIEW.
- You forgot the COM_INTERFACE_ENTRY in your ATL server's header.

E_OUTOFMEMORY • Ran out of memory.

This message may be unrelated to the actual error. Uncommonly seen.

- DefaultAccessPermissions does not include the SID, or security identifier, for the SYSTEM account. Use DCOMC-

NFG or the OLE/COM Object Viewer to add "SYSTEM" to the security id's in the default access permissions.

E_POINTER • Invalid pointer.

This error indicates a general problem with pointers. You probably passed a NULL to a method that was expecting a valid pointer.

- You passed a null or invalid pointer in a method call.
- You passed in invalid (IUnknown*) or (void**). Did you forget an ampersand?
- Check the ref and unique attributes in the IDL code.

ERROR_INVALID_PARAMETER • The parameter is incorrect.

You have a problem in one of the parameters to your function call. This is commonly seen in functions such as CoCreateInstance, CoCreateInstanceEx, CoInitializeSecurity, etc.

- Check for missing ampersand (&) on pointers.
- Check for missing ampersand on references, such as REFCLSID parameter.
- Check all parameters carefully.

ERROR_SUCCESS • The operation completed successfully.

The same message as S_OK and NO_ERROR. This message is a wonderful oxymoron.

- You did everything right.

REGDB_E_CLASSNOTREG • Class not registered.

You'll get this error if you had problems registering the server. It may also indicate in incorrect CLSID was requested.

- You called CoCreateInstance or CoGetClassObject on a class that has no registered server. The CLSID was not recognized.
- Check the registry. See if the CLSID is registered. Look up the CLSID under the HKEY_CLASSES_ROOT\CLSID key.

- Try re-registering the server. Type "servername -regserver" at the DOS prompt.
- Check the GUID's.
- Use OLEVIEW to verify that the server is properly registered.

RPC_S_SERVER_UNAVAILABLE • RPC server is unavailable.

This problem is very common when working with remote servers. This is a generic remote connection error. RPC is the protocol used to implement DCOM. This can be a system set-up problem or a security problem. You're about to learn a lot about networking!

- Test the remote connection with PING.
- Test the remote connection with TRACERT
- You may have entered an invalid server name. This may also be a name resolution problem. If so, try using the TCP/IP address of the server instead of the name.
- The server computer may not be running, or it may be disconnected from the network. It may also be unreachable because of the network configuration.
- The RPCSS service may not be started on the remote Windows NT computer.
- Be sure DCOM is configured on the server. Run DCOMCNFG and check all the possibilities. Check EnableRemoteConnect and EnableDCOM.
- If you are connecting to a remote windows 95/98 system, the server must be running. DCOM will not automatically start a server remotely on Windows 95/98.
- Be sure the COM server is registered on the remote computer. You may get this instead of "Class Not Registered" when connecting to remote machines.
- Check the TCP/IP installation on both the client and server computer.
- See if you can run the client and server locally on both computers. It is a lot easier to debug COM problems on a local computer. If it doesn't work locally, it probably won't work over the network.

DCOM Errors

This section discusses some of the problems you will encounter when working across a network. I've tried to outline some of the approaches I've found useful in diagnosing and fixing problems.

My company deploys a product that uses DCOM to connect a GUI and server. We have installed it at several thousand sites. Our customers have a tremendous variety of networks and configurations, and we spend a lot of time debugging remote connections. The software itself if very stable, but network configurations are not. I have learned a lot about DCOM trying to debug difficult installations.

Unfortunately, when you use DCOM over a network, you're probably going to encounter a lot of problems. The more network configurations you work with, the more problems you'll have.

Debugging network issues falls somewhere between a science and voodoo. Having access to a competent network administrator is a blessing. In the real world however, network administrators are often not available, so the programmer has to resolve problems himself or herself.

Following is a series of steps that I use when working on DCOM issues.

Get It Working Locally

The first step in the debugging process is to get the client and server working locally. Install both components on the server machine, and keep at it until you can successfully communicate. If a component won't work locally, it won't work across a network. You probably developed and tested the application on a single computer, but be sure to test it on the server system also.

By getting the system to work locally, you've eliminated most of the common programming errors. There are still a few things, like security and remote activation, that you can only test across the net. Specify your local computer in the COSERVERINFO structure, which will exercise some of the network-related code.

Be Sure You Can Connect

Before you even try to install your program, debug the network configuration using whatever tools you have available. Start by checking the network neighborhood, and ensure that you can browse the remote computer. This is not always possible, and a failure to browse doesn't preclude DCOM working. In most cases, however, browsing is good starting place for checking connections. Check the connection in both directions.

Perhaps the most useful tool is PING. Ping sends a series of network packets to the server and waits for a response. Most installations support PING.

```
C:\>ping www.ustreas.gov

Pinging www.treas.gov [207.25.144.19] with 32 bytes of data:

Reply from 207.25.144.19: bytes=32 time=209ms TTL=247
Reply from 207.25.144.19: bytes=32 time=779ms TTL=247
Request timed out.
Reply from 207.25.144.19: bytes=32 time=852ms TTL=247

Ping statistics for 207.25.144.19:
    Packets: Sent = 4, Received = 3, Lost = 1 (25% loss),
Approximate round trip times in milli-seconds:
    Minimum = 209ms, Maximum =  852ms, Average =  460ms
```

Figure A–1 The ping command

PING does a number of interesting things for you. First, it resolves the name of the remote computer. If you're using TCP/IP, the name of the remote computer will be turned into a TCP/IP address. In the above example, PING converts the name "Raoul" into the TCP/IP address [169.254.91.12].

You should also try PING from both directions. If you are using callbacks or connection points, you must have COM working in both directions. Callbacks and connection points can be very difficult to debug.

Try Using a TCP/IP Address

Name resolution can be a vexing problem in remote connections. Most people want to work with names like "\\RAOUL" and "\\SERVER", rather then TCP/IP addresses. The process of turning that readable name into a network address is called "Name Resolution", and it can be very complicated on some system configurations. A common work-around is to refer to the server by its TCP/IP address. This will eliminate many name resolution problems - which are outside the scope of this discussion. You can easily put a TCP/IP address into the COSERVERINFO structure, instead of a standard computer name.

Use TRACERT

You can also glean interesting information from the TRACERT utility. If you have any weird network configurations, they may show up here. Here is typical output:

```
c:\>tracert www.ustreas.gov

Tracing route to www.treas.gov [207.25.144.19]
over a maximum of 30 hops:

  1   181 ms   180 ms   169 ms  ct1.intercenter.net [207.211.129.2]
  2   188 ms   188 ms   170 ms  ts-gw1.intercenter.net [207.211.129.1]
  3   176 ms   187 ms   190 ms  ilan-gw1.intercenter.net [207.211.128.1]
  4   547 ms   505 ms   756 ms  core01.rtr.INTERPATH.NET [199.72.1.101]
  5   516 ms   323 ms   338 ms  tysons-h2-0.rtr.INTERPATH.NET [199.72.250.26]
  6   184 ms   708 ms   216 ms  mae-east2.ANS.NET [192.41.177.141]
  7   576 ms   981 ms   423 ms  h12-1.t60-8.Reston.t3.ANS.NET [140.223.61.25]
  8   419 ms   804 ms   570 ms  f5-0.c60-14.Reston.t3.ANS.NET [140.223.60.210]
  9   314 ms   641 ms   621 ms  www.treas.gov [207.25.144.19]
Trace complete.
```

Figure A–2 The tracert command

As you can see, the route your DCOM packets are taking to their destination may be surprising! Beware of gateways, routers, proxies, and firewalls - they can and will block your connection.

Windows 95/98 Systems Will Not Launch Servers

Hopefully this will change in the future. If your server is on Windows 95/98, you must manually start it before connecting from a remote computer. There is actually a very good security reason for this limitation. Because authentication on Windows 95/98 is so limited, there is no way to ensure that unauthorized users don't launch your server.

Windows NT systems have no such limitation. NT is fully capable of validating remote users and launching servers safely. Unfortunately, it is also capable of rejecting legitimate users because of set-up problems.

See Chapter 14 for details.

Security is Tough

Assuming you've got the physical network connections working, you're going to have to get through several layers of security. This is especially an issue on Windows NT, which has an extremely rich and complicated security layer.

A discussion of network security is well beyond the scope of this book. We can, however, point out a few useful tools. See also chapter 14 for a detailed discussion.

DCOMCNFG is your first line of defense when working with COM security. DCOMCNFG allows easy access to most security settings.

If you look at the "Common Error Messages" section above, you'll see that many of the error messages are related to security. This is not accidental. One of the tenets of good security is to deny outsiders any information about your security set-up. This makes error messages especially unhelpful. When the security sub-system detects an error, it won't give you a useful error message. By telling you what you did wrong, it is also giving you information about its configuration - which is a security no-no.

If you're working with NT, it logs some security messages in the event viewer. Be sure to check this information if you're getting un-helpful security messages.

Using the OLE/COM Object Viewer

This utility is also known as OLEVIEW. This utility is a useful tool when diagnosing registration issues. This tool was originally developed for viewing OLE interfaces, but it works for all COM interfaces. This tool is essentially a view of the registry and type libraries. The information seen in OLEVIEW all originates in the registry. OLEVIEW does more than just view registry keys; it also runs servers and interrogates type libraries for information.

Under newer versions of the Developer Studio, OLEVIEW shows up under the TOOLS|OLE/COM Object Viewer menu. Note that there are a number of different versions of OLEVIEW in circulation, and you'll get different results from each of them. The older versions show much more limited information.

When you start the viewer, you'll be presented with a number of folders. COM classes and interfaces may show up under several of these folders. We're going to use OLEVIEW to find our IDLTestServer server.

Select the "Object Classes" folder. Inside that folder select the "Automation Objects" folder and expand it. Search down for the "BasicTypes Class". This is the class we created in the chapter on MIDL. If you haven't built or installed the example programs, this class will not exist. If this is the case, just pick some other class for viewing.

When you double click on the "BasicTypes Class" object, several very interesting things happen.

The left hand column will show the interfaces supported by the class. In this case, we'll see our custom IBasicTypes class, as well as a number of standard COM interfaces that are implemented through ATL.

The right hand column displays detailed information about the server and its interfaces. You can make changes to several aspects of the server here. For example, you can designate that the server runs on a remote machine by making changes in the "Activation" tab.

One of the most fascinating aspects of OLEVIEW is that it actually activates and connects to the server, if possible. When you examine the running processes while using OLEVIEW, you'll

actually see the highlighted server is running. Of course, if there's a problem with the server, you'll get an error message.

This means that you can use OLEVIEW to actually debug your COM classes. If you can expand the server with OLEVIEW, the registration was successful.

There are several types of information visible through OLEVIEW. We've already seen where it lists automation classes. If you want to see all the interfaces registered on your system, open the "Interfaces" folder. This folder lists all the interfaces, custom and dispatch that are registered. There are a lot of them.

You can also open and view type libraries. Look under the "Type Libraries" tab. Expanding the type libraries under this folder shows you the stored IDL information in the library.

You should spend some time exploring this tool. It can be very useful finding and fixing registration problems. It is also a useful way to change security settings.

INDEX

#IMPORT 253
$(OutDir) 103
$(TargetPath) 103
__uuidof 256
__variant_t 135
_com_error 258
_com_issue_error 258
_COM_SMARTPTR_TYPEDEF 256
_ICpTestEvents 215

A

access permission 235
AccessPermission 235
Activation 71
Active Template Library 28
ActiveX 27, 125
adding properties 142
AddRef 82
Advise 184, 225
AfxBeginThread 205, 208
AfxGetApp 200
aggregation 23, 269
AllocSysString 118
angle brackets 253
apartment threads 153, 155, 158
    using 163
API 178
AppID 170, 172
    registration 174
application identifier 174
argument
    named 130
array
    conformant 119
    fixed 120
    fixed length 115
    multi-dimensional 120
    open 120
    varying 119, 121
asynchronous event 181
ATL 28, 246
    fundamental classes 252
    generated code 55
    server self registration 174
    threading models 156
ATL wizard 30, 156
attribute 97, 109, 113
authentication 237
automation
    OLE 125

B

base type 108
bi-directional 183, 213
binding 127
    early 136
boolean 108, 113
Both threads 156
both threads 161
browsing 136
BSTR 113
byte 108

C

callback 183
    chronology of events 201
    connection points 213
    custom interface 183
    example 185
callback interfaces 181
calling methods 24
CBeepObj 60

CCallBack 215
CComAggObject 252
CComCoClass 145, 148, 252
CComDynamicUnkArray 227
CComModule 177, 193
CComMultiThreadModel 160, 252
CComObject 198
CComObjectRoot 194, 252
CComObjectRootEx 158, 227, 252
CComPtr 252, 256
CComQIPtr 252, 256
CComSingleThreadModel 158, 252
CComVariant 135
CCpTest 215
char 108
class
    definition 62
    factory 77
    store 168
class declaration 2
class wizard
    adding properties 142
client 45
    connectivity 20
    running with server 40
    simplest 19
CLSCTX 23, 88
CLSCTX_INPROC_SERVER 88
CLSCTX_LOCAL_SERVER 88
CLSCTX_REMOTE_SERVER 88
CLSID 22, 87, 170, 171
    registration 171
CLSIDFromString 257
CO_E_BAD_SERVER_NAME 280
CO_E_CANT_REMOTE 280
CO_E_NOTINITIALIZED 280
CO_E_SERVER_EXEC_FAILURE 281
coclass 6, 100
CoComObject 79
CoCreateInstance 22
CoCreateInstanceEx 231, 233
CoGetClassObject 79
CoGetInterfaceAndReleaseStream 207
COINIT 157
COINIT_APARTMENTTHREADED 157
COINIT_MULTITHREADED 157
CoInitialize 21
CoInitializeEx 21
CoInitializeSecurity 239
COM
    array attributes 120
    class context 23
    client 19
    creating clients 43
    creating servers 43
    directional attributes 109
    distributed 14, 229
    error handling 277
    identifiers 87
    interfaces 68
    language independent 68
    map 139, 194
    MFC 246
    network 4
    object viewer 289
    pointer values 110
    principles 67
    process 3
    registry structure 168
    server threading models 153
    string attributes 114
    subsystem initializing 21
    threading model 151
    transparency 69
    typical errors 280
    vocabulary 5
COM interface
    pure virtual 74
COM object
    adding 33
    interface 8
    typical 11
    unique 5
COM server 4, 14
    DLL based 29
    simple 27
CoMarshalInterThreadInterface-
  InStream 206

communication 6
compiler support 245
component class 100
components 10
conformant 119
connection point 213
    classes 215
    container 224
    interfaces 215
containment 269
contract 69
cookie 187, 188
COSERVERINFO 231
CoTaskmemAlloc 117
CoTaskMemFree 117
CoUninitialize 22
coupling 89
CreateInstance 78, 79, 198
CreateObject 128
CString.AllocSysString 231
custom build 97, 103
custom callback 182
custom marshaling 95
CWinApp 195

D

data transfer 107
DCOM 14, 229
    errors 285
DCOMCNFG 288
default pointer 99
DEFAULT_CLASSFACTORY 79
delta pinging 243
design of COM 3
destructors 257
disconnection 242
dispatch interface 140
DISPID 129
DISPPARAMS 129
distributed COM 14, 229
DLL 14, 29
DllRegisterServer 178
DllUnregisterServer 178
domain name 12
double 108, 112
dual interface 98, 99, 125, 137
    VTABLE 138
Dynamic Link Library 14, 29

E

E_ACCESSDENIED 281
E_FAIL 282
E_NOINTERFACE 282
E_OUTOFMEMORY 282
E_POINTER 111, 283
early binding 127, 136
enumerations 121
err 144
err object 134
Error
    try/catch 259
error
    properties 135
    run-time 134
error handling 277
    DCOM 285
error messages 267
    common 279
ERROR_INVALID_PARAMETER 283
ERROR_SUCCESS 283
ErrorInfo 145
ErrorMessage 259
exception 258
EXE 14
EXECPINFO 134
ExitInstance 199, 225
export file 58

F

facility 265
FAILED 264
fiber 152
FindConnectionPoint 224
first_is 120
fixed 119
float 108
FormatMessage 268
Free 156

free threads 153, 155, 160, 164
FreeLibrary 29

G

garbage collection 242
GetIDsOfNames 128
GetMessage 154
GetTypeInfo 128
GetTypeInfoCount 128
Globally Unique IDentifier 12
GUID 5, 12
GUIDGEN 13

H

hard typing 136
heap 2
heartbeat 242
helpcontext 97
helpstring 97
hives 168
HKEY 168
HKEY_CLASSES_ROOT 168
HKEY_LOCAL_MACHINE 168
HRESULT 22
    bit fields 264
    facility codes 265
    severity codes 266
HRESULT_CODE 265
HRESULT_SEVERITY 266
HRESULTS 263
hyper 108

I

i.c file 94, 194
ICallBack 215
IClassFactory 79
IConnectionPointContainer 224
IConnectionPointContainerImpl 214, 216
IConnectionPointImpl 214, 225
ICpTest 215
id binding 127
identity 238
IDispatch 98, 127
IDispatchImpl 129, 139, 252
IDL boolean 113
IDL definitions 126
IDL language 91, 95
IDLdefinitions 126
IDLTESTLib 102
IID 22, 87
impersonation 237
    levels 241
import directive 260
importlib 100, 103, 107
inheritance 88
    object 61
Init 197
InitInstance 196
in-process server 14, 29, 178
InprocServer32 172
int 108
interactions
    between client and server 16
interface 7, 68, 73
    as contract 69
    attributes 99
    callback 181
    defining and using 107
    defining with IDL language 92
    dispatch 140
    dual 98, 125, 137
    execute a method 24
    IDispatch 127
    inheritance 89
    isolate 8
    key 170
    map 194
    polling 183
    release 24
InterfaceSupportsErrorInfo 146
InterlockedDecrement 84
InterlockedIncrement 84
inter-process 6
inter-thread marshaling 204, 207
invasion of the body-snatchers 255
Invoke 128, 133
IPersistFile 98

IRegister 175
isolate 8
    implementation 10
IStream 207
ISupportErrorInfo 144, 145
IUnknown 10, 74

K

keys 170

L

language indpendent 5
language-independent 68
last_is 120
late binding 127
launch permissions 236
launching servers 288
LaunchPermission 236
length_is 120
library
    statement 102
    type 102, 126
lifetime 83
LoadLibrary 29
Local Procedure Calls 229
local/remote transparency 69, 234
LocalServer32 171, 172
LocalService 174
LocalSystem 239
lock 227
long 108
LPC 229
lstrlen 113

M

marshaling 85, 86, 151
    between threads 162
    custom 95
    inter-thread 204, 207
    standard 94
max_is 120
message loop 153
method 144
    adding 36
method call 24, 85
MFC 27, 245, 246
MFC dialog 191
Microsoft Interface Definition Language 91
MIDL 70, 91
    base types 108
    compiler 91
    post-processing 103
    special tab 96
    structures 121
MKTYPLIB 95
model 70, 152
MTA 155
MULTI_QI 232
multi-dimensional 120
multiple inheritance 98
multi-threaded programming 204

N

name 10
name resolution 287
named arguments 130
namespace 254
native compiler directives 253
network 4
    traffic 182
new 2
NMAKE 103
no_namespace 254
NoRemove 176
NT LAN Manager Security Support Provider 237
NTLMSSP 237
NULL 109

O

OAIDL.IDL 129, 133
object
    browsing 136
    inheritance 61
    maps 58
    stateless 164
    viewer 289

objected oriented model 2
OBJIDL.IDL 99
OLE 27
    automation 125
OLE/COM object viewer 289
OLE32.DLL 72
oleautomation 99
OLECTL.H 129
OLEVIEW 289
on _com_ptr_t 256
open 120
Open Software Foundation 12
OSF 12, 107
out 109
out-of-band 264
out-of-process server 14

P

parameter
    boolean 113
    double 112
permissions
    access 235
    launch 236
ping 286
PLG file 105
pointer 110
pointer_default() 99
polling 183
post build step 104
PostThreadMessage 158
POTS 152
pre-processor 97, 247
principles of COM 67
process 3, 152
ProgID 170, 172
    registration 172
programmatic identifier 172
progress 182
property 140
    attributes 141
    standard 129
propget 141
propput 141
propputref 141
protocol 230
proxy 85
published 70
pure virtual 74

Q

QueryInterface 76, 81

R

raw method 260
ref 110
reference counting 82
REG scripts 175
RegCreateKey 178
REGDB_E_CLASSNOTREG 283
RegDeleteValue 178
REGEDIT 175
registration
    server 64
registry 73, 167, 178
    editor 167
    resources 175
    scripts 65
RegServer 177
REGSVR32 103, 178
Release 82
Remote Procedure Calls 229
RemoteServerName 174, 134
retval 110, 141
RGS 175
RPC 229
RPC_C_AUTHN_NONE 241
RPC_S_SERVER_UNAVAILABLE 284
RPC_X_NULL_REF_POINTER 111
RPCDCE.H 241
RPCSS 72
RunAs 174

S

SCM 14, 73, 167, 230
SCODES 267
security 234
    custom 239

self-registration 168, 174
server 43
    adding a method 36
    context 88
    creation using ATL wizard 30
    in-process 14, 29, 178
    multi-threaded 203
    out-of-process 14
    registration 64
    running with client 40
service 15
Service Control Manager 167, 230
ServiceParameters 174
short 108
signed 108
single threads 155, 159
singleton classes 79
sink 184
size_is 113, 114, 115, 120
size_is(llen) 116
smart pointer
    classes 256
    error handling 258
smart pointers 255, 220
source 184
STA 155
stability 89
standard keys 170
standard marshaling 94
standard properties 129
stateful 164
stateless 164
static member 208
STDAFX.H 191
string 113, 114
strlen 113
struct 74
structure 121
stub 85
SUCCEEDED 22, 264
Support Connection Points 186, 216
surrogate 15
synchronization 151, 162
synchronous 190
SysAllocString 118
SysFreeString 118

T

TCP/IP address 232
template 247
    classes 250
    example 248
this pointer 210
thread 152
    apartment 155, 158
    free 155, 164
    single 155, 159
    testing different models 165
    worker 205
thread local storage 152
threading models 151, 153
ThreadingModel 172
THREADPROC 153
ThreadProc 208
TLH 260
TLS 152
TRACERT 287
transparency 69
TRG file 103
try/catch 259
type library 87, 95, 102, 126, 290
typedef 121
typelib 170
    header 260

U

UDP 230
UnAdvise 189
Unadvise 225
UNC 232
unicode 113
unique 5, 110
universal naming convention 232
Universally Unique Identifier 12
Unlock 227
unregistration 176
UnRegserver 177
unsigned 108

UpdateRegistryFromResource 177
User Datagram Protocol 230
user interface thread 153
uuid 12, 99

V

v1_enum 122
VARIANT 128
VARIANTARG 130
VariantInit 132
varying arrays 119, 121
VB run-time error 134
version 99
VersionIndependantProgID 172
very early binding 127
Virtual Function Table 75
Visual Basic 125
VT 132
    types 132
VTABLE 24, 75

W

wchar_t 108, 113
wcslen 113
Win32 Debug 41
Windows Service Control Manager 14
WINERROR.H 266
WinMain 177
WM_CLOSE 154
WM_QUIT 154
worker thread 205
    implementing 209
    simple 208
    starting 207
wrapper method 260