Coding Conventions for MFC Extensions

Coding Conventions

While this chapter is a precursor to a practical example of writing an MFC extension, you can apply the information contained to any project you care to write, whether you use the MFC or not. I suppose what I'm going to talk about is how Microsoft wrote MFC, giving you the opportunity to write code in the same style. The merits (or lack of) of this style have been debated heatedly several times in newsgroups and mailing lists, so whether you like it or not, it *is* Microsoft's style. So please, no flames!

What is an MFC Extension?

When you're developing Windows applications with MFC, you may not realize it, but 90% of the code you write is extending MFC. Since MFC is a set of 'Foundation' classes, an MFC application comprises a set of MFC derivatives that enhance MFC to provide the functionality you're after.

In short, an MFC extension is a C++ class that is derived from an existing MFC class and extends that class by adding some new (usually application-specific) functionality. MFC itself is mostly extensions. You can think of the 'core' MFC classes (**`CObject`**, **`CWnd`**, etc.) as the underlying framework and classes like **`CSplitterWnd`**, **`CView`** as extensions on this framework layer. Since MFC extensions borrow lots of their logic from MFC itself, they are the easiest way to extend MFC and add reusable functionality to your applications. If you've been around the Windows programming block, you've probably written controls or subclassed existing Windows controls to add new GUI functionality to your programs. MFC makes doing this much easier. The functionality provided in classes like **`CWnd`** can easily be extended by creating a derivative, overriding some virtual functions or handling various messages in a map. You no longer have to worry about things like the Windows process or making sure that all of the default windows procedures are getting called. MFC takes care of all this for you!

In this chapter we'll give you some helpful design techniques and rules of thumb to help you write better MFC extensions. Before we start, let's answer one of the most common questions that MFC developers are faced with:

How do MFC extensions compare with the OLE controls?

MFC Extensions versus OLE Controls

The answer to this question lies in the basic architecture of each approach. An MFC extension is usually tightly bound to MFC and can't be used in another programming environment like Delphi. An OLE control, on the other hand, is a very general component and can be used in a variety of applications, such as Delphi, Visual Basic and Visual C++. Every feature has a drawback, though. While OLE controls are compatible with many environments, they're not as flexible as something written specifically for one environment. You can't create an OLE control derivative and easily modify its behavior beyond the properties provided. Enough high level gobbledy-gook, let's look at an example.

Let's say that you're working on a real-time control application and you need a gauge display that shows

the heat of a nuclear reactor. If you write this gauge as an OLE control, developers working in any environment will be able to use it, but, if you write the gauge as an MFC extension, you can only use the control in an MFC application.

Now let's say you write the gauge as an OLE control and one of the users wants to add a little picture of a nuclear mushroom cloud instead of a 'too hot' area on the dial. If you haven't specified this as a property for your OLE control, the developer will have to come back to you and ask for you to enhance your OLE control. If you had implemented the gauge as an MFC extension with a `virtual OnDrawTooHotZone()` function, the developer would be able to easily extend the gauge to display the mushroom cloud.

The point is that it's best to use OLE controls when you're writing for multiple development tools, but MFC extensions are best when you want to provide much more flexibility and object-oriented features like inheritance. Another solution to the question of MFC extensions vs. OLE controls is to write MFC extensions first and then wrap them with an OLE control. This provides the best of both worlds.

Writing Great MFC Extensions

Now let's focus on some of the issues that you'll face when you're writing MFC extensions. In some cases, the decisions are very cut and dried. You use ClassWizard to create your new class and fill in the `//TODO` sections. You're extending MFC, but it feels just like writing C functions, because there aren't many object-oriented hurdles to leap.

However, once you start working on larger classes and leave behind the safety net of ClassWizard's `//TODO` sections, the number of decisions that you have to make about your MFC extension class can be mind boggling:

- What do you name everything?
- Which MFC class do you derive from?
- Should you derive from `CObject`?
- What should be `private`/`protected`/`public`?

If you want your classes to be useful to other developers, there are even more issues that you need to focus on for that object-oriented mantra, *reuse*:

- What member functions should be `virtual`?
- How can I best document my classes for readability?
- What C++ features should I use/avoid to help other developers?

The MFC Extension Golden Rule

The best way to approach MFC extension class design issues is to look at what Microsoft's own MFC team do in the MFC classes. If you follow the design conventions used by Microsoft, other MFC developers will stand a better chance of using and reading your class.

So, the MFC extension golden rule is:

Write MFC extensions that follow Microsoft's conventions.

You may have heard this before as, "When in Rome, do as the Romans do."

Microsoft don't reveal much about how they design MFC, but some fishing around the MFC source code shows that there is indeed a method to their madness.

MFC Coding Conventions

Before we delve into hard-core MFC class design issues, lets first look at the coding conventions used by the MFC team when they write classes. Even if you're just starting MFC development, you've probably noticed that ClassWizard automatically generates some comments in your class declarations. These comments are used by the MFC team to split the class declaration into the following subsections, which are usually in this order:

```
// Constructors
// Attributes
// Operations
// Overrides
// Implementation
```

When you're doing a quick **`CDialog`** derivative, these subsection comments may seem like overkill, but when you start writing classes with tens or hundreds of member functions and data, using them makes your life much easier and your class much more understandable.

Here's a quick run-down of what each class declaration subsection comment means and some examples of what you should put in that subsection:

`// Constructors` - obviously, your C++ constructors live in this subsection, but, not so obviously, this is where MFC usually puts any other 'initialization' type member functions, such as **`Create()`**.

`// Attributes` - member data lives here, along with member functions that manipulate (**`Set`** or **`Get`**) member data. These member functions are sometimes called **mutator** (**`Set`**) **accessor** (**`G et`**) functions.

`// Operations` - place member functions that other classes will directly call to perform an operation in this section. In some cases, there are enough operations that you might want to subdivide this subsection even further. (e.g. **`//Operations - Grid functions`**, **`//Operations - Graphing functions`**, etc.)

`// Overrides` - you should place functions that are meant to be overridden by derived classes here (i.e. **`virtual`** functions).

`// Implementation` - implementation-specific member functions should live in this subsection. In MFC, overridden functions from **`virtual`** base class members go in this section too, since they are considered an implementation detail.

One nice side-effect from the ordering of these subsection comments is that when another MFC developer looks through your class declaration, class members are presented in a very logical order, starting with their creation and ending with their implementation. Although these subsection comments cover the majority of class member functions that you'll write, there are some gray areas that might come up. For example:

Destructors - Microsoft places all destructors in the **`// Implementation`** section, even if they can

be overridden. They are usually the first entry in the `// Implementation` section.

Pure virtual functions - these should live in the `// Overrides` section, but you may want to make them stand out by adding a `// Overrides - pure virtual` subsection as well.

Message map functions - ClassWizard automatically places these below the `// Implementation` section in a section called `// Generated message map functions`.

Debug specific functions - most MFC classes have `Dump()` and `AssertValid()` member functions. Microsoft places these in the `// Implementation` section, even though some of them are overridable.

Variable Naming - Think Hungarian!

If you learned Windows programming reading the good ol' Petzold book, you'll already be very familiar with Hungarian notation. In a nutshell, it's a variable naming convention that prefixes variable names with the type of the variable. After the prefix, you should capitalize the next letter and also capitalize the first letter of any new words in the variable name. The following table gives a quick recap of the variable name prefixes suggested for the basic C++ types, with examples:

Type	Prefix	Example	Comment
`char`	`c`	`cDirSeparator`	
`BOOL`	`b`	`bIsSending`	
`int`	`n`	`nVariableCount`	
`float`	`f`	`fAngle`	
`double`	`d`	`dSalary`	
`UINT`	`n`	`nMyUnsignedInt`	
`WORD`	`w`	`wListID`	
`LONG`	`l`	`lAxisRatio`	
`DWORD`	`dw`	`dwPackedmessage`	
`*` (pointer)	`p`	`pWnd`	
`FAR *`	`lp`	`lpWnd`	Interchangeable with `p` under Win32
`LPSTR`	`lpsz`	`lpszFileName`	`z` indicates null terminated.
handle	`h`	`hWnd`	
callback	`lpfn`	`lpfnHookProc`	Pointer to a function

MFC adds some twists to the Hungarian naming conventions. To start with, you should prefix each class with `C` and member data with `m_`; for example: `CObject`, `m_hWnd`. Also, there are 'standard' prefixes for some of the more common MFC classes:

Class	prefix	Example

`CRect`	`rect`	`rectScroll`
`CPoint`	`pt`	`ptMouseClick`
`CSize`	`sz`	`szRectangle`
`CString`	`str`	`strFind`
`CWnd`	`wnd`	`wndControl`
`CWnd *`	`pWnd`	`pWndDialog`
`CDialog`	`dlg`	`dlgFileOpen`

To avoid collision with Microsoft classes and to make your classes stand out, I recommend that you come up with some other character or combinations of characters, rather than **`C`**, for the class name; for example, **`CWroxMyClass`**. If you use a company-specific class prefix, when you pass your application to Joe (he's the new guy with very little experience) he'll know that "Hey! **`CWroxMyClass`** must be one of our reusable MFC extensions!"

Symbol Naming Conventions

MFC also follows a strict set of conventions for naming resource symbols. ClassWizard/AppWizard will automatically generate these symbol prefixes, but knowing what they stand for and the ranges they cover will help you use the correct prefixes in your extension classes:

Type	Prefix	Example	Range
Shared by multiple resources	`IDR_`	`IDR_MAINFRAME`	1 - 0x6FFF
Dialog resource	`IDD_`	`IDD_ABOUT`	1 - 0x6FFF
Dialog resource help context ID (for context-sensitive help)	`HIDD_`	`HIDD_HELP_ABOUT`	0x2001 - 0x26FF
Bitmap resource	`IDB_`	`IDB_SMILEY`	1 - 0x6FFF
Cursor resource	`IDC_`	`IDC_HAND`	1 - 0x6FFF
Icon resource	`IDR_`	`IDI_ICON`	1-0x6FFF
Menu or toolbar command	`ID_`	`ID_CIRCLE_TOOL`	0x8000 - 0xDFFF
Command help context	`HID_`	`HID_CIRCLE_TOOL`	0x1800 - 0x1DFFF
Message box prompt	`IDP_`	`IDP_FATALERROR`	8 - 0xDFFF
Message box help context	`HIDP_`	`HIDP_FATALERROR`	0x30008 - 0x3DFFF
String resource	`IDS_`	`IDS_ERROR12`	1 - 0x7FFF
Control in dialog template	`IDC_`	`IDC_COMBO1`	8 - 0xDFFF

Technote 20 (InfoView/Visual C++ Books/MFC 4.1/MFC Technical Notes/MFC Technical Notes Index/TN020) gives more details on resource ranges. Be sure to read it if you're concerned about your resources colliding with those provided by whoever uses your MFC extension class.

Follow the Leader

If you understand and follow Microsoft's conventions for MFC class declaration, variable naming and symbol naming, you'll already have taken the important first step of making your classes much more

usable. Other MFC programmers will be able to immediately understand your class interface and should also be able to pick up on the implementation quickly.

Following these coding conventions is just the tip of the iceberg of writing great MFC extension classes. It's the design and behavior of the class, or the semantics, that generate the hard questions that the MFC extension class designer must answer.

Design Issues

The best way for us to look at the different issues that you'll will encounter as you design MFC extension classes is to answer the questions that you'll be asking yourself during the design process.

What C++ language features should I use/avoid?

MFC doesn't use any multiple inheritance and you should avoid it in your extension classes if you can. If you choose to use multiple inheritance, be careful! Since almost all of the MFC classes share the single `CObject` root, things get messy very quickly. (OK, now I think that's silly because now the user's going to want an example of how to get around this problem and that's another story for another day. 99% of people will do just fine without having to mess with multiple inheritance in MFC.) Also, MFC uses only `public` inheritance. For example:

```
class CMyView : public CView
```

Non-`public` inheritance provides stricter member access, but confuses many C++ newcomers. Follow the MFC convention and use only `public` inheritance in your extension classes.

Another source of confusion for C++ beginners are templates. Templates are good for writing generic well-typed classes, but you must weigh the complexity of using templates against the advantage of this type safety. In most MFC extension classes, using templates is definitely over-kill unless you're writing collections or some other class that you want to operate on a wide variety of data types.

Should I derive from CObject?

Yes, you should. The only cases that don't warrant `CObject` derivation are usually type-like classes like `CString`. For example, a complex number class, or a large integer class might benefit from `CObject` derivation, as it gives you serialization, while small helper classes, like locking resources, gains nothing.

When you derive from `CObject`, your class gains serialization, run-time class information and object debug diagnostics. You may be asking yourself, "But what's the cost of deriving from `CObject`?"

If you already have `virtual` functions in your classes, the cost of deriving from `CObject` is the addition of four more `virtual` member functions. Since you already have `virtual` functions, the cost of having a vtable pointer has already been added and four entries in your vtable is a small object-size price to pay for the features you gain from `CObject`.

In summary, the cost of `CObject` derivation is not as expensive as some people would have you believe, especially if you already have some `virtual` functions.

Which members should I make private/public/protected?

You may have noticed that MFC has very few private members. While this doesn't provide strict data hiding, it does let MFC users examine and modify the behavior of classes that they're using in ways the class designer might not have anticipated. For example, all of the window classes let you access the

m_hWnd member data, which lets you quickly and easily call the Windows API with the handle if you need to, but it's up to the user to not change the member data and totally hose the internals of the object. Avoiding the use of **private** class security will make your classes more usable for your users, but you have to trust them (some **ASSERT**s help too) not to modify member data in dangerous ways. Of course, if you're a die-hard, OO, Booch-diagrammer, you're probably cringing at this statement. That doesn't change the fact that this is how every class in MFC is implemented and we're just reporting Microsoft's style for writing MFC extensions. Of course it's entirely up to you to decide which (if any) of these conventions to follow.

As for **public** and **protected** class security, you can use the class definition subsections described earlier as a guideline:

// Constructors - public

// Attributes - public/protected

// Operations - public

// Overrides - protected (should only be called/overridden by derivatives)

// Implementation - protected/public/private

The **// Attributes** and **// Implementation** sections are where you have to make the most class security choices. If you think any user classes will need access to data members or implementation-specific members, use **public**. If you think that only derivative classes will need access, use **protected**. Finally, if you think some implementation-specific member is going to change soon, or you don't want to grant access to it, make it **private** (but remember that Microsoft uses **private** very sparingly).

Which member functions should be virtual?

The more **virtual** functions you provide, the more behavior a deriving class can modify for their use. For example, in MFC, many print-preview functions are not **virtual**, so to change the look and feel of print preview, you have to cut and paste code from the MFC source (which could change in the next version) and then modify it in your own routines. Most areas of MFC don't have this problem, but you can avoid it by making everything but very implementation-specific functions **virtual**. Here are the member functions that Microsoft makes **virtual** by class definition subsection comment:

// Constructors - these functions aren't **virtual** as the derived class calls the base constructor.

// Attributes - these aren't **virtual** as they shouldn't change.

// Operations - can be **virtual** or not; it's up to you as the designer.

// Overrides - always **virtual**.

// Implementation -can be **virtual** or not; it's up to you as the designer.

Finally, if your class uses **virtual** functions, make sure that it has a **virtual** destructor, so you can destroy objects without knowing their type. The **virtual** (polymorphic) mechanism will call the correct destructor for you. Note that Microsoft use all **virtual** destructors for this very reason.

Should I provide a copy constructor?

No. You should avoid them because they confuse the user and don't fit in with MFC's two-phase construction (see the next question).

If you haven't been exposed to copy constructors, they are declared to take a **const** reference to the object and are used by the compiler to create copies of an object. If no copy constructor is provided, the compiler uses a member-wise copy. In MFC, only non-**CObject** objects (i.e. type classes) have copy constructors. In fact, **CObject** has these lines which makes sure a **CObject** derivative never calls a copy constructor:

```
private:
   CObject(const CObject& objectSrc);            // no implementation
   void operator=(const CObject& objectSrc);     // no implementation.
```

Why does Microsoft do this? The reason is because there is no clear semantic meaning for something like:

```
CWnd myCwnd = yourCWnd;
```

So, instead of letting the MFC user get into trouble by trying something like this, Microsoft make both the copy constructor and = operator **private**, so that, instead of undefined behavior, you'll get a compiler error when you try something like this.

The only MFC extensions that you would need copy constructors for are type-like classes (**CString**) that you choose not to inherit from **CObject**.

What constructors should I provide?

In MFC's two-phase construction model, Microsoft provide a default constructor (one that takes no arguments) that initializes the C++ object and also a **Create()** function that takes arguments for initializing the Windows object (the graphical part of the class, e.g. window creation etc.). You should follow this technique in any framework extension classes. The exception is type-like extensions. You might want to provide constructors that take arguments, since two-phase construction isn't really necessary.

For example, a **CView** derivative should definitely adhere to MFC's two-phase construction, but a complex number class' constructor would most likely take two arguments: the real and imaginary numbers.

How should I pass parameters to member functions?

MFC follows different rules of thumb depending on the purpose of the parameter (input/output/input-output) and the type of the parameter (i.e. is it a framework class or a type/type-like class?).

In general, since you can't copy framework objects, you should never pass them by value. Passing by pointer works, but can be misleading because it appears that the function is going to modify the parameter to give a result. You should pass by **const** pointer.

For type and type-like parameters, if the data is small, you should pass by value. If the data is large, you should pass by **const** reference instead.

Here are some examples of how to apply these rules:

For input parameters:

If you're using a type or type-like class, pass it by value. If the object is large, pass by **const** reference:

```
void MyFunction(int nType); void MyFunction(const CBigDataRecord & bigtype);
```

For framework objects, pass in as a **const** pointer:

```
void MyFunction(const CMyView * pMyView);
```

For output parameters:

Pass always as a pointer:

```
void RetrieveData(int* pInt);
void RetrieveData(CMyView* pView);
```

Note that the lack of **const** informs your callers that you'll be modifying the data.

For input-output parameters:

You should avoid passing types/type-like objects as input-output parameters, as the side effect of both using and changing a parameter is often unclear to the caller. If you must, use a pointer without a **const**:

```
void UpdateCount(int *nCount);
```

For framework objects, use a pointer too:

```
void UpdateView(CMyView *myView);
```

Should I return values or pointers?

For framework classes, the return type should be a pointer to the object. For type-like classes, you should return the value of the object. Some examples:

```
CRect GetMyRect();
int  GetNumDocuments();
CPoint GetCenterPoint();
CView * GetCurrentView();
```

What operators should I provide?

Usually, in framework classes, there is no logical meaning for most operators (e.g. **CMyView + CYourView**, or **CMyDocument == CYourDocument** are meaningless operations), so you shouldn't have operators in your framework classes. No operators are defined in the MFC framework classes.

In your type-like classes, you can follow the MFC type-like classes, **CRect**, **CSize**, and **CPoint** as examples. They define operators that make sense for the type and also serialization operators.

Should I use char * or CString?

Internally, you should use **CString** whenever possible because it handles all memory allocation, localization and destruction for you. Since you want your classes to be very flexible, you may want to provide both **char*** and **CString** versions of some member functions so that you can handle both. For example, in MFC's **CDC**, there are two versions of **TextOut()**:

```
virtual BOOL TextOut(int x, int y, LPCTSTR lpszString, int nCount);
BOOL TextOut(int x, int y, const CString& str);
```

In the implementation, the **TextOut()** that takes a **CString** performs a cast which invokes the **CString** to **char*** conversion operator and then calls the **virtual TextOut()** which takes a **char***:

```
BOOL CDC::TextOut(int x, int y, const CString& str)
{
   ASSERT(m_hDC != NULL); return TextOut(x, y, (LPCTSTR)str, str.GetLength());
} // call virtual
```

This 'trick' keeps your users from having to typecast between **char*** and **CString** when they call your member functions.

What accessor/mutator functions should I provide?

Some object-oriented gurus suggest that you make all member data **private** and provide access only through accessor /mutator (i.e. **Get**/**Set**) functions. MFC takes a more laid back approach by making the member data **public** and then providing access functions for the data that 'add value'. For example, in class **CWnd**, you can get directly to the member data via **m_hWnd**, or you can call the access function **GetSafeHwnd()**. This accessor function adds value by verifying the state of the class and the state of the window handle before returning it. It's left up to the class user to decide whether or not they need this level of checking for their particular situation.

Which class members should be declared const?

Apart from using **const** to protect arguments against side affects, you can make entire member functions **const**, to guarantee that they don't change the object member data. MFC designates all **Get** accessor functions as **const** and other functions that are appropriate.

Some examples of **const** member functions in MFC are:

```
CMenu* GetMenu() const;
CMenu* GetSystemMenu(BOOL bRevert) const;
BOOL IsIconic() const;
BOOL IsZoomed() const;
BOOL operator==(POINT point) const;
BOOL operator!=(POINT point) const;
```

If you know that a function is not going to change any member data, marking it **const** will ensure that a derived class doesn't override the function and add dangerous side effects to it. It also gives users a valuable clue about the member function and its behavior.

Data members that are **const** are vary rare, since the data usually changes during the life of an object MFC only uses them for message map entries and interface maps. You probably won't need **const** data members in your class unless you're really pushing the MFC envelope.

Which class members should be static?

MFC only has a few **`static`** class member functions. Because there's only one copy of the state of a **`static`** member function for multiple instantiations, the static functions in MFC are functions that don't pertain to a single instance of a class. For example:

```
static CWnd* PASCAL FromHandle( HWND hWnd );
static CGdiObject* PASCAL FromHandle(HGDIOBJ hObject);
static CBitmap* PASCAL FromHandle(HBITMAP hBitmap);
static CGdiObject* PASCAL SelectGdiObject(HDC hDC, HGDIOBJ h);
```

The behavior of these functions depends only on the arguments and not on the current object they are being called against.

MFC declares all message maps, which are members, as **`static`**. Aside from that, there aren't many data members in MFC that are non-unique between different instances.

Use **`static`** members in your classes to conserve space like MFC does, and be sure that you are not making something that is instance-specific into a **`static`** member. If you're going to be doing any multithreaded programming, you should also consider making the **`ThreadProc()`** function a **`static`** member. This puts it in the namespace of the class, but still allows you to take the address of the function properly to pass to **`CreateThread()`**.

Should I provide default arguments?

Definitely! MFC uses default arguments to make the user's life much easier. In most cases, you only have to supply MFC framework member functions with a couple of arguments and the defaults are sane enough so that you only have to provide non-default arguments a small percentage of the time.

For example, **`CWnd::SendMessageToDescendants()`** is declared:

```
void SendMessageToDescendants(UINT message, WPARAM wParam = 0,
   LPARAM lParam = 0, BOOL bDeep = TRUE, BOOL bOnlyPerm = FALSE);
```

So, in cases where the caller doesn't need to supply a **`wParam`**, **`lParam`**, **`bDeep`** or **`bOnlyPerm`** argument, C++ fills in the defaults automatically.

Use default arguments in your member functions as much as possible and be sure to document them.

Lessons from the Field

When we put out our first MFC extension class library, we followed the previous rules of thumb to the letter. After releasing the libraries, we quickly got feedback from our users that made us amend the rules of thumb with some lessons learned the hard way:

ASSERT Yourself

Be liberal with **`ASSERT`**ions. Check all arguments and internal state where applicable. The sooner you can let the user know that they've made a mistake, the better.

Be sure to include an **`AssertValid()`** and **`Dump()`** for every MFC extension you write. Your **`AssertValid()`** should validate the state of the class' data members and **`ASSERT`** if there's a problem. Developers using the **`ASSERT_VALID`** macro will appreciate you including this member. In your **`Dump()`** member function, display the state of the class in a **`printf()`** style. Not many people use the dump diagnostics, but, in a sticky situation, they may resort to this and your class will need to help by displaying its state.

Leverage MFC

Use **`CDC`**, **`CString`** and other MFC classes internally whenever possible. There are thousands of lines of well-written and tested code in MFC, so leverage them whenever possible and avoid using Windows API calls if you can. Sure, you're a macho programmer from the Windows 2.x days and can do things much better than MFC, but sticking to MFC is a good idea because everyone else on the project probably won't understand your more arcane approach to Windows programming (*"Why is he using a switch for messages?!*").

Const Correctness

The **`const`** keyword is both powerful and rather confusing in C++ programming. Microsoft uses **`const`** carefully in MFC to give the class user a hint about what a member function or argument is doing. Here's a look at some of the best uses of **`const`**:

Use **`const`** to let a member function caller know that an argument won't be modified. For example:

```
void SetWindow(const CWnd * pWnd); //Won't change pWnd on you!
```

Mark member functions with **`const`** if they don't change the state of the object (e.g. if they don't change any data members). For example:

```
CTime GetTime()  const;
```

This immediately lets the class user know that they can call this member function without having to worry about any side affects.

Use **`enum`** and **`const`** declarations instead of **`#defines`** if you can. It's much safer to have:

```
typedef enum {MODE_ONE,MODE_TWO} myModeType;
void SetMode(myModeType newMode);
```

rather than

```
#define MODE_ONE 1
#define MODE_TWO 2
void SetMode(int nMode);
```

With the first version, C++ will catch any modes that don't exist and in the second version, you'll have to check that the modes are in range. When you're using **`enum`**, instead of making them global, place them inside of a class declaration if they are class-local. This causes less name space pollution.

Also, use this,

```
const int MIN_INT = 2500;
```

instead of:

```
#define MIN_INT 2500
```

Again, the C++ type checking will catch type mistakes for you and your user.

Virtual Functions Rule!

The votes are in and MFC class users unanimously vote for more virtual functions. Here are some lessons we learned about **`virtual`**s:

> Class users will customize things you never imagined. Try and break up every class operation into smaller functions and make key functions **`virtual`**. For example, instead of having a big **`Draw()`** routine that draws a gauge, break the routine into **`virtual`** members, such as **`DrawBackground()`**, **`DrawNeedle()`**, **`DrawTicks()`**, **`DrawLabel()`**, etc.. This gives users more opportunities to jump into the drawing loop and customize to their heart's content.
>
> Use **`virtual`**s as callbacks instead of sending messages or calling **`CALLBACK`** functions. For example, if you were writing a serial communication class, you could internally call a function **`DataReady()`**. While your implementation of **`DataReady()`** doesn't do anything, the user can override this virtual function in a derivative and do something in **`DataReady()`**.
>
> Instead of embedding an object in your class, create a **`virtual`** function that returns the object. This lets users override the **`virtual`** to return a customized derivative (polymorphism at its finest!), thus letting them customize the embedded object. OK, that description is a little hard to grab the first time, so here's an example. Let's say you write a class that creates a file dialog. The first reaction is to write:

```
CMyClass::MyFunction(int nSomething)
{
   //...
   //Embedded member

   if  (myDialog.DoModal() != IDOK)
      return;
   //...
}
```

> But what if the user has a specialized **`CFileDialog`** for their application that provides a preview of

files? Instead of embedding objects as above, you could give the user the flexibility to override this embedded object by writing the above snippet as:

```
class CMyClass : public CSomeClass {
   //
   // CFileDialog customization members
   virtual CFileDialog * GetFileDialog();
   virtual void ReleaseFileDialog(CFileDialog* pDlg);
       // ...
};

CFileDialog * CMyClass::GetFileDialog()
{
   CFileDialog pDialog = new CFileDialog(TRUE,"*.ext","foobar.ext",0,,this);
   //TODO - verify
   ASSERT(pDialog != NULL);
   return pDialog;
}

void CMyClass::ReleaseFileDialog(CFileDialog * pDlg)
{
   ASSERT_KINDOF(CFileDialog,pDlg);
   delete pDlg;
}

CMyClass::MyFunction(int nSomething)
{
   //…
   //Use to have an embedded member…
   CFileDialog * pDialog = GetFileDialog();
   ASSERT_KINDOF(CFileDialog,pDialog);

   BOOL bResult = (pDialog->DoModal() == IDOK);
   ReleaseFileDialog(pDialog);
   //…
}
```

Now, instead of having to hack up the **`MyFunction()`** code, the user can add a custom **`CFileDialog`** derivative by overriding the **`virtual GetFileDialog()`** and **`ReleaseFileDialog()`** members. For an example of where MFC itself uses this technique, check out **`CDocument::GetFile()`**, **`CDocument::ReleaseFile()`** and **`CDocument::OnOpenDocument()`** in **`Mfc\Src\Doccore.cpp`**.

The bottom line is you can't have too many **`virtual`** functions. It's nearly impossible to get this right the first time, but try putting yourself in a class user's shoes and look at your class from different angles to make sure that you've built in flexibility.

Internationalization

The subject of internationalization in MFC is enough to fill several chapters of this book, so here are some quick tips and pointers to other information:

DBCS or UNICODE? Since Win95 doesn't support UNICODE, using this makes internationalization much more complex.

How complex? Check out the Visual C++ Technotes 57 and 59.

Place any end-user visible strings in a resource. This makes internationalization much easier, since you won't have to search around the source code for all of the strings.

If you don't follow the above suggestion, use the **`_T`** macro to convert your inline strings to UNICODE strings. For example: **`_T("blah");`**

`CString` is already internationalized, so use it instead of **`char*`** type strings.

Use **LPTSTR** instead of **LPSTR**.
Use **LPTCSTR** instead of **LPCSTR**.
Use **TCHAR** instead of **char**
All of the above are **#define**d to different values depending on a UNICODE definition.

Multithreading

You'll often come across the question, "What do you need to do to make a class multithread safe?" In MFC, the philosophy is that a class is guaranteed to work inside a thread, but not to be sharable between threads. In the sharing case, it's up to the class user to protect any critical data with synchronization objects. A clear example of this philosophy is the MFC **CString** class. It has no special code for multithread support. However, MFC maintains internal maps of window handles to **CWnd**s that has tons of synchronization code so that it will work properly when accessed between different threads.

Portability

When you were writing classes for earlier versions of MFC, portability wasn't a concern, but today's MFC has leapt the surly bounds of the Windows operating system and now acts as a portable class library for the Macintosh using Microsoft's cross-platform Visual C++ system, and even to UNIX using third party tools. For most of you writing MFC extensions, this isn't a big deal, but if you do have plans to move to one of these non-Windows environments, here are some pointers:

Use **#ifdef _MAC_** for the Macintosh and **#ifdef _UNIX_** for UNIX platforms.
The Win32 implementations on these platforms generally lag Windows by six months to two years. For example OLE, OCX, MAPI, TAPI, ODBC and other Win32 extensions are probably pushing the envelope. This means that you'll have to either choose a safe subset of functionality or be prepared to **#ifdef** out areas that use additional functionality. Be sure to research your options before going too far down the path and having to rip out some key feature because it's not supported on another platform.
Use **_MFC_VER_** for any MFC-version specific differences if there are any.

Packaging

The 'neatest' way to package an MFC extension is to write a Component Gallery Gizmo. Unfortunately, the interface to the Component Gallery is not public knowledge, so you have to resort to less exciting packaging, such as Extension DLLs (formerly known as AFXDLLs). This is another topic that could take a chapter on their own, so check out the VC++ Technote 33 for more information on Extension DLLs and how to build one. In a nutshell, an Extension DLL is a special DLL that exports MFC classes and shares an MFC DLL with your application. MFC automatically initializes Extension DLLs for you and adds the resources in the DLLs to its list of places to look for resources.

Putting it All into practice

Once you've read this chapter, you might want to open up the MFC header files **Afx.h** or **Afxwin.h** and search through them to locate an example of each MFC extension class design convention presented. Then review your classes and see which conventions you follow and which you don't. In your next class, try to use some of the conventions that you haven't used before. Who knows, if your MFC class design skills are good enough, maybe Microsoft will eventually hire you for the MFC team! Once you know and

understand the conventions that MFC follows and start using them in your classes, you'll be able to spend less time on class design and more time on class implementation. Plus, your class users will thank you for writing classes that are easier to use and understand.

Summary

In this chapter, we've looked at the 'theory' of MFC extensions and some techniques for writing them. In the next chapter, we put the pedal to the metal and look at some real-world examples of MFC extensions, review some of the techniques introduced here and how they apply to these real-world classes. Stay tuned!

Real-world MFC Extensions

The Game Plan

Now that we've looked at some tips for writing great MFC extensions, let's get to work and actually write some, so that you get a feel of what's involved. In this chapter, we'll design and implement two 'real-world' extensions (this means that they're not just some cheesy busy-work classes, but actually cool new classes you can start using in your MFC programs today!). Along the way, I'll try and show you some of the thought processes that go into each class, along with some pitfalls to avoid and even some new hints that we didn't cover in the last chapter.

If you're impatient and want to start using the extensions, you can fire up the CD from the book and pull them out of the `Source\Chap02` directory. If you do that, I still urge you to read along and see how the classes are developed. You could learn a lot here as we work through some of the decisions that are made along the way.

Abandon Your Wizards!

Many MFC developers are hopelessly addicted to wizards. They can't generate a single class without the help of ClassWizard, or an MFC application without using AppWizard. Now is the time to break the Wizard crutches and start programming the old fashioned way. Why? Well, ClassWizard isn't particularly helpful when it comes to extending MFC in ways that it's not familiar with. For example, what if you want to create a `CStatusBar` derivative? Since this class isn't one of the base class options in ClassWizard, you're on your own.

Once you start writing serious MFC extensions, you'll want to know exactly what's going on, so you'll need to wean yourself from the wizards. With this in mind, take note that you won't be seeing any wizard screen shots in this chapter!

Building a Better Status Bar

The MFC status bar is a pretty powerful tool that helps you set up panes of text which are updated depending on different program states. MFC gives you a great deal of functionality in the status bar 'for free'. For example, when the user moves through a series of menus, MFC automatically displays the menu prompt in the status bar to give them a clear indication of what the menu does. The interface to the status bar could be clearer, but most MFC programmers get around that by writing a quick routine that displays some text in a pane, so they don't have to remember the rather arcane ways to do this through the `CStatusBar` class.

Visual C++ even includes a component gallery wizard that adds a clock to a status bar that is updated every second. What more could you want???

Have you ever noticed when you load a project or large-ish file in Visual C++ that the status bar displays some text and a progress bar? Once the operation is over, the status bar resorts to the more familiar MFC-like status bar that we all know and love. In fact, several Microsoft products exhibit this same behavior (try it with Word, for example). Unfortunately, MFC's `CStatusBar` doesn't support a 'progress' mode that lets you change the status bar into a quick progress-status bar.

Well, this certainly isn't fair. Why should Microsoft's applications have a nifty feature like this and not yours? Let's put this right by extending MFC!

Designing a Progress-status Feature…

Wise men say that only fools rush in, and extending MFC is no exception. Before we start banging out the code for our cool new feature, let's think about what class in MFC we want to extend.

Usually, an application's different 'bars' (status and tool) live in the **CMainFrame CFrameWnd** derivative class. The first impulse may be to add some new members (e.g. **DisplayProgress()**, **IncreaseProgress()**, **HideProgress()**) to **CMainFrame** that handle the progress-status feature. This makes sense for several reasons:

> The **CMainFrame** is easy to access from anywhere in the code with MFC's **AfxGetMainWnd()** call. So, anywhere in your code, you could call something like:
>
> ```
> ((CMainFrame*)AfxGetMainWnd())->DisplayProgress();
> ```
>
> This is certainly handy because most of the actions that you'll want to display a progress bar most likely won't be in the **CMainFrame**, but in a **CDocument** or **CView**.
>
> Since **CStatusBar** is embedded inside **CMainFrame**, there is access to the **CStatusBar**, so you can easily get to its windows and draw the progress bar.

This is the perfect design, so let's get started…

Not!

OK, I have to admit that I've led you down the wrong path so far. Can you guess why this isn't the best way to go? The problem is the **CMainFrame** solution is not object-oriented at all! What happens in six months when you're working on a new product and you want it to also have the progress-status feature? You have to copy all of that **CMainFrame** code out of the current application and paste it into the new one. In the process, you're probably going to forget some code, have to dig back into the old code and waste a lot of time on a feature that you've already implemented.

Also, what if Fred down the hall in the WinWin product group wants to use your progress-status feature on his project? You'll have to point him to the important parts of your **CMainFrame** class and explain to him how the whole thing works. When you're extending MFC, you have to get rid of those old C Windows programming habits and think more object-oriented....

A Better Design

A much better solution to the problem is to extend the **CStatusBar** class. After all, aren't we adding functionality to the status bar and not the frame window? Now, if Fred starts bugging you for your progress-status code, you can e-mail him your enhanced **CStatusBar** and point out the enhancements. The code is more self-contained and also easier to manage, so you're bound to get a promotion! In honor of the book you're reading, let's call our **CStatusBar** derivative **MCStatusBar,** or the Master Class Status

Bar! Once you've decided where in the MFC class tree to plant your extension, the next step is to design the interface.

> *This approach is more object-oriented, because we're encapsulating all the functionality of the new status bar inside the* **`MCStatusBar`** *class.* **`CMainFrame`** *itself doesn't need to know how to manipulate the progress control directly. You may still have* **`CMainFrame::DisplayStatus()`**, *but this would simply call* **`MCStatusBar::DisplayStatus()`**.

The Progress-status Interface

There are a pretty small number of 'operations' that anyone who uses **`MCStatusBar`** will need to call:

1. Start the progress-status bar. This clears the current status bar and displays some progress-status text, such as Saving: or Reading Project:. At this point, the user will probably want to set up the size of the progress bar and perform other initializations. Let's go with **`StartProgress()`** as a good first name for this action.

2. Increment the progress-status-bar. You would make this operation in your saving or loading code that increments the progress-status bar indicator and lets the user know what percentage of the action is complete. Let's call this operation, **`IncrementProgress()`**.

3. Stop the progress-status bar. Once the action whose progress is being shown in the progress-status indicator is complete, the class user will want to return the status bar to its pre-progress state. Yep, you guessed it, we'll call this **`StopProgress()`**.

We could go crazy and add some accessors and virtual functions, but remember—these class interfaces usually pop out after you've started writing code and find a nice member function that can be used for customization, or a data member that needs an accessor.

So, keep the interface minimal. Don't add stuff unless you're sure that you need to.

Since this is our first extension, let's keep it simple for now and go with these three operations and see where the design takes us from there. Now we can start thinking about how we're going to write these operations.

Remember the Golden Rule

Take a few minutes to flip back to the last chapter and reread the MFC mantra, also known as the golden rule, *Write MFC extensions that follow Microsoft's conventions*. Now with the golden rule in mind, let's think about how we want to go about drawing a progress bar in a base class **`CStatusBar`**. There are two approaches I can think of:

Be macho and write the drawing logic from scratch. This would be something like:

1. Create a progress area with a bevel that holds X progress 'ticks'.
2. Draw the 'X' ticks in the **`IncrementProgress()`** using a blue pen (user configurable maybe?).

3 After drawing, erase the area and then redraw the status bar.

MFC already has a class for drawing progress bars: `CProgressCtrl`. I wonder if we could use that here? If we could, we could do something like:

1 Create the `CProgressCtrl`.

2 Make some calls to it.

3 Destroy it when we're done.

What would Microsoft do? Of course, they would use the second approach because there's no reason to recreate all of the logic that is nicely encapsulated by `CProgressCtrl` unless we hit some roadblock that keeps us from using a `CProgressCtrl` in a `CStatusBar` derivative.

In fact, since we've decided that `CProgressCtrl` is going to be integral in `MCStatusBar`, let's think again about our class interface. Instead of start/stop progress members, we could create/delete `CProgressCtrl`, using the MFC new/create/delete paradigm. `CStatusBar` already has the members and we could override them to take new arguments. The nice thing about keeping the start/stop progress member functions is that it lets us keep the progress operations separate from the general creation logic of `CStatusBar`. For example, what if someone uses `MCStatusBar` and doesn't want to create a progress bar? Another argument for keeping separate progress member functions is that there could be other functionality added in the future, and it would be easier to add more operations instead of changing the `Create()` every time and potentially breaking old code. Let's stick to the original plan.

A `CProgressCtrl` has a range with an upper and lower limit. These are all good arguments to pass through the `StartProgress()` function. `CProgressCtrl` can also have a different 'step' which defines how many progress indicators are used to fill the progress area. The default is ten. This could be a good default argument for `StartProgress()`, but it's also something that someone might want to call after `StartProgress()`, so let's add a new operation, `SetStep()`,which lets the user set the step any time after they've called `StartProgress()`.

`CProgressCtrl` has two calls similar to the `IncrementProgress()` operation that we need in `MCStatusBar`:

`StepIt()` This increments the progress indicator by the current step amount (set with `SetStep()`).
`SetPos()` This sets the progress indicator to a specified amount. For example, if a progress bar had a minimum of 0 and a maximum of 100, `SetPos(50)`, would set 1/2 of the progress indicator full (I'm an optimist).

Since any `MCStatusBar` user could also be familiar with `CProgressCtrl`, let's rethink the `IncrementProgress()` member function and, instead, go with `StepIt()` and `SetProgressPos()`.Why `SetProgressPos()` and not `SetPos()`? Well, in the context of `MCStatusBar`, `SetPos()` could mean *set the position of the status bar or a frame*. Naming the member function `SetProgressPos()` clearly indicates what's going to happen and keeps some remnants of the `CProgressCtrl` interface.

Remember the hint from the last chapter to leverage as much of MFC as possible? Using `CProgressCtrl` and some of its class interface is a prime example of that.

The MCStatusBar Header

Now that we've designed the class' interface and made some key implementation decisions, it's time to write the header for the class:

```
#ifndef _MC_SBAR_H_
#define _MC_SBAR_H_

///////////////////////////////////////////////////////
// MCStatusBar class for VC++ MasterClass

class MCStatusBar : public CStatusBar
{
// Construction
public:
    MCStatusBar();

// Attributes -none
public:

// Operations
public:

    int SetProgressPos(int nPos);
    int StepIt();
    int SetStep(int nPos);
    void StopProgress();

// Overrides -none

// Implementation
public:
    virtual ~MCStatusBar();

#ifdef _DEBUG
    virtual void Dump(CDumpContext& dc) const;
    virtual void AssertValid() const;
#endif //_DEBUG

protected:
    // Pointer to our progress control.
    CProgressCtrl * m_pProgressCtrl;

    DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////
#endif //_MC_SBAR_H_
```

Some points of interest from the **MCStatusBar** definition are:

Notice that the class uses the MFC subsections introduced in the last chapter.
Every member function name, argument and member variable uses the extended Hungarian notation.
The interface we designed has been added to the **//Operations** section where it belongs.
There's a virtual public destructor in the **//Implementation** section.
There is a protected **CProgressCtrl** pointer in the **//Implementation** section.

We're not doing anything fancy, like redefining lots of the **CStatusBar** member functions. Be sure to keep things as simple as possible and avoid unnecessary complexity whenever you can.

MCStatusBar Implementation Details

Now let's look at the implementation of each of the **MCStatusBar** member functions. One important implementation decision is how to create/destroy the **CProgressCtrl**. One implementation could create

the control in the **MCStatusBar** constructor and destroy it in the destructor, but this could be wasteful if the user never creates a progress-status.

Another implementation would be to create the **CProgressCtrl** in **StartProgress()** and delete it in **StopProgress()**. What if the user wanted to draw 100 different progress-status indicators over the course of the programs invocation? Creating and destroying that many **CProgressCtrl** objects could be pretty time-consuming.

A good solution to the shortcomings of the above solutions is to create a **CProgressCtrl** in **StartProgress()** and cache it until the **MCStatusBar** is destroyed where it can be freed. With this caching scheme in mind, let's look at the implementation of each **MCStatusBar** member function.

MCStatusBar::StartProgress() Implementation

```
BOOL MCStatusBar::StartProgress(int nLower, int nUpper,
    CString strCaption)
{
    //Hey the user needs a progress bar - create him!
    if (m_pProgressCtrl == NULL)
        m_pProgressCtrl = new CProgressCtrl;


    ASSERT(m_pProgressCtrl != NULL);

    SetWindowText(strCaption);
    UpdateWindow(); //force update...

    CDC* pDC = GetDC();
    CSize sizeCaption = pDC->GetTextExtent(strCaption);

    CRect rectSB;
    GetWindowRect(&rectSB);

    CRect rectProgress;
    rectProgress.left = sizeCaption.cx + 10;
    rectProgress.right = rectProgress.left +
        rectSB.Width()/2; //about 1/2 width
    rectProgress.top = 5;
    rectProgress.bottom = rectProgress.top + rectSB.Height() - 5;

    // Create the control
    BOOL bResult = m_pProgressCtrl->Create(0, rectProgress, this, 1);
    ASSERT(bResult == TRUE);

    m_pProgressCtrl->SetRange(nLower, nUpper);
    m_pProgressCtrl->ShowWindow(SW_SHOW);

    return bResult;
}
```

StartProgress() is really the meat and potatoes of **MCStatusBar**, so let's take a second to analyze it in detail.

First, the function checks to see whether a **CProgressCtrl** has been created already. If not, a new **CProgressCtrl** is created and stored in the **m_pProgressCtrl** cache variable. Next, there's a seemingly redundant **ASSERT** that double checks that the **m_pProgressCtrl** is non-**NULL**.

If you have some potentially confusing logic in a function, an extra **ASSERT** is always a good tool to help keep anyone using or reading through your class on track about what the state should be at that point in time. It's also a good sanity check to make sure you haven't messed up or some derivative has caused

some change in important behavior.

After creating the **CProgressCtrl**, **StartProgress()** sets the text of the status bar to the **strCaption** string argument. Then **StartProgress()** makes a call to **UpdateWindow()** to force the status bar to change.

Once the text is displayed, **StartProgress()** makes some calculations to determine where to place the **CProgressCtrl**. First, the size of the caption string is used to determine where on the left to start the progress control. Next, the length of the progress control is calculated to be about 1/2 the length of the status bar. Finally, the top and bottom of the progress control are based on the height of the status bar, plus a small border so that the progress control is clearly contained within the status bar but is still visible.

Once the size of the progress control has been calculated, **StartProgress()** creates the **CProgressCtrl** with **this** (which is the status bar) as the parent window and the calculated rectangle as the dimensions of the control.

Next, once the **CProgressCtrl** has been created, **StartProgress()** verifies that it was created with an **ASSERT** (***always*** call **ASSERT** after **Create()**; you'll track down many problems this way), then calls **SetRange()** on the object to set the lower and upper bounds.

Finally, **StartProgress()** calls **CProgressCtrl::ShowWindow()** to display the **CProgressCtrl** inside of the status bar.

MCStatusBar::SetStep(), MCStatusBar::StepIt() and MCStatusBar::SetProgressPos() Implementations

```
int MCStatusBar::SetStep(int nPos)
{
    ASSERT(m_pProgressCtrl != NULL);
    // ASSERT->Did you call StartProgress First?
    return m_pProgressCtrl->SetStep(nPos);
}

int MCStatusBar::StepIt()
{
    ASSERT(m_pProgressCtrl != NULL);
    // ASSERT-> Did you call StartProgress first?!
    return m_pProgressCtrl->StepIt();
}

int MCStatusBar::SetProgressPos(int nPos)
{
    ASSERT(m_pProgressCtrl != NULL);
    // ASSERT-> Did you call StartProgress first!?
    return m_pProgressCtrl->SetPos(nPos);
}
```

SetStep(), **StepIt()** and **SetProgressPos()** are protected calls through the m**_pProgressCtrl CProgressCtrl** pointer. Before calling through the data member, each function **ASSERT**s to make sure that the user didn't call one of the operations before they called **StartProgress()**. Another option would be to detect this situation and then automatically call **StartProgress()**, but let's stick to our basic interface for now.

Note that the **ASSERT**s here are followed by a helpful comment that suggests to the user what is most likely going wrong. It's always a good idea to describe possible **ASSERT** causes in your code after an

ASSERT if you have an idea of what could be wrong. If the class user hits the **ASSERT**, he or she will hopefully fire up the debugger and see the comment that we've placed near the assertion. Most MFC programmers are familiar with this procedure, since Microsoft usually comments each of their **ASSERT**s with some possible reasons for the **ASSERT**.

> *You should also not that using* **ASSERT**s *doesn't protect your code in release build. Unless you can guarantee that your code is fully tested, you may still have a situation that results in one of these functions being called with* **m_pProgressCtrl** *set to* **NULL**. *With these functions, you can't return an error condition back to the calling function, as a correct return value can be any valid integer. If you want to add more protection, you'll have to throw an exception if* ***m_pProgressCtrl*** *is* ***NULL***.

MCStatusBar::StopProgress() Implementation

```
void MCStatusBar::StopProgress()
{
    ASSERT(m_pProgressCtrl != NULL);
    //ASSERT->Did you call StartProgress first?

    //Don't delete this guy - we've already incurred
    //the expense of displaying it, why do it again?
    //If the user's used it once, likely to do so again..

    // Cache that puppy and reset to the
    // CProgressCtrl defaults
    // for next time.

    m_pProgressCtrl->ShowWindow(SW_HIDE);

    // Refresh the status bar
    SetWindowText(NULL);
    UpdateWindow(); //force update

    //Nuke its window
    m_pProgressCtrl->DestroyWindow();

}
```

MCStatusBar::StopProgress() first hides the progress control, clears the caption from the status bar by calling **SetWindowText(NULL)** and then calls **UpdateWindow()** to make the status bar redraw itself and its panes. After redrawing the status bar, the Windows window is destroyed by calling **DestroyWindow()**.Note that the **CProgressCtrl** object is not destroyed, but the Windows window associated with it is. This resets the **m_pProgressCtrl** object to be reused in the next call to **StartProgress()**.

MCStatusBar Constructor/Destructor Implementation

```
MCStatusBar::MCStatusBar()
{
    m_pProgressCtrl = NULL;
    //Create on an 'as-needed' basis...
}

MCStatusBar::~MCStatusBar()
{
    //Nuke it if it was ever created
    if (m_pProgressCtrl != NULL){
        delete m_pProgressCtrl;
        //Be sure to reset to NULL
        m_pProgressCtrl = NULL;
    }
```

```
}
```

The **MCStatusBar** constructor initializes the **m_pProgressCtrl** to **NULL** so that the progress-status functions can detect that a **CProgressCtrl** has not been created, and create one. The **MCStatusBar** destructor frees the **CProgressCtrl** if it has been created and resets the **m_pProgressCtrl** data member back to **NULL**.

MCStatusBar Dump() and AssertValid()

As we mentioned in the previous chapter, every good MFC extension needs to implement a **Dump()** and **AssertValid()** so that the user can use the standard MFC memory checking and debug output methods on them. **MCStatusBar** is a good citizen and accordingly provides basic **Dump()** and **AssertValid()** implementations:

```
#ifdef _DEBUG
void MCStatusBar::Dump(CDumpContext& dc)
{
    CStatusBar::Dump(dc);

    if (m_pProgressCtrl != NULL){
        dc << "\nMCStatusBar has a progress control--->\n";
        dc << m_pProgressCtrl;
    }
    else
        dc << "\nMCStatusBar has no progress control\n":

}

void MCStatusBar::AssertValid()
{
    CStatusBar::AssertValid();
    if (m_pProgressCtrl != NULL)
        ASSERT_VALID(m_pProgressCtrl);

}
#endif //_DEBUG
```

These implementations are pretty self explanatory. Notice how both **Dump()** and **AssertValid()** are sure to call the overridden base class' function first, to catch errors in that class if they exist, before catching the added functionality in **MCStatusBar**.

Using MCStatusBar

Now that we've implemented **MCStatusBar**, let's think about how the user will use it in a MFC application.

First, the user will have to change the usual **CStatusBar** object in **CMainFrame** from,

```
CStatusBar m_wndStatusBar;
```

to,

```
MCStatusBar m_wndStatusBar
```

It's a good idea to have the user write an **MCStatusBar** accessor like,

```
MCStatusBar * GetStatusBar() {return &m_wndStatusBar;};
```

so that instead of calling,

```
AfxGetMainWnd()->m_wndStatusBar.StartProgress();
```

which wouldn't work, as **m_wndStatusBar** is a protected member of **CMainFrame**, the user can call:

```
AfxGetMainWnd()->GetStatusBar()->StartProgress();
```

You may even want to suggest that the user write wrappers around the progress-status APIs in the **CMainFrame** such as:

```
BOOL StartProgress(int nLower, int nUpper, CString strCaption)
{
    return m_wndStatusBar::StartProgress(nLower,nUpper,strCaption);
};
```

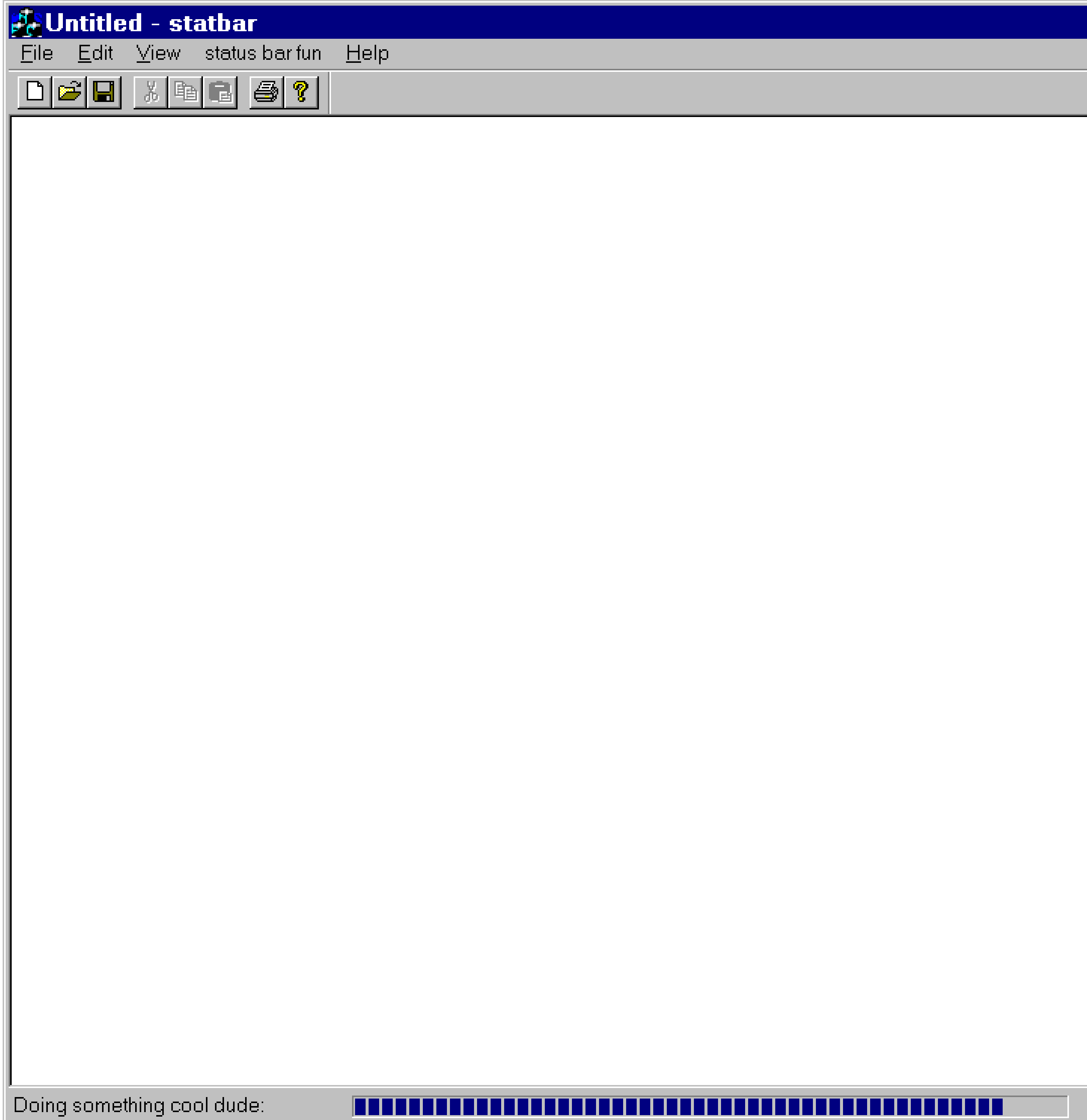
Then the user can just call:

```
AfxGetMainWnd()->StartProgress();
```

Here's some sample code that simulates a lengthy action:

```
m_wndStatusBar.StartProgress(0,10, "Doing something cool dude:");

for (int i = 0; i < 10;i++){
    m_wndStatusBar.SetProgressPos(i);
    for(int j =0;j < 1000000;j++);
}
m_wndStatusBar.StopProgress();
```

The following figure shows **MCStatusBar** in action:



Improving MCStatusBar from Here

Now that we've written our extension and have a sample working with it, let's think about ways to improve `MCStatusBar`. From a class user's standpoint, one of the biggest drawbacks of the class could be the lack of a custom placement.

If you recall, in the **`StartProgress()`** implementation we calculated the position based on a pretty non-scientific algorithm. (Beware whenever you see things like **`+10`**, **`-5`**. Unfortunately, MFC is full of these kinds of hard-coded values.) For vanilla AppWizard- generated applications running on Windows 95, this will probably be good enough, but what if someone has some panes in their status bar that they don't want to be overwritten by the progress bar? What if some future operating system has a skinnier scroll bar control and our progress bar turns out to be too big?

There are several ways to solve the problem. We could just have **`StartProgess()`** take a **`CRect&`** argument and make the user always pass in the desired size. Unfortunately, only power users are likely to make this calculation correctly, so it would be
nice to provide at least a default implementation to give the user a hand with the sizing.

This is a job for a virtual function! (Remember, virtuals rule!). What if we add a virtual function to **`MCStatusBar`** like:

```
virtual CRect GetProgressRect() const; // Remember to be const correct!!
```

Then the implementation of **`StartProgress()`** and **`GetProgressRect()`** becomes:

```
BOOL MCStatusBar::StartProgress(int nLower, int nUpper,
    CString strCaption)
{
    //Hey the user needs a progress bar - create him!
    if (m_pProgressCtrl == NULL)
        m_pProgressCtrl = new CProgressCtrl;


    ASSERT(m_pProgressCtrl != NULL);

    SetWindowText(strCaption);
    UpdateWindow(); //force update...

    CDC* pDC = GetDC();
    CSize sizeCaption = pDC->GetTextExtent(strCaption);

    CRect rectSB = GetProgressRect();
    // Create the control
    BOOL bResult = m_pProgressCtrl->Create(0, rectProgress, this, 1);
    ASSERT(bResult == TRUE);

    m_pProgressCtrl->SetRange(nLower, nUpper);
    m_pProgressCtrl->ShowWindow(SW_SHOW);

    return bResult;
}

//Override GetProgressRect() to customize the progress bar
//placement. This default implementation works well on Win95
//with the normal MFC AppWizard status bar configuration…

CRect MCStatusBar::GetProgressRect() const
{
    CRect rectSB;
    GetWindowRect(&rectSB);

    CRect rectProgress;
    rectProgress.left = sizeCaption.cx + 10;
    rectProgress.right = rectProgress.left +
        rectSB.Width()/2; //about 1/2 width
    rectProgress.top = 5;
    rectProgress.bottom = rectProgress.top + rectSB.Height() - 5;
    return rectProgress;
```

```
}
```

If the user needs to customize the size of the progress indicator in the status bar, they just have to create a quick **`MCStatusBar`** derivative, override **`GetProgressRect()`** and they're done. Before this change, they would have had to manually change **`StartProgress()`** and potentially break some of the other logic in there.

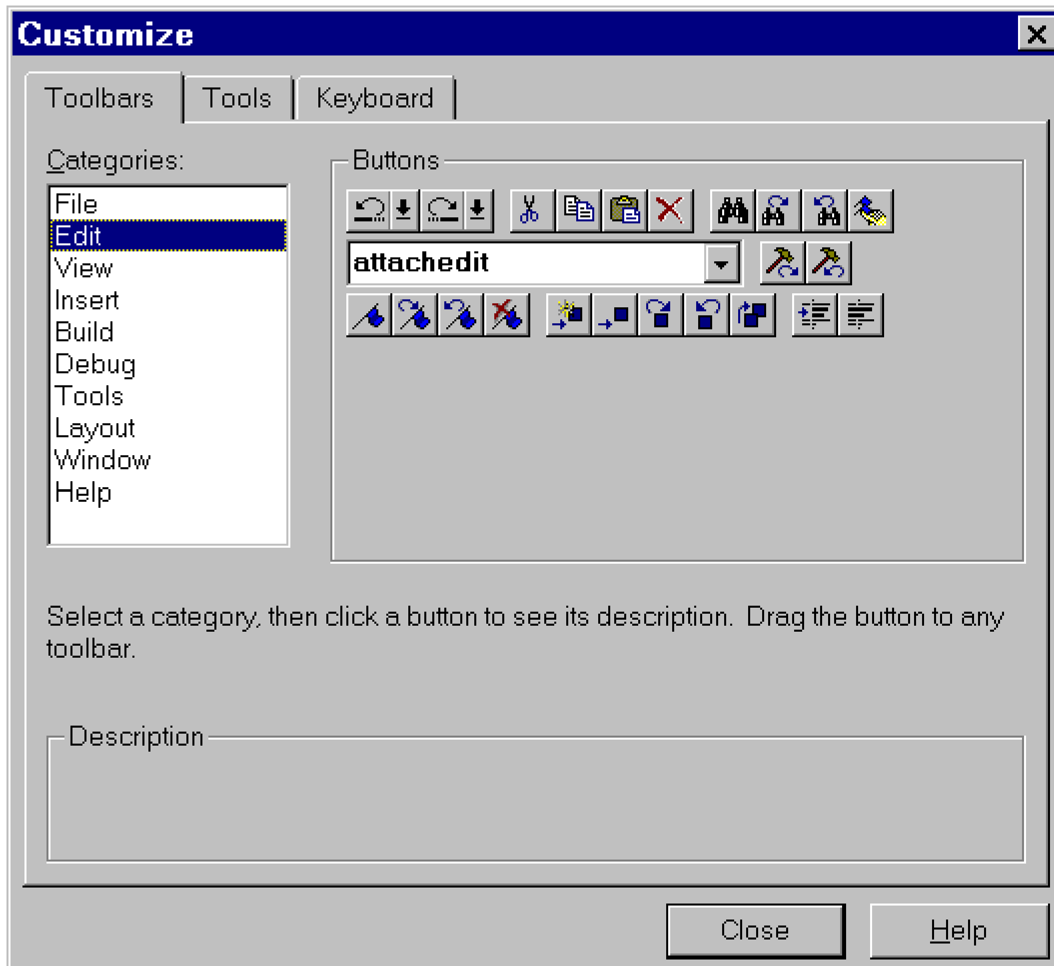
This example also illustrates another good MFC class writing habit: break up long functions with embedded calculations and objects (e.g. a **`CFileDialog`**) into smaller virtual member functions. This practice makes your classes more customizable and also makes them easier to understand for a novice.

Can you think of any other ways to enhance **`MCStatusBar`**? It might be interesting to try and make **`MCStatusBar`** work in it's own thread and accept events as a signal to step to the progress indicator (there could be other critical section issues if you want threads to share a **`MCStatusBar`**). This is a pretty specialized implementation of the class, so we'll leave it up to you to implement the feature for a fun homework assignment.

Now that you've seen how to add a progress indicator to your status bar, let's tackle a more complicated problem: implementing customizable toolbars.

A Customizable Toolbar: MCCustToolBar

Most of Microsoft's applications include a facility that lets the end user customize toolbars. For example, in Visual C++ 4.x, select Tools/Customize.../Toolbars and you'll see the toolbar customization dialog below:



You'll also find similar toolbar customization dialogs in Word, Excel and other Office products. Wouldn't it be nice to be able to add this kind of functionality to your application so that your users could customize toolbars to suit their needs?

MFC provides a pretty handy toolbar that is based on the Win32 common control toolbar. The Win32 common control toolbar supports customization, but, unfortunately, the MFC toolbar class doesn't, so we're going to have to write an MFC extension to solve this. But before we do, let's answer an important question:

How Do They Do That?

Now, the obvious question is "How do Microsoft implement these dialogs and how can we do the same thing in our applications?" There are two possibilities:

Microsoft uses the Win32 common control toolbar which supports customization.
Microsoft has written custom code to do it.

How can you figure out what Microsoft is doing? Fire up Spy++ and take a look at the class name for the Visual C++ toolbar. It turns out to be something like Afx:400000:b:14ee:10:0. Now do the same thing with a normal MFC application and you'll see that the class name for the toolbar is ToolBarWindow32. This tells us that Microsoft isn't using the MFC toolbar class in Visual C++, but has instead written their own enhanced toolbar that supports customization.

Now we're faced with a dilemma: do we follow Microsoft's path and write our own toolbar, or do we extend the MFC toolbar to use the Win32 common control customization? In this chapter, we'll take the easy way out of the problem and use the functionality found in the Win32 common control. At the end of the chapter, we'll look at what it would take to provide a 100% Visual C++ like customizable toolbar and you'll see why we chose to leverage the Win32 common control functionality instead of writing a custom toolbar.

Adding Customizability to MFC's CToolBar

Microsoft's **`CToolBar`** class is actually a thin implementation (that provides a MFC 3.x and lower interface) wrapper around the Win32 toolbar common control. MFC provides an even thinner wrapper around the Win32 toolbar, called **`CToolBarCtrl`**. You can access the underlying **`CToolBarCtrl`** control by calling **`CToolBar::GetToolBarCtrl()`** to get a reference to a **`CToolBarCtrl`**.

One solution to the problem would be to switch your MFC applications to use **`CToolBarCtrl`** instead of **`CToolBar`**. The problem with this solution is that most MFC applications use **`CToolBar`** *plus* **`CToolBar`** provides lots of convenience that you would have to rewrite in a **`CToolBarCtrl`** extension.

So, to add customization to toolbars, we'll take the approach of extending **`CToolBar`**, using some of the untapped **`CToolBarCtrl`** APIs in our implementation.

CToolBar Customization

Before we start thinking about how to extend **`CToolBar`**, we need to first understand how the Win32 common control customization works. What follows are the highlights that you'll need to know to understandthe **`CToolBar`** customization. If you have any questions, detailed information is contained in the Visual C++ online help.

`CToolBarCtrl` lets a user move a tool by dragging it while they hold down the *Shift* or *Alt* key. A customization dialog (that also supports drag-and-drop) is displayed via the **`CToolBarCtrl::Customize()`** API. To enable the **`CToolBarCtrl`** customization functionality, you have to create the control with the **`CCS_ADJUSTABLE`** style bit set. The **`TBSTYLE_ALTDRAG`** bit flag can be used to specify *Alt* key dragging versus *Shift* key dragging.

Once the user starts customizing the **`CToolBarCtrl`**, the control sends a variety of notifications via **`WM_NOTIFY`** messages to the parent window. The **`wParam`** of the **`WM_NOTIFY`** contains different notification codes and the **`lParam`** contains either a **`TBNOTIFY`** or a **`NMHDR`** structure pointer. **`NMHDR`** is a generic notification structure and **`TBNOTIFY`** is a toolbar-specific structure that contains a **`NMHDR`** plus toolbar specific notification information (more on this later).

The notifications serve many purposes. In some cases, they let the application know that the user is doing something and gives the app a chance to stop the action. In other cases, the notifications are requesting information from the application, or simply providing some information about a user's action. The notification messages of interest are:

Notification Message	Meaning
TBN_BEGINADJUST	The user is starting a customization. **lParam** is a **NMHDR** pointer.
TBN_BEGINDRAG	The user is starting a drag operation. **lParam** is a **TBNOTIFY** pointer.
TBN_CUSTHELP	The user selected Help in the customize dialog. **lParam** is a **NMHDR** pointer.
TBN_ENDADJUST	The user has stopped customizing the toolbar. **lParam** is a **NMHDR** pointer.
TBN_ENDDRAG	The user has stopped a drag (dropped). **lParam** is a **TBNOTIFY** pointer.
TBN_GETBUTTONINFO	Retrieves information about the toolbar from the application. **lParam** is a **TBNOTIFY** pointer. This message is usually sent to query the parent for the items to be displayed in the customize dialog.
TBN_QUERYDELETE	Asks the application if a button can be deleted. The application can return **TRUE** to allow the button to be deleted or **FALSE** to stop a button from being deleted. **lParam** is a **TBNOTIFY** pointer.
TBN_QUERYINSERT	Asks the application if a button can be inserted in a certain position. The application can return **TRUE**/**FALSE** to accept/deny an insertion. **lParam** is a **TBNOTIFY** pointer.
TBN_RESET	The user has reset the toolbar. **lParam** is a **NMHDR** pointer.
TBN_TOOLBARCHANGE	Notification that the toolbar has changed. **lParam** is a **NMHDR** pointer.

The **TBNOTIFY** structure is defined as:

```
typedef struct {
    NMHDR hdr;
    int iItem;
    TBBUTTON tbButton;
    int cchText;
    LPTSTR pszText;
} TBNOTIFY, FAR* LPTBNOTIFY;
```

Where **NMHDR** is a normal notification header, **iItem** specifies the button being customized, **tbButton** is a **TBBUTTON** structure that contains toolbar button information, **cchText** contains a count of characters in the toolbar button text and **pszText** is a **NULL** terminated string pointer that contains the button text.

The **TBNOTIFY** of each **CToolBarCtrl()** notification is context-sensitive. For example, in a **TBN_BEGINDRAG** operation, the **iItem** field contains the zero-based index of the button being dragged and the rest of **TBNOTIFY** is not used.

As well as **TBNOTIFY**, we'll need to know about **TBBUTTON** to add customization to **CToolBar**. The **TBBUTTON** structure describes a toolbar button and is declared as:

```
typedef struct _TBBUTTON {
    int iBitmap;
    int idCommand;
```

```
    BYTE fsState;
    BYTE fsStyle;
    DWORD dwData;
    int iString;
} TBBUTTON,* LPTBBUTTON;
```

Member	Meaning
`iBitmap`	Zero-based index of the button bitmap.
`IdCommand`	The command ID for the button.
`fsState`	Holds the button state flags.
`FsStyle`	Contains the button style flags.
`dwData`	Holds application data.
`iString`	Contains the index for the button string.

MCCustToolBar Implementation

Now we know enough about the underlying toolbar customization functionality to start implementing our **`MCCustToolBar`** class, which is derived from **`CToolBar`**. We're going to take a different approach with this class than we did with **`MCStatusBar`**. Instead of presenting the class and explaining how it works, we'll implement the features of the class and then show how it fits together in the end. (You can peak ahead if the suspense is too much for you.) The implementation of **`MCCustToolBar`** breaks down into these steps:

1. Initialization; making sure the toolbar is created in customize mode.
2. Message handling; implementing message handlers for the customization notifications and helpers used in implementing them.
3. Toolbar persistence; implementing serialization for the toolbar so that customization is not lost at application exit.

Each of these steps is covered in a section below.

MCCustToolBar Initialization

The first task in writing **`MCCustToolbar`** is making sure that the underlying **`CToolBarCtrl`** is created with the style bits necessary to turn on the customization feature. Whenever you need to customize window creation, you should override **`PreCreateWindow()`**. Here's the **`PreCreateWindow()`** for **`MCCustToolbar`**:

```
BOOL MCCustToolBar::PreCreateWindow(CREATESTRUCT& cs)
{
    cs.style |= CCS_ADJUSTABLE | TBSTYLE_ALTDRAG;
    return CToolBar::PreCreateWindow(cs);
}
```

All we do in **`PreCreateWindow()`** is add **`CCS_ADJUSTABLE`** and **`TBSTYLE_ALTDRAG`** to the **`CREATESTRUCT`** style bits and then call the overridden **`CToolBar::PreCreateWindow()`**.

As well as overriding **`PreCreateWindow()`**,we also need to override both **`CToolBar::LoadToolBar()`** functions:

```
BOOL MCCustToolBar::LoadToolBar(UINT nIDResource)
{
    BOOL bResult = CToolBar::LoadToolBar(nIDResource);
    GetButtons(m_nSavedCount, m_pSavedButtons);
    return bResult;
}

BOOL MCCustToolBar::LoadToolBar(LPCTSTR lpszResource)
{
    BOOL bResult = CToolBar::LoadToolBar(lpszResource);
    GetButtons(m_nSavedCount, m_pSavedButtons);
    return bResult;
}
```

The reason for overriding **LoadToolBar()** is to store the initial precustomized state of the toolbar in the **m_nSavedCount** and **m_pSavedButtons** data members. The **GetButtons()** helper is implemented as:

```
void MCCustToolBar::GetButtons(int& nSavedCount, TBBUTTON*& pSavedButtons)
{
    // delete previous state
    delete[] pSavedButtons;
    pSavedButtons = NULL;
    nSavedCount = 0;

    // capture current state
    int nButtonCount = GetToolBarCtrl().GetButtonCount();
    pSavedButtons = new TBBUTTON[nButtonCount];
    for (int i = 0; i < nButtonCount; i++)
        GetToolBarCtrl().GetButton(i, &pSavedButtons[i]);
    nSavedCount = nButtonCount;
}
```

GetButtons() takes two reference arguments: **nSavedCount** is a count of the number of buttons and **pSavedButtons** is a pointer to an array of **TBBUTTON** structures (see earlier **TBBUTTON** discussion).

First, **GetButtons()** clears out the arguments and then initializes them, based on the number of buttons in the **CToolBarCtrl**. After initializing the **TBBUTTON** array and saved count, **GetButtons()** iterates through the **CToolBarCtrl** buttons and saves the button state. You'll see why we need to do this at initialization later in the chapter.

Part of the initialization is making sure the customization dialog is created when the user right clicks on the **CToolBar**. A simple **OnRButtonDown()** message map entry and message handler that calls **CToolBarCtrl::Customize()** will take care of that (you should be able to handle the message map part by now!):

```
void MCCustToolBar::OnRButtonDown(UINT nFlags, CPoint point)
{
    CToolBar::OnRButtonDown(nFlags, point);

    // display the toolbar customization dialog
    GetToolBarCtrl().Customize();
}
```

Once the toolbar has been created, it's initial state is stored, and the user can bring up the customize dialog with the right mouse button. All we need to do is handle the customization messages and implement persistence. Remember that after initialization, **m_nSavedCount** and **m_pSavedButtons** contain the initial toolbar button state.

Message Handling

As we've said already, when the user begins customization, it's the application's responsibility to respond to certain **WM_NOTIFY** messages. **WM_NOTIFY** messages are sent to a window's parent (in this case the mainframe) instead of to the toolbar that generates them. This means that we'll have to make every **MCCustToolBar** user add message handlers to their application's mainframe.

Luckily, the folks at Microsoft realized this was a problem and fixed it by adding something called **message reflection** in MFC 4.0 and greater. Message reflection is implemented through special message map entries that tell MFC to route a message to the control which generated the message instead of to its parent window. Using message reflection, we can write a self-contained **MCCustToolBar** and not have to worry about the class user handling the notifications.

Here are the message map entries for the toolbar reflected notifications we need to handle:

```
IMPLEMENT_DYNAMIC(MCCustToolBar, CToolBar)

BEGIN_MESSAGE_MAP(MCCustToolBar, CToolBar)
    //{{AFX_MSG_MAP(MCCustToolBar)
    ON_WM_RBUTTONDOWN()
    ON_NOTIFY_REFLECT(TBN_BEGINADJUST, OnBeginAdjust)
    ON_NOTIFY_REFLECT(TBN_BEGINDRAG, OnBeginDrag)
    ON_NOTIFY_REFLECT(TBN_CUSTHELP, OnCustHelp)
    ON_NOTIFY_REFLECT(TBN_ENDADJUST, OnEndAdjust)
    ON_NOTIFY_REFLECT(TBN_ENDDRAG, OnEndDrag)
    ON_NOTIFY_REFLECT(TBN_GETBUTTONINFO, OnGetButtonInfo)
    ON_NOTIFY_REFLECT(TTN_SHOW, OnToolTipShow)
    ON_NOTIFY_REFLECT(TBN_QUERYDELETE, OnQueryDelete)
    ON_NOTIFY_REFLECT(TBN_QUERYINSERT, OnQueryInsert)
    ON_NOTIFY_REFLECT(TBN_RESET, OnReset)
    ON_NOTIFY_REFLECT(TBN_TOOLBARCHANGE, OnToolBarChange)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

Since **MCCustToolBar** doesn't customize the default drag-and-drop or insertion/deletion operations, some of the notification handler implementations are trivial. Here are the implementations for **OnBeginDrag()**, **OnEndDrag()**, **OnQueryDelete()** and **OnQueryInsert()**:

```
void MCCustToolBar::OnBeginDrag(NMHDR* pNMHDR, LRESULT* pResult)
{
    *pResult = FALSE;
}

void MCCustToolBar::OnEndDrag(NMHDR* pNMHDR,    LRESULT* pResult)
{
    *pResult = FALSE;
}

void MCCustToolBar::OnQueryDelete(NMHDR* pNMHDR, LRESULT* pResult)
{
    // allow any button to be deleted
    *pResult = TRUE;
}

void MCCustToolBar::OnQueryInsert(NMHDR* pNMHDR, LRESULT* pResult)
{
    // allow button insertion anywhere
    *pResult = TRUE;
}
```

Another easy notification handler to implement is the **OnCustHelp()** which is called

whenever the user presses the Help button in the customize dialog. Here's an implementation that displays a fun dialog and a trace statement:

```
void MCCustToolBar::OnCustHelp(NMHDR* pNMHDR, LRESULT* pResult)
{
    // Sample displays a message box but a valid help topic
    // can be displayed for the customize dialog
    AfxMessageBox(_T("This space for rent!"));
    TRACE("{ Help ID = %d }\n", pNMHDR->idFrom);
}
```

Now let's look at two of the more sophisticated handlers: **OnBeginAdjust()** and **OnEndAdjust()**:

```
void MCCustToolBar::OnBeginAdjust(NMHDR* pNMHDR, LRESULT* pResult)
{
    // the customize dialog box is about to be displayed
    // save toolbar state before customization using the dialog

    GetButtons(m_nResetCount, m_pResetButtons);
}
```

OnBeginAdjust() saves the current state so that if the user presses Reset, we can make sure that the toolbar is restored. In the first invocation of the customize dialog, **m_pResetButtons** and **m_nResetCount** will be the same as the saved state, but having two variables allows us to maintain two states:

1. The original application invocation state (actually toolbar creation) stored in **m_nSavedCount** and **m_pSavedButtons**.
2. The latest customized state stored in **m_pResetButtons** and **m_nResetCount**.

The implementation of **EndAdjust()** clears out the reset buffer since the user has left the customize dialog and any reset operations have been performed.

```
void MCCustToolBar::OnEndAdjust(NMHDR* pNMHDR, LRESULT* pResult)
{
    delete[] m_pResetButtons;
    m_pResetButtons = NULL;
    m_nResetCount = 0;
}
```

Since we're on the subject of supporting the reset operation, here's the **OnReset()** handler. We've added a special feature that resets the toolbar back to the original state (if there have been multiple state changes) when the user presses the *Ctrl* key:

```
void MCCustToolBar::OnReset(NMHDR* pNMHDR, LRESULT* pResult)
{
    // restore button state to previous state
    if (::GetKeyState(VK_CONTROL) < 0) // CTRL-key
        SetButtons(m_nSavedCount, m_pSavedButtons);
    else
        SetButtons(m_nResetCount, m_pResetButtons);

    // let OnToolBarChange handle the layout changes
    OnToolBarChange(pNMHDR, pResult);
    *pResult = TRUE;
}
```

This function calls the **SetButtons()** helper with either the **m_nSavedCount**/**m_pSavedButtons** state or the **m_nResetCount**/**m_pResetButtons** state depending on if the control key is pressed to restore the stored **TBBUTTON** to the

CToolBarCtrl. After calling **SetButtons()**, **OnReset()** calls **OnToolBarChange()** to update the toolbar and returns **TRUE**, indicating a successful reset, through the **pResult** pointer.

Before we look at more notification handlers, let's look at the implementation of **SetButtons()**:

```
void MCCustToolBar::SetButtons(int nSavedCount, TBBUTTON* pSavedButtons)
{
    // remove all buttons
    int nButtonCount = GetToolBarCtrl().GetButtonCount();
    while (nButtonCount--)
        GetToolBarCtrl().DeleteButton(0);

    // add new state of buttons
    GetToolBarCtrl().AddButtons(nSavedCount, pSavedButtons);
}
```

First, **SetButtons()** deletes every button in the **CToolBarCtrl**, and then it adds the new buttons via the **AddButtons()** API.

One of the most complex notification handlers is **OnGetButtonInfo()**. This notification is sent by the customize dialog to get information about the buttons the user is adding, deleting and rearranging in the dialog. The ID of the button is specified in a **TBNOTIFY::iItem** field and the return **TBBUTTON** information needs to be returned in the **TBNOTIFY::tbButton** field. **OnGetButtonInfo()** also needs to copy a string that describes the button into the **TBNOTIFY::pszText** buffer which is **TBNOTIFY::cchText** in length.

Here's the implementation of **OnGetButtonInfo()**:

```
void MCCustToolBar::OnGetButtonInfo(NMHDR* pNMHDR, LRESULT* pResult)
{
    TBNOTIFY *pNotify = (TBNOTIFY*)pNMHDR;

    if (pNotify->iItem >= m_nSavedCount)
    {
        *pResult = FALSE;
        return;
    }
    // set the button info
    pNotify->tbButton = m_pSavedButtons[pNotify->iItem];

    // get the string associated with the command ID
    CString buffer;
    buffer.LoadString(pNotify->tbButton.idCommand);

    // use custom text if available, otherwise use tooltip
    CString descript;
    AfxExtractSubString(descript, buffer, 2);
    if (descript.IsEmpty())
        AfxExtractSubString(descript, buffer, 1);

    lstrcpyn(pNotify->pszText, descript, pNotify->cchText);

    *pResult = TRUE;
}
```

If the button requested in **pNotify->iItem** is larger than the initial button range, a **FALSE** indicating error is returned. If that's not the case, **OnGetButtonInfo()** retrieves the button state from **m_pSavedButtons**. Next, **OnGetButtonInfo()** gets a description string from the string table entry for the button's command ID by calling **LoadString()**.

OnGetButtonInfo() then calls the undocumented **AfxExtractSubString()** routine to grab the description from a string that is formatted with **\n** as separators. If there isn't a string in the second section

of the string where it belongs, **OnGetButtonInfo()** tries gets the first string. If you're curious, you can look at how **AfxExtractSubString()** is implemented in the MFC source file: **Msdev\Mfc\Src\Winstr.cpp**.

Finally, **OnGetButtonInfo()** copies the button description string into **pNotify->pszText** and returns **TRUE** (Success!) in the **pResult** pointer.

The only notification function we haven't covered so far is the **OnToolBarChange()** notification that is called by both the customization logic and our **OnReset()** handler.

This function is called to let the toolbar know that the user has completed customizations and the toolbar should update itself accordingly.

```
void MCCustToolBar::OnToolBarChange(NMHDR* /*pNMHDR*/,
    LRESULT* /*pResult*/)
{
    // make the frame recalculate the size of the toolbar
    GetParentFrame()->RecalcLayout();

    if (!m_strProfileName.IsEmpty())
        SaveState(m_strProfileName);
}
```

OnToolBarChange() calls the parent frame's **RecalcLayout()** routine to make sure the toolbar still fits in the frame correctly and then calls **SaveState()** if a profile name is stored in **m_strProfileName** (we'll come to this when we cover **LoadState()**). This brings us to an excellent segue for the next section:

MCCustToolbar Persistence

Customizable toolbars that don't maintain their customizations between application invocations aren't very useful, so we need to add a mechanism to store and restore the state of the toolbars. The normal MFC persistence mechanism is serialization (which we need to support in **MCCustToolbar**, of course) which is great for storing information in a data file. We need to store the toolbar information in the application's profile or registry entry, so let's implement that first and deal with serialization support later. We could do something fancy like serialize to a memory file and then write the binary data to the profile/registry, but for this example, we'll take a more brute force approach.

As we pointed out earlier, **MCCustToolBar** automatically stores its state in every **OnToolBarChange()** notification if desired by calling **SaveState()**. Here's the implementation of **SaveState()**:

```
void MCCustToolBar::SaveState(LPCTSTR lpszProfileName)
{
    // get state of the toolbar buttons (#buttons, button-content)
    BYTE* pState;
    UINT nSize = BuildStateBinary(pState);

    // store it in the registry or .INI file
    CString str;
    str.Format(_T("ToolBar State(0x%04X)"), (UINT)(WORD)GetDlgCtrlID());
    AfxGetApp()->WriteProfileBinary(lpszProfileName, str, pState, nSize);

    // free up the state
    delete[] pState;
}
```

SaveState() calls **BuildStateBinary()** to create an array of **BYTE**s that describe the toolbar state and stores the state in **pState**. Next, **SaveState()** writes the binary state information in the

lpszProfileName entry under the ToolBar State(0x<control ID>) key of the registry or profile depending on what the MFC application is using through the **WriteProfileBinary()** helper routine. Finally, **SaveState()** destroys the state pointer.

> *Note that using the control ID in the registry could lead to the application not being able to find the state information if the ID changes between versions of the app.*

The **BuildStateBinary()** routine is important for understanding how **MCCustToolBar** persistence works. Here's the implementation:

```
UINT MCCustToolBar::BuildStateBinary(BYTE*& pState)
{
    // get state of the toolbar buttons
    // e.g. number of buttons, button-content, etc..
    int nButtonCount = GetToolBarCtrl().GetButtonCount();
    UINT nSize = nButtonCount*sizeof(TBBUTTON)+sizeof(int);

    pState = new BYTE[nSize];

    *(int*)pState = nButtonCount;

    TBBUTTON* pButtons = (TBBUTTON*)(pState + sizeof(int));
    for (int i = 0; i < nButtonCount; i++)
        GetToolBarCtrl().GetButton(i, &pButtons[i]);

    return nSize;
}
```

BuildStateBinary() first allocates a **BYTE** buffer large enough to hold **GetButtonCount() TBBUTTON** structures. Next, **BuildStateBinary()** stuffs the button count into the first four **BYTE**s and then iterates through the buttons placing them into the **BYTE** array by calling **GetButton()**.

The opposite of **SaveState()** is **LoadState()**. **LoadState()** knows how to retrieve the stored state from the profile/registry and resets the toolbar with the loaded state. Here's the implementation of **LoadState()**:

```
void MCCustToolBar::LoadState(LPCTSTR lpszProfileName,
    BOOL bAutoSave /*=FALSE*/)
{
    if (bAutoSave)
        m_strProfileName = lpszProfileName; // save

    // attempt to get the state from registry or profile
    CString str;
    str.Format(_T("ToolBar State(0x%04X)"), (UINT)(WORD)GetDlgCtrlID());
    BYTE* pState;
    UINT nSize;
    if (!AfxGetApp()->GetProfileBinary(lpszProfileName, str, &pState,
        &nSize))
        return;

    // set it as the current state
    ParseStateBinary(pState, nSize);
}
```

First, **LoadState()** saves the profile name in **m_strProfileName** if the **bAutoSave** argument is specified. Next, **LoadState()** formulates the registry/profile key, based on the toolbar's control ID and calls **GetProfileBinary()** to load the information. Finally, **LoadState()** calls **ParseStateBinary()**, passing it a pointer to the binary data from the registry/profile and the size of the data.

ParseStateBinary() is kind of the evil twin of **BuildBinaryState()**. It takes a **BYTE** array and decodes it into a button count and **TBBUTTON** array. Here's the implementation:

```
void MCCustToolBar::ParseStateBinary(BYTE* pState, UINT nSize)
{
    // set it as the current state
    int nButtonCount = *(int*)pState;
    ASSERT(nButtonCount* sizeof(TBBUTTON)+sizeof(int) == nSize);
    TBBUTTON* pButtons = (TBBUTTON*)(pState + sizeof(int));
    SetButtons(nButtonCount, pButtons);

    // free up the state
    delete[] pState;
    OnToolBarChange(NULL, NULL);
}
```

Note that after decoding the **nButtonCount** and a **TBBUTTON** array pointer, **ParseStateBinary()** goes ahead and calls **SetButtons()** and **OnToolBarChange()** to re-initialize the toolbar with the parsed state.

The last function to look at in the **MCCustToolBar** persistence implementation is the **Serialize()** routine. Since we've done most of the leg work for **Serialize()** in the **SaveState()**/**LoadState()** member functions, the serialization implementation is a breeze. Here it is:

```
void MCCustToolBar::Serialize(CArchive& ar)
{
    BYTE* pState = NULL;
    UINT nSize;
    int i;

    if (ar.IsStoring())
    {
        nSize = BuildStateBinary(pState);
        ar << (WORD) nSize;
        for (i=0; i < (int) nSize; i++)
            ar << pState[i];

        // free up the state
        delete[] pState;
    }
    else
    {
         TRY
         {
             // attempt to get the state from disk
             WORD w; ar >> w; nSize = (UINT) w;
             pState = new BYTE[nSize];
             for (i=0; i < (int) nSize; i++)
                 ar >> pState[i];
         }
         CATCH(CArchiveException, e)
         {
             delete pState;
             return; // error
         }
         END_CATCH

         // set it as the current state
         ParseStateBinary(pState, nSize);
    }
}
```

The **Serialize()** implementation is pretty self explanatory. Now that we've seen how to implement most of **MCCustToolBar**, let's look at the declaration for the class based on what we have so far:

MCCustToolBar Declaration/Interface

(Drum role please!) And now for the moment you've been waiting for, the **MCCustToolBar** class declaration:

```
class MCCustToolBar : public CToolBar
{
    DECLARE_DYNAMIC(MCCustToolBar)

// Construction
public:
    MCCustToolBar();
    virtual BOOL LoadToolBar(UINT nIDResource);
    virtual BOOL LoadToolBar(LPCTSTR lpszResource);

// Operations
    // state save and restore
    virtual void SaveState(LPCTSTR lpszProfileName);
    virtual void LoadState(LPCTSTR lpszProfileName,
        BOOL bAutoSave = FALSE);
    virtual void Serialize(CArchive& ar);

// Overrides
    virtual void GetButtons(int& nSavedCount, TBBUTTON*& pSavedButtons);
    virtual void SetButtons(int nSavedCount, TBBUTTON* pSavedButtons);
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);

// Implementation
public:
    virtual ~MCCustToolBar();

protected:
    // profile name for reg/prof entry.
    CString m_strProfileName;

    // starting state save
    int m_nSavedCount;
    TBBUTTON* m_pSavedButtons;

    // reset state save (used during customization)
    int m_nResetCount;
    TBBUTTON* m_pResetButtons;

    //Registry helpers.
    UINT BuildStateBinary(BYTE*& pState);
    void ParseStateBinary(BYTE* pState, UINT nSize);

protected:
    //{{AFX_MSG(MCCustToolBar)
    afx_msg void OnRButtonDown(UINT nFlags, CPoint point);
    afx_msg void OnBeginAdjust(NMHDR* pNMHDR, LRESULT* pResult);
    afx_msg void OnBeginDrag(NMHDR* pNMHDR, LRESULT* pResult);
    afx_msg void OnCustHelp(NMHDR* pNMHDR, LRESULT* pResult);
    afx_msg void OnEndAdjust(NMHDR* pNMHDR, LRESULT* pResult);
    afx_msg void OnEndDrag(NMHDR* pNMHDR, LRESULT* pResult);
    afx_msg void OnGetButtonInfo(NMHDR* pNMHDR, LRESULT* pResult);
    afx_msg void OnQueryDelete(NMHDR* pNMHDR, LRESULT* pResult);
    afx_msg void OnQueryInsert(NMHDR* pNMHDR, LRESULT* pResult);
    afx_msg void OnReset(NMHDR* pNMHDR, LRESULT* pResult);
    afx_msg void OnToolBarChange(NMHDR* pNMHDR, LRESULT* pResult);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

As you read through the declaration, try to think of any ways you can improve the interface and hold that thought for a second while we look at how to use **MCCustToolBar** and then examine some ways to make it better.

Using MCCustToolBar

To use `MCCustToolBar`, the user only has to make 3 or 4 changes to a standard MFC application:

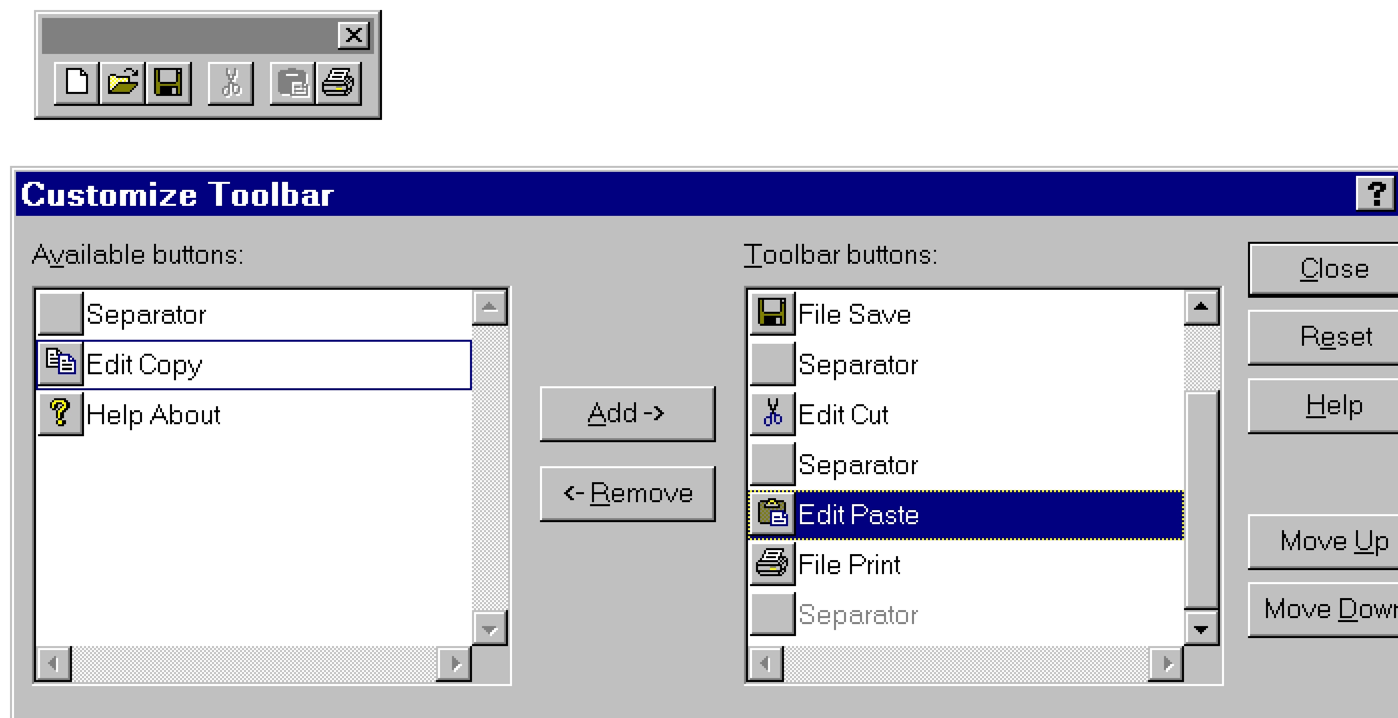
1 Change `CToolBar` to `MCCustToolBar` in `Mainfrm.h`.

2 Add an `#include "mc_tbar.h"` at the top of `Mainfrm.h`.

3 Add a call to `m_wndToolBar.LoadState()` to `CMainFrame::OnCreate()`, passing the name you wish to use for the storage, and `TRUE` if you want to automatically save the state.

4 If you didn't pass `TRUE` in `LoadState()`, add `m_wndToolBar.SaveState()` to a handler for `WM_CLOSE`, passing the same name you used in `LoadState()`.

Here's a figure of `MCCustToolBar` in action:



Extending MCCustToolBar Further...

At this point, we have a good start on a fully customizable toolbar. Here are some ideas for how you can try and improve it:

Add virtual callbacks to hide message reflection. If a user wants to customize the drag-and-drop support, they will have to understand message reflection. Instead, you could create a set of virtual functions that are called from within the `MCCustToolBar` so that the class user only has to override a virtual function instead of having to add message reflection to their `MCCustToolBar` class. For example, you could overload the notification handler with a virtual function, then modify the original handler to call this new function:

```
protected:
```

```
    virtual BOOL OnBeginDrag(NMHDR * pNMHDR) const;

BOOL MCCustToolBar::OnBeginDrag(NMHDR * pNMHDR) const
{
    // Default implementation - do nothing,
    // override this to customize OnBeginDrag().
    return FALSE;
}

void MCCustToolBar::OnBeginDrag(NMHDR* pNMHDR, LRESULT* pResult)
{
    //Give derived classes the opportunity to
    *pResult = OnBeginDrag(pNMHDR);
}
```

As presented, **MCCustToolBar** doesn't support non-button controls in the toolbar. Most advanced applications have buttons, combos and other non-button controls in the toolbar. The **MCCustToolBar** implementation on your CD has been enhanced to support a variety of different controls.

Adding controls to the toolbar is also covered later in the section on enhancing Windows UI components.

For the MFC Developer Looking for a REAL Challenge..

Unfortunately, the Win32 common control customization isn't nearly as nice as the customization found in Visual C++ and other Microsoft applications. You can't drag-and-drop between toolbars, the Visual C++ toolbar customization dialog is much nicer and easier to use than the Win32 implementation, etc..

If you really want that level of customization, you'll have to:

Write your own toolbar that knows how to accept dragging and dropping of toolbar items in and out of the bar.

Re-implement all of the normal MFC toolbar logic like docking, bitmap loading, disabling, etc..

Write your own dialog that enumerates the toolbars, tools and allows the user to drag and drop them. The user can start a new toolbar by dragging a tool to the desktop.

Be sure to save the toolbar configurations out to the registry so that user customizations are maintained.

Where Do I Go from Here?

Now that we've presented you with two fairly different MFC extensions, you should be ready to venture forth on your own into the exciting world of MFC extension writing. Here are some of the key points to take with you from this chapter:

MFC class design is iterative. Start out simple and enhance your class interfaces over time by adding virtual functions and other conveniences that you missed in the first pass.

Be sure to follow Microsoft's conventions. They make it much easier to communicate your class ideas to other MFC developers. Think of it as the Lingua Franca of Windows programmers.

Try using **MCStatusBar** and **MCCustToolBar** in your own projects and think of new and exciting enhancements for them.

Have fun and happy MFC programming!

Special thanks goes to Harlan Seymour with HyperCube (`http://www.hcube.com`*) an MFC extension company. Harlan already had a good start on a customizable toolbar that he let me use for the basis of* `MCToolBar` *in this chapter. There's a demo of HyperCube's HyperView++ product on the book's CD-ROM for your enjoyment.*

Porting from 16-bit to 32-bit

Judging from the abundance of talk concerning 32-bit development, it's inevitable that, some day, you'll have to port your existing 16-bit code to 32-bit and carry on from there.

What you do with the code once it's in a 32-bit format is your business, but getting the code from a 16-bit to a 32-bit format is my business, which is why I wrote this chapter.

We'll begin with the basics of porting your code. There are actually two avenues that you can take. The first is porting your code from 16-bit C code to 32-bit C code. The second involves moving your code to the 32-bit version of MFC. There are tools for each of these avenues to assist you in your porting efforts.

If you work for a corporate environment or release software for the retail market, you'll notice that management has had to do some serious thinking about whether or not to move to a 32-bit operating system. The most common question is, "What benefits will 32-bit development provide?". Well, let's step back for a second and consider the limitations of 16-bit development first, then I'll explain how things are different (and better) with 32-bit development.

On another note, some programmers might find that, to keep up with the growing market, they must port their applications to a different Windows platform. Things change fast in this market, so, if you don't keep up with the changes, you may lose your market value. (If you don't believe me, go ask all the COBOL programmers who are out of work.)

An Overview

I have written applications under Windows 3.x that have dealt with graphics, data management, file I/O… and the list goes on. When you're dealing with graphics, especially bitmaps that you're loading from a file, you have to allocate sufficient memory for the bitmap's bits and parts (the header information, the color table, etc.). Once you reach the 64K boundary, you have to use *huge* pointers (which are costly and slow) or manage segmentation yourself (good luck to you if you do).

Managing memory under Win32 gives you less to worry about. There are no such things as memory models or far pointers. There's actually no need whatsoever to deal with segments and boundaries. Win32 functions treat memory as linear flat-memory and, since all the operating systems that will run Win32 support 32-bit registers, we're allowed to address a much greater amount of memory (four gigabytes to be exact, although about two gigabytes out of the four are used by the operating system).

Moving to a different platform involves treating data differently, since what might be 16 bits under one platform could be 32 bits in another. For example, integers and unsigned integers are 16 bits under Windows 3.x, but 32 bits under Win32. This means that you can pass a number like one billion to an integer under Win32, but, if you passed this same number to an integer under 16-bit Windows, you would introduce bugs into your application.

There are also other data types that were introduced for Windows development. For example, **`HANDLE`**, **`UINT`**, **`BOOL`**, and **`WORD`**. These types are 16 bits under Windows 3.x but 32 bits under Win32. The reason much of this has changed from 16 to 32 bits is that you need 32 bits to address the memory that is now available to your application. It also turns out that your code will be quicker. Instead of imposing something different, it's much faster to treat values as their native size. Furthermore, the operating system is able to deal with data alignment much more efficiently. As a result, 32-bit applications end up performing much better that their 16-bit counterparts, not only because the processor can move greater amounts of memory faster, but also because the operating system has to do a lot less to massage the data.

When you're developing your own structures to hold data for your application, start thinking in 32 bits. In other words, try to make structures align with four-byte boundaries (or eight-byte boundaries). This will provide better

performance, because the operating system will attempt to move the chunks of data as four byte blocks anyway. Of course, your programs will need a bit more memory, but you have got a lot more available to you with flat memory addressing, so this shouldn't be a problem.

Basic Differences

The first difference between the two platforms (16-bit Windows versus 32-bit Windows) that you should be aware of is that the calling conventions have changed from **`__pascal`** to **`__stdcall`**. It still has the same efficiency, but with greater power. Most functions are declared with **`APIENTRY`** instead of **`PASCAL`**. **`APIENTRY`** breaks itself down to **`__stdcall`**.

When **`WinMain()`** is called, the parameter names have remained the same but they have grown. The variable **`hPrevInstance`** is always sent as **`NULL`**, so you'll have to find some other means of detecting whether an instance of an application is already running (such as find a window by calling the **`FindWindow()`** function). The other parameters still contain the same type of information.

The **`hInstance`** parameter has also changed slightly. In Windows 3.x, you could use **`hInstance`** to retrieve resources. To do this in Win32, you need to pass an **`hModule`**. This is easy because Win32 passes the **`hModule`** as the **`hInstance`** to **`WinMain()`** when it first loads the application. In other words, you can continue to use the **`hInstance`** as you did under 16-bit Windows. You can also call **`GetModuleHandle()`** later in your application if you don't save the handle to a global variable in **`WinMain()`**.

Speaking of **`WinMain()`**, which is where most programmers place their message loops, things have also changed slightly here. Because Windows 95 and NT now offer multitasking, you can create multiple threads with separate message queues (one per thread).

This introduces a couple of points. Since the model is now multithreaded, different applications can process messages at the same time (conceptually). The ramifications are that it's harder to manage the input model, because more than one application might receive messages at the same time.

If you have written code to take advantage of messages and expect those messages to be sent out in a particular order, forget it. You'll have to rewrite those pieces of code before you can port your application safely to a Win32 platform.

The bottom line is that when you begin to port your application to Win32, make sure you study how the application receives input. It doesn't matter whether the input is mouse, keyboard, or the system, you still need to look at the process and make sure that it will work properly under Win32.

Message Handling

Window procedures and the messages they receive have changed considerably from their 16-bit counterparts. Instead of using **`WORD`**s for the message ID and **`wParam`**, you use **`UINT`** and **`WPARAM`** respectively. The **`lParam`** has changed from a **`long`** to an **`LPARAM`**.

The information that **`wParam`** and **`lParam`** contain has also changed from their 16-bit counterparts. Certain messages in 16-bit Windows send a handle packaged with something else (such as flags) in the **`lParam`**, and an ID in the **`wParam`**. Since handles have grown from 16 bits to 32 bits, they now take up the whole **`lParam`** and the packaging is performed in the **`wParam`**. Here's a list of the messages that have changed:

Message	**`wParam`**	**`lParam`**
`WM_ACTIVATE`	**`LOWORD`** = activation flag **`HIWORD`** = minimized flag	Window handle
`WM_CHARTOITEM`	**`LOWORD`** = key value	List box handle

	HIWORD = caret position	
WM_COMMAND	**LOWORD** = item, control or accelerator ID **HIWORD** = notification code	Control handle
WM_CTLCOLOR	This message has been replaced with a series of messages to handle the individual types of controls. All the messages begin with **WM_CTLCOLOR**xxx, where xxx is **BTN**, **DLG**, **EDIT**, **LISTBOX**, **MSGBOX**, **SCROLLBAR** or **STATIC**.	
WM_MENUSELECT	**LOWORD** = menu item or pop-up menu index **HIWORD** = menu flags	Menu handle
WM_MDIACTIVATE	Child window handle being deactivated	Child window handle being activated
WM_MDISETMENU	Frame menu handle	Window menu handle
WM_MENUCHAR	**LOWORD** = ASCII character **HIWORD** = menu flags	Menu handle
WM_PARENTNOTIFY	**LOWORD** = event flags **HIWORD** = child ID	Window handle of the child or cursor coordinates
WM_VKEYTOITEM	**LOWORD** = virtual-key code **HIWORD** = caret position	List box handle
WM_HSCROLL	**LOWORD** = scroll bar value **HIWORD** = scroll bar position	Scroll bar handle
WM_VSCROLL	**LOWORD** = scroll bar value **HIWORD** = scroll bar position	Scroll bar handle
EM_LINESCROLL	Number of characters to scroll horizontally	Number of characters to scroll vertically
EM_SETSEL	Starting position	Ending position
EM_GETSEL	Starting position	Ending position

Therefore, if you want to support code for both 16-bit Windows and Win32, you'll have to provide a block of code that is compiled for one compiler (e.g. 16-bit) and another block of code that can be compiled with another compiler (e.g., 32-bit). There's a precompiler definition called **WIN32** that does just this. The following code snippet shows exactly how:

```
#ifdef WIN32
   // Code specific to Win32
#else
   // Code specific to Win16
#endif
```

There's also a header file that you can include and use in your source code that contains several macros to do this exact same work for you. These macros are called **message crackers**. They basically wrap themselves around a message and call a function that you must provide with the available variables independently. This obviates the need to use the **#ifdef WIN32** construct shown above when you are cracking messages. The header file is called **Windowsx.h** and contains all of the crackers that you'll need for your application. You must catch the message ID in a **switch** statement as a case, then crack the message while providing a handler for the message and parameters (very similar to MFC message handlers). For example, if you're trying to catch the **WM_MOVE** message, you would first catch the message:

```
switch(uMessage)
{
   case WM_MOVE:
      HANDLE_WM_MOVE(hWnd, wParam, lParam, OnMove);
      break;

   default:
```

```
        // Do something
    }
```

Then you would provide the functionality to handle the **WM_MOVE** message in the function passed to the message cracker:

```
void OnMove(HWND hWnd, int x, int y)
{
   // Do something
}
```

If I wish to pass on the message to another function (such as the default windows procedure), I would simply use the **FORWARD** macro of the message cracker inside my function:

```
void OnMove(HWND hWnd, int x, int y)
{
   // Do something
   FORWARD_WM_MOVE(hWnd, x, y, DefWindowProc);
}
```

This code allows me to write my code independently of the compiler I'm using. Since the appropriate compiler already ships with the respective version of **Windowsx.h**, I don't have too much to worry about. I simply recompile my code in the appropriate compiler and I'm ready to ship.

You'll also have to worry about messages coming from menus. These message are now packaged differently. The data contained in the **wParam** and **lParam** has been moved around to make up for the extra size in the handles. Fortunately, there are some message crackers that we can use to pull the individual data pieces from the two variables passed to the window procedures. For example:

```
WORD wCmd       = GET_WM_COMMAND_CMD(wParam, lParam);
WORD wID        = GET_WM_COMMAND_ID(wParam, lParam);
HWND hWndCtrl   = GET_WM_COMMAND_HWND(wParam, lParam);
```

The last important message that has changed is **WM_CTLCOLOR**. There was just no way to fit all the necessary information into two 32-bit fields, so Microsoft chose to send a message corresponding to the type of control for which the color is being requested. For example, you'll receive **WM_CTLCOLORBTN** for a button, or **WM_CTLCOLOREDIT** for an edit control. For this message, you'll have to provide the **#ifdef WIN32** solution as mentioned above, since the message is handled differently (even with the use of the wrappers).

API Functions

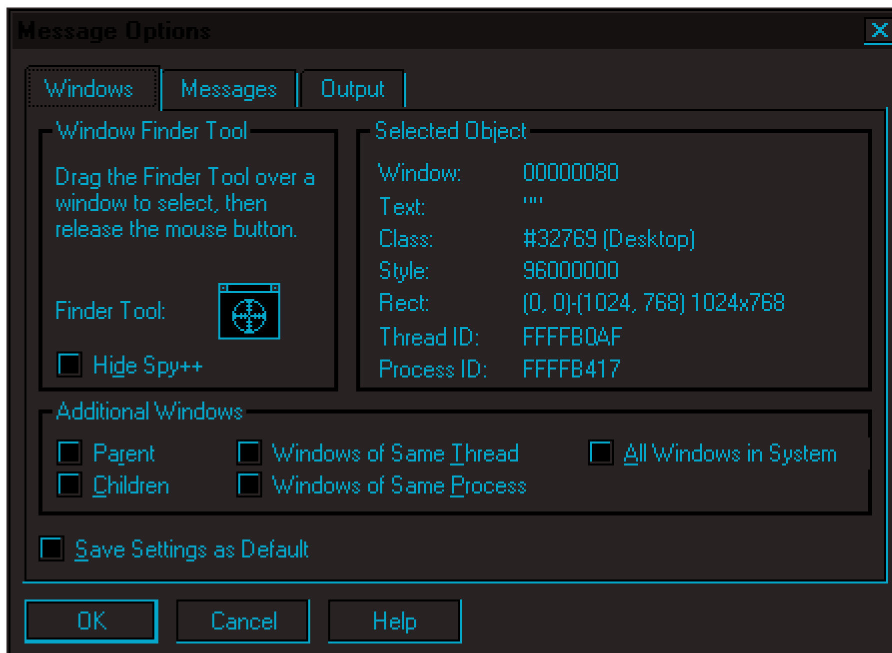
Messages aren't the only thing to change in your Win32 development efforts—API functions have also changed. There are certain functions that carry the same name, but the parameter list or the functionality might have changed. There are also many functions that now have extended capabilities. Extended functions are mostly implemented in the GDI area. Several functions have also been dropped altogether, because they didn't make sense under a 32-bit platform (such as many of the DOS functions).

One function that will cause a lot of headaches is **GetWindowWord()** or **GetClassWord()**. Much of the behavior has now been implemented in **GetWindowLong()** and **GetClassLong()**. It makes sense after all, since many of the data types have been expanded to 32 bits. If you need to have a code base that will work for both 16-bit and 32-bit platforms, you'll need to use the preprocessor definition (**WIN32**) to determine which function should be used. The table below is a list of all the window and class offsets. This list shows the replacements for the 16-bit offsets and what they should be under the 32-bit platforms:

16-bit Index	32-bit Index Replacement
GCW_CURSOR	GCL_CURSOR

GCW_HBRBACKGROUND	**GCL_HBRBACKGROUND**
GCW_HICON	**GCL_HICON**
GWW_HINSTANCE	**GWL_HINSTANCE**
GWW_HWNDPARENT	**GWL_HWNDPARENT**
GWW_ID	**GWL_ID**
GWW_USERDATA	**GWL_USERDATA**

Capturing mouse messages is also different under Win32. If you capture the mouse while the mouse is not down (**WM_xBUTTONDOWN** has not been received), you can only see mouse move messages while the mouse is above the window that captured the mouse or any window owned by the same thread. Once the mouse moves to a window of another thread, you no long receive mouse messages, but you can fix this by capturing the mouse after the user has clicked the mouse button. Keep in mind that the user must continue to hold the mouse button down so that you can receive messages even when the mouse moves over windows of other threads. Therefore, you should release the captured mouse after a **WM_xBUTTONUP** message. For an example of this behavior, take a look at the Spy++ application that ships with Visual C++. In Spy++, when you're searching for a window, you must first click and hold down the mouse on the Finder Tool: icon and then drag the mouse around to the other windows. This dialog box from Spy++ contains the Finder Tool: icon:



As I mentioned above, some functions have been extended. There are different reasons for this. For example, the **CreateWindow()** function has additional functionality in the extended version which is called **CreateWindowEx()**. A set of GDI functions returned a packed 32-bit value containing possibly the *x* and *y* values of the previous position before being moved to the new given position. For example, **MoveTo()** receives the new position and returns a packed 32-bit **long** containing the previous position. This function can no longer be used, since positions are now 32-bits for the *x* and *y* (not 16-bits). The new version, **MoveToEx()** receives an additional parameter, a pointer to a **POINT** structure (which contains 32-bits for both the *x* and *y* members) that will contain the previous position before moving to the new position.

There are several functions which have been extended to provide additional information. Under 16-bit Windows, these functions returned 32-bit values which contained information packed into the double word value (as two 16-bit values). In order to support real-world values (which can potentially be larger then a 16-bit value will support), Win32 supports functions that can now return better (and wider band-width) information. For example, under 16-bit

Windows, the **MoveTo()** function returned a **DWORD** value that contained two 16-bit values (packed into the **DWORD**) for the old *x* and *y* before the function was called. Under Win32, the **MoveTo()** function no longer exists (surprise, surprise), so you have no choice but to use the **MoveToEx()** function. The difference is that the function now returns a **BOOL** (for success or failure) and accepts a new parameter, a pointer to a **POINT** structure of which it will fill with the old *x* and *y* position. Why a **POINT** structure? Well, if you think about it, a **POINT** structure contains two integers (which means two 32-bit values). This is perfect for returning real-world coordinates (or simply wider values). Second, the fewer parameters you have to pass to a function, the better.

The other functions available under this category also have an **Ex** extension and their 16-bit counterparts have been dropped from Win32. Most of these functions are related to window management or graphics output. For example, **OffsetViewportOrg()** has been changed to **OffsetViewportOrgEx()**, and **GetBrushOrg()** has been changed to **GetBrushOrgEx()**. The main thing to remember is that these functions have not been included in the Win32 API, so, if you have used them in your 16-bit applications, you'll have to change them in your port from 16 bits to 32 bits. That's the bad news. The good news is that, if you've been using MFC all along to write your applications, you won't have to change a thing. For example, if you call **CDC::MoveTo()**, the 16-bit version of MFC called the 16-bit version of **MoveTo()** (implemented by Windows 3.1). Under the 32-bit version of MFC, it calls **MoveToEx()** (implemented by Win32).

Here's a list of all of the 16-bit functions that that have been extended in Win32:

16-bit API	Portable Extended API
DlgDirSelect	**DlgDirSelectEx**
DlgDirSelectComboBox	**DlgDirSelectComboBoxEx**
GetAspectRatioFilter	**GetAspectRatioFilterEx**
GetBitmapDimension	**GetBitmapDimensionEx**
GetBrushOrg	**GetBrushOrgEx**
GetCurrentPosition	**GetCurrentPositionEx**
GetTextExtent	**GetTextExtentPoint**
GetTextExtentEx	**GetTextExtentExPoint**
GetViewportExt	**GetViewportExtEx**
GetViewportOrg	**GetViewportOrgEx**
GetWindowExt	**GetWindowExtEx**
GetWindowOrg	**GetWindowOrgEx**
MoveTo	**MoveToEx**
OffsetViewportOrg	**OffsetViewportOrgEx**
OffsetWindowOrg	**OffsetWindowOrgEx**
ScaleViewportExt	**ScaleViewportExtEx**
ScaleWindowExt	**ScaleWindowExtEx**
SetBitmapDimension	**SetBitmapDimensionEx**
SetMetaFileBits	**SetMetaFileBitsEx**
SetViewportExt	**SetViewportExtEx**
SetViewportOrg	**SetViewportOrgEx**
SetWindowExt	**SetWindowExtEx**
SetWindowOrg	**SetWindowOrgEx**

Finally, several functions were dropped from the API; those associated with DOS and the 16-bit architecture. Because we're working on and developing applications for a 32-bit environment, functions like **GlobalDOSAlloc()** or **GlobalDOSFree()** have no place in the 32-bit world. For example, the **AccessResource()** function used DOS file handles and treated files in a way not supported by Win32. The

following table contains a list of functions that you should avoid like the plague in your own Win32 applications (especially since they have not been implemented in Win32):

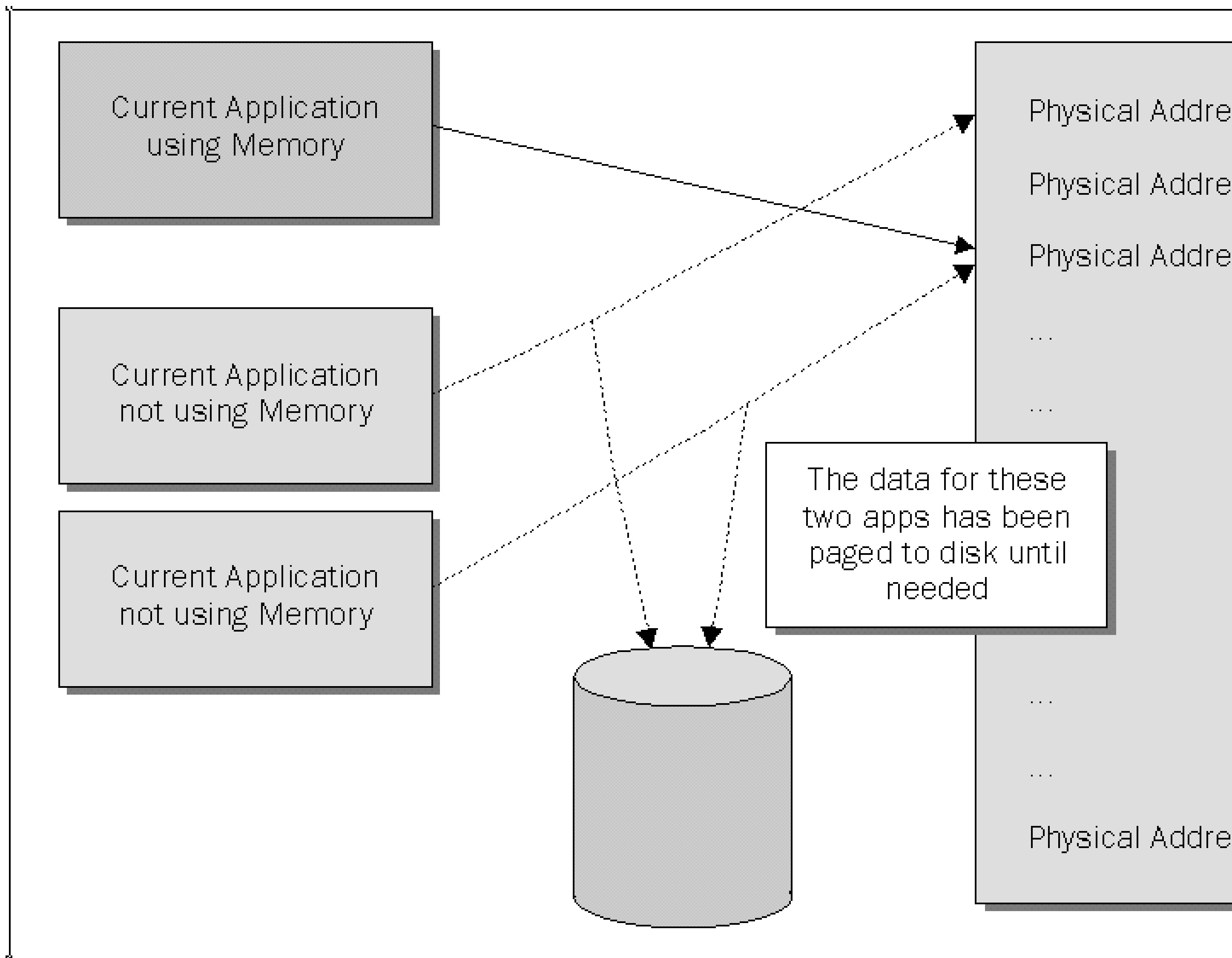
AccessResource	**GetInstanceData**	**SetEnvironment**
AllocDSToCSAlias	**GetKBCodePage**	**SetResourceHandler**
AllocResource	**GetModuleUsage**	**SwitchStackBack**
AllocSelector	**GlobalDOSAlloc**	**SwitchStackTo**
Catch	**GlobalDOSFree**	**Throw**
ChangeSelector	**GlobalNotify**	**UnlockData**
FreeSelector	**GlobalPageLock**	**ValidateCodeSegments**
GetCodeHandle	**IsGdiObject**	**ValidateFreeSpaces**
GetCodeInfo	**IsTask**	**Yield**
GetCurrentPDB	**LockData**	
GetEnvironment	**NetBIOSCall**	

Under Win16 development, you might have called several DOS file functions by calling **`Int21()`**. These functions now have Win32 equivalents. For a list of these functions, see the section on long filenames, in Chapter 9 *Windows Shell Programming*.

Memory Usage

Since applications are now given their own process address space in which to run, memory is no longer sharable by default as it was in Windows 3.x. System memory is still shared amongst the different processes, but is mapped into the appropriate place in the process's address space.

The rest of the applications using the same piece of memory are simply paged out to disk or are moved to a different place in memory (in cases where they still need to access memory in the background). What makes all this possible is the virtual memory manager. The following figure depicts how an application can use the same physical address as another application but still access totally different data. In reality, it doesn't matter if two applications use the same virtual addresses or the same physical addresses, the operating system will properly manage the addresses and data for them. This is why several functions which allowed direct access to the memory pages have been dropped from Win32's vocabulary.



The whole point is that you can't read or write to a piece of memory allocated by another process. Using **`GMEM_DDESHARE`** or **`GMEM_SHARE`** when you're allocating memory to share with different processes doesn't work anymore.

The only way to share memory now is to use memory-mapped files or Dynamic Data Exchange Management Library (DDEML).

In Win32, the concept of global and local memory is gone. Both types of memory allocations are simply mapped to the process's memory space, so you can now go back to using the old **`malloc()`** and **`free()`** C run-time functions. There's no need to lock memory, since the linear addresses are guaranteed to remain the same withinthe process space. However, the underlying physical address might (and most definitely *will*) change to provide for different applications becoming active at different times. Since the memory will have to be paged (which by definition makes the memory virtual), things will often move around in memory, causing the physical memory addresses to contain different things depending on which process is active.

The good news is that, as a programmer, you don't have to worry about these details, since all of this is completely hidden.

Where to Learn More

You can find all of the details for the individual messages and functions I have mentioned in the Win32 SDK documentation (including what's now received via **wParam** and **lParam** for the individual messages).

When I first started working with Win32, I always tried to look up the functions or messages I was about to call or receive in the help files, just to make sure that there weren't any changes I had to be aware of. This not only allowed me to learn a lot about Win32 and its functionality, but it also caused me to drop functions from my vocabulary that had better replacements in Win32.

Porting Tools

If you're moving 16-bit C code to a 32-bit environment such as Windows NT or Windows 95, you'll most likely want to take advantage of any tools you have at your disposal. There are certain rules that you should follow to make sure the port goes smoothly, with very few glitches.

First, I'll discuss a tool called PORTTOOL which ships with the Win32 SDK. You can also find it on the Microsoft Developers Network CD (MSDN) or in the **\msdev\samples\sdk\sdktools** directory on the Visual C++ CD. This tool ports your 16-bit C code to a 32-bit platform. There's also a tool that you can use to port your C code to MFC. I'll discuss these tools in this section.

PORTTOOL

When PORTTOOL is used with an existing 16-bit source file, it flags where there might be problems when you begin to convert the source, but it's not smart enough to make the changes for you. PORTTOOL isn't very sophisticated. As a matter of fact, it's so simplistic that it will flag you on comments that have words which are similar to keywords which are now incompatible with 32-bit development. If you want to add extra features to the tool, Microsoft provides the source code so that you can modify it (but I wouldn't waste my time).

PORTTOOL displays a dialog box whenever it comes across a piece of code that you should be warned about. If you have ever written code that involved splitting up the **lParam** to pull out things like handles and item IDs which were crammed together under Windows 3.x., PORTTOOL will warn you of potential problems when you're splitting up the **lParam** in your code.

PORTTOOL uses settings in the file **Port.ini** to determine what items to look for. This file is based on rules which are summarized on the Visual C++ CD (it can be found in InfoView under Visual C++ Books; C/C++; Programming Techniques; Porting 16-bit Code to 32-bit Windows; Summary of API and Message Differences).

When you start using the PORTTOOL on your files, you'll begin to receive all kinds of messages. It will flag you for things like usage of the **FAR** keyword, alerting you that this keyword is no longer necessary for 32-bit development, since all addresses are now linear (there is no local or global memory anymore).

One thing that you'll notice when you're running the tool is that it doesn't flag the **PASCAL** keyword. If you look through the 32-bit **Windows.h** file, you'll find that this keyword is defined to be **__stdcall**. The same applies for the lower case version, **pascal**.

Despite all the problems with PORTTOOL, it does suggest some helpful hints from time to time. For example, when it finds window procedures declared using a **WORD** for the **wParam**, it will suggest that you use the portable type, **WPARAM**. This means that your code would end up like this:

```
LONG FAR PASCAL _export MainWndProc(HWND hWnd,
UINT wMsgID, WPARAM wParam, LPARAM lParam);
```

Having said that, PORTTOOL isn't very consistent. It flags **`wParam`**, which made me assume that it would flag **`lParam`** (from **`LONG`** to **`LPARAM`**), but it doesn't.

PORTTOOL will also alert you that you need to change the way you process menu commands, conversion of old 16-bit types to new 32-bit types (such as the **`MAKEPOINT`** macro which has now been replaced with the **`LONG2POINT`** macro) and API calling with inappropriate data types or casts. If there are functions or macros that have been replaced, you will need to use the preprocessor if you wish to maintain your code's portability.

If you're simply moving your C code from 16-bit to 32-bit Windows, the PORTTOOL is a good starting point, despite the awful, inconsistent user interface. Also, keep in mind that PORTTOOL looks for issues using a standard naming scheme. For example, it expects a Windows procedure to have the word **`WndProc`** somewhere in its name. This also includes dialog box procedures.

MFC Migration Tool

The previous sections assumed that you have a bed of code which needs to be ported to a 32-bit Windows environment. PORTTOOL works for C code and gives suggestions for making the C code compatible with 32-bit Windows, but it hasn't got a clue about MFC (or C++ for that matter).

After hearing so much about MFC and its benefits, you might want to port your code directly from C to MFC. Keep in mind that once you port your code to MFC, the job of getting the code to a 32-bit platform becomes much easier. Microsoft has written a guide and a tool to help. The guide is called *The MFC Migration Guide*, and the tool is called *The MFC Migration Tool*.

Together, these items help you to move a body of code from C to MFC. There are, however, some initial steps that you must perform before you use the migration tool. Let's go over these steps in more detail. For the overall procedures and more information, see *The MFC Migration Guide* in the Microsoft Developer's Network CD.

The first step in getting your code ported to MFC is to make the C code as clean as possible. The best way to achieve this is by compiling the code with the highest possible warning level you can use (at least level 3).

Once the code is clean, you can begin to use the migration tool. The tool (**`Migrate.exe`**), looks at the code (like PORTTOOL does) using a set of porting guidelines. It will alert you of any porting issues such as, "**WinMain** is not needed in an MFC application". You're allowed to make the changes immediately, since the tool can act as a simple editor.

When you're porting your code, the guide says that you should perform the task in three different phases. Phase one involves migrating the**`WinMain()`** and **`WndProc()`** code. In phase two, you need to get rid of your message-handling **`switch`** statements. Finally, in phase three, you integrate further with MFC. Following this set of phases allows you to safely move your code while still allowing it to run correctly at each phase, enabling you to debug your code immediately if something goes wrong.

Phase one begins with you having to rip out your **`WinMain()`** function, since MFC provides one for you. Since most Win32/C applications receive their window messages into a window procedure, you'll likely have a large **`switch`** statement with a long list of cases in between. You're probably wondering, "What the heck are you going to do with this large **`switch`** statement once you port the code to MFC?". Well, the good news is that you simply move your **`WndProc()`**code into MFC by overriding of the **`WindowProc()`** member function in your MFC-derived view class and copying your window procedure code into the **`WindowProc()`** function. As for the command messages, you should move those into another override, called **`OnCmdMsg()`**, which is another member function in your MFC-derived view class. The **`case`** entries of the **`switch`** statement will later become individual message-mapped handler functions in your application.

Another code section that you should move is the code provided for your **`WM_PAINT`** code. This code belongs in an

override of the **OnDraw()** member function in your MFC-derived view class.

Since MFC provides a thin wrapper for most of the Win32 APIs, there will be times when you end up with a function call in your application that has already been included in a class with the same name (perhaps even a class for which you're providing overrides). You should prefix these function calls with the C++ scope resolution operator (**::**) to distinguish it from the MFC equivalent. For example, you might have called **ShowWindow()** or **GetClientRect()** from your application without having any prior knowledge that the **CWnd** class has already provided a function with the same name, that, as a matter of fact, ends up calling the actual Win32 function. Furthermore, if you derive a class from **CWnd** and call one of these functions from within your member functions, you might have problems, or, at the very least, someone else might not understand why you're not simply calling the MFC equivalent.

You should make certain that you get rid of calls to **DefWindowProc()**. This function moves messages to their appropriate place and disposes of messages back to the operating system for you. If you have registered your own window classes (which most Win32/C applications do), you need to check these calls to make sure that they don't collide and that they can coexist with the MFC registered window classes.

Phase two begins when you start to move Windows-based messages into their individual message response functions. A **case** for **WM_KEYDOWN** should become **ON_WM_KEYDOWN**, **WM_COMMAND** should become **ON_COMMAND** (one per command ID), and so on. Instead of handling your menu's user interface by responding to **WM_INITMENU**, you should instead provide **ON_UPDATE_COMMAND_UI** handlers. The use of **cbClsExtra** and **cbWndExtra** should also be replaced with another method, since the window class information is generated by MFC.

In phase three, you'll need to start changing your Win32 API function calls to MFC function calls. As I said before, since MFC provides a thin layer of the Win32 API functions, you'll at some point need to start calling these functions from or to the appropriate classes. There are two other areas where MFC provides great assistance; printing and serialization. The printing work should be provided in your view class and the serialization work should be performed in your document class. You'll need to look at the MFC documentation for more information on these topics.

Once these phases are complete, you'll end up with an MFC application capable of performing any code that involves document/view architecture coding. This means that you can add code to support OLE using the MFC classes, or you can add code to support ODBC or DAO via the MFC classes as well.

The bottom line is that, once you get your application ported to MFC, it becomes much easier to move the application to a 32-bit environment (such as NT or windows 95). It also becomes much easier to integrate new features like OLE or ODBC into your application.

The extra power that your applications receive from MFC means that there's more room for things to go wrong. The key to succeeding and delivering powerful solutions is to learn the application frameworks as much as possible. This way, when something breaks, you can:

Take a deep breath.
Have a drink of caffeine.
Jump right in and fix the problem.

I'd advise that you get familiar with the MFC code, because I've found that this is what has helped me the most. I often take the time to walk through the MFC source code that ships with VC++ (I guess that's the curious nature in me). Try to understand the purpose of the main functions or overrides in MFC (such as **CWinApp::InitInstance()** and **CView::OnDraw()**).

Having this knowledge will make the difference later on, when something breaks and you need to fix it or find a way around it. Knowing how the MFC source code is arranged will also allow you to provide solutions faster. For example, let's say that your boss asks that you make the dialog boxes in your application have a red background instead of the normal gray. Where would you look for this setting so that you can change it? Or what if you had to

provide your own window class instead of the default? Simply spending sometime in the MFC code and getting to know it a little better can really benefit you in the long run.

Porting MFC Applications to 32-bit

If you've been working with MFC for a while, you've probably built a lot of 16-bit code either into applications or tools and libraries. Chances are that you'd like that code base to follow your development efforts and future endeavors. The question is, "How do you move 16-bit MFC code to a 32-bit platform?".

I can personally tell you that I've had the wonderful experience of porting MFC code from 16-bit to 32-bit and it turned out to be pretty easy, especially since there was very little to change in the first place. There are however, a few things that you need to look out for.

Differences between MFC Versions

Some functions and classes have been added or deleted, and there are certain other changes that you need to be aware of.

Porting applications from a 16-bit to a 32-bit version of MFC is painless if you haven't been using VBXs, inline assembly code, 16-bit **`int`**s, or 16-bit only API calls (such as some of the DOS functions).

If you have ever used a view object in your application (instantiated from the **`CView`** class or one of its derivatives), there are a few changes and additions in this area. First of all, the **`CEditView`** has a new parent (it has been adopted). The new parent class is derived from **`CView`** and is called **`CCtrlView`**. Microsoft thought that **`CEditView`** should share the light with a couple of sibling classes, so they made **`CCtrlView`** the parent to three new classes, **`CListView`**, **`CRichEditView`** and **`CTreeView`**. In case you've gone ahead and provided your own control wrapper classes (such as **`CEditView`**), to give your code more uniformity, you might want to rethink things and derive your custom class from **`CCtrlView`** .

As you probably have already imagined, all of these classes have one thing in common: they all support a Windows common control within their respective client areas. The **`CEditView`** class has always contained an edit control within its client area, but now the other classes have followed suit.

The benefit of deriving all of these classes from a common base class is that now you can share common code with all of them. If you need to provide a class that supports another control or maybe an OLE custom control, you can simply derive your view from **`CCtrlView`** and implement code similar to the other control classes.

Project Conversion

The first thing you'll have to do is to convert the project file from its native 16-bit format to its 32-bit counterpart. The VC++ environment will ask if you want to do this when you attempt to load the project file under the 32-bit environment.

Whether you're converting the project from 16-bit VC++ or an older version of 32-bit VC++, the conversion will still work.

No More VBXs

If you use VBX controls in your application (either for windows or dialog boxes), you'll need to remove them. You'll also need to remove any calls to classes or functions involving VBX (since they don't exist in the 32-bit version of MFC).

I once used VBX controls and found that it's not easy to move to 32-bit development without them. Fortunately, most major VBX vendors have ported their VBXs to the 32-bit platform as 32-bit OLE custom controls (OCXs).

The best part of all is that, from version 4.0, Visual C++ has built-in support for OLE control containers, which makes it easy to place the controls onto dialogs and windows.

This will alleviate the torture of replacing VBXs with OCXs. However, if the manufacturer who produced the VBX that you were using has not written a 32-bit OCX equivalent, you have three choices. Number one, wait until the manufacturer produces one; two, switch to using a different OCX from another manufacturer (which involves rewriting some of your code to work with the new control); or three, write your own control (providing the same interface as the VBX for the control).

Pen Computing

You'll also need to rip out any code that uses the 16-bit Windows for Pens extensions. The extensions are implemented in the **`CHEdit`** and the **`CBEdit`** classes in MFC. As yet, these classes are not available for Win32 and, therefore, 32-bit applications written using MFC (unless you implement your own classes for this purpose).

Assembly Language

You should replace any assembly language implemented in your code with C or C++ code. This makes your code more portable across other Win32 platforms. If you want to keep your code as assembly language, keep in mind that you'll have to provide assembly for all of the platforms that your code will run on (if it's on different hardware). Using assembly language doesn't necessarily mean that you used the **`_asm`** keyword. You might have called an API function to execute an **`Int21`** function (for possible file I/O). You should replace these functions with the MFC or Win32 equivalents.

In earlier versions of VC++, you had to explicitly tell the linker which library files it should link to the application. This is no longer necessary, since VC++ now detects the libraries and links to the appropriate libraries. This means that you should remove any list of libraries from your project and let VC++ do the work for you.

Strings

If you have done any OLE development, either under VC++ 1.x or VC++ 2.x, when you compile your code, you'll probably get a bunch of errors. The reason is that MFC contained a DLL which converted MBCS characters to Unicode (which is what OLE APIs and interfaces require). This DLL was called**`Mfcans32.dll`** and handled converting back and forth between the character formats transparently. The advantage is pretty obvious: you didn't have to perform the conversion yourself. However, there was a disadvantage: you were hit with a performance overhead.

With the introduction of VC++ 4.x, Microsoft has stopped shipping the **`Mfcans32.dll`** and now requires that you perform the conversions yourself. The MFC code has been revamped to support converting to the different character formats, but if you make any calls to OLE API functions or interface functions, you'll also have to perform the conversion when you call those functions.

Microsoft now provides some macros (we all know how much they love macros) to perform the conversion on the fly. The macros are included in a header file, called **`Afxpriv.h`**, which you must include in your project if you want to use them.

They have included macros to convert from MBCS to Unicode and Unicode to MBCS. As you might know, when you work with OLE, it also supports a portable string type, called **`OLESTR`**, which actually is a Unicode string under Win32.

Keep in mind that, to some degree, MFC also supports string portability in the **`CString`** class via the **`TCHAR`** type. This type can be anything (MBCS or Unicode). If you need to send a string in a portable manner to a **`CString`**, you

simply use the **_T** macro as follows:

```
CString my_string = _T("This is a string");
```

The macro will automatically convert the string to the native format supported in the compilation (MBCS or Unicode). You can control which format is used when you compile via the project settings.

Microsoft has also provided macros to convert from **TCHAR**s to **OLESTR**, but these macros are not dependent on some project setting or flag. Instead, they use the underlying Win32 conversion functions to perform their magic. Here are the macros, with a short description:

Macro Name	Description
A2CW	MBCS to constant Unicode string
A2W	MBCS to Unicode string
W2CA	Unicode to constant MBCS
W2A	Unicode to MBCS
T2COLE	Portable type (**TCHAR**) to constant OLE (Unicode under Win32)
T2OLE	Portable type to OLE (Unicode under Win32)
OLE2CT	OLE to constant portable type
OLE2T	OLE to portable type

When you use these macros, you must put a special macro, called **USES_CONVERSION**, at the top of the function performing the conversion. This defines an automatic variable, called **_convert**, which is used by the other macros. Don't forget to include the **Afxpriv.h** in the files that must make use of the string conversion macros.

The macros allocate a string to receive the converted value and then return it, so that any function making use of the macro can convert on the fly, as follows:

```
USES_CONVERSION;
OLEAPI_FUNCTION(T2COLE(lpsz));
```

The memory allocated to contain the converted string is taken from the stack (not from the heap as in other allocations). The reason for this is because allocating memory from the stack is much faster than allocating memory from the heap, and you don't have to free the memory when you're done with it. When the function exits, the memory will automatically be freed. Keep in mind that, since the memory is allocated from the stack, you don't want to place calls to these macros in a loop. This can drain your stack, causing your application to crash. The following code is a bad idea and an example of what you should *not* do:

```
void SomeFunction()
{
   USES_CONVERSION;
   for(int I = 0; I < nMax; I++)
      Call_OLEFunction(T2COLE(lpsz));
}
```

Instead, this should be coded like this:

```
void SomeFunction()
{
   USES_CONVERSION;
   LPCOLESTR lpszOLE = T2COLE(lpsz);
   for(int I = 0; I < nMax; I++)
      Call_OLEFunction(lpszOLE);
}
```

32-bit Issues

The address space issue is one that I can't stress enough. Basically, the segmented architecture is gone … for good (thank goodness). You no longer have to worry about moving between 64K segments, memory models and far calls (no more long pointers). Feel free to use things like **`CRect*`** instead of **`CRect FAR *`**.

Keep in mind that, since **`int`**s are now 32-bits long, the size of any structure containing them grows. This makes our **`wParam`**s, **`CPoint`**s, and many more, much larger than they were under 16-bit Windows.

Since MFC manages unpacking **`wParam`** and **`lParam`** structures for notifications, events and commands internally, you won't have to worry about this yourself. MFC automatically unpacks the values and passes them to functions that you can override in your applications or allow them to perform their respective jobs. As I mentioned, this involves taking the values and passing them to handler functions in your code (if the handlers exist).

The functions that you can override are **`CWnd::OnCommand()`**, **`CWnd::OnParentNotify()`**, and **`CWnd::WindowProc()`**. When you follow *The MFC Migration Guide*, these are the functions that you will override to get your code up and running as a base MFC application in your initial phase. If your application provides message and command handlers using MFC's message maps, you obviously won't need to override these functions (which is what we want to achieve in the second phase of the migration guide).

32-bit Issues and Platform Differences

After everything that has been said about porting your software between 16-bit and 32-bit Windows, you might think that, by now, you should know everything there is to know about it. Well, guess again. Although, we've been talking about the differences between 16-bit and 32-bit Windows, there are some differences that exist even within the Win32 implementations.

Platform Differences

When you write an application using the Win32 API, chances are that your application will run on both Windows NT and Windows 95. Microsoft has gone to great extremes to make sure that applications can be written using one API that will be compatible with all Windows operating system implementations. In some instances, you don't even have to recompile the source code to run the application on the target Windows platform. With other platforms, making the application run normally is just a matter of recompiling the source code under the given platform.

> *If you're using MIPS or Alpha versions of Windows NT, this isn't strictly true, as both contain Intel op-code emulation. However, this significantly reduces the performance of the application, so you should avoid it if possible.*

The Win32 API uses the same function names, messages and structures across the different flavors of Windows. Although the Win32 may not be different, there are features that might not be completely implemented under a particular version of Windows. For example, under Windows 95 and Win32s, there's no security or Unicode support. In Win32s, there's no multitasking either. The most complete Win32 implementation can be found in Windows NT. For this reason, I usually try to perform my Windows development under this platform (unless I'm implementing something not supported under Windows NT, such as Windows 95 shell extensions).

If you intend to port your application to a standard version of Win32 using functions and messages supported by both Windows 95 and NT, you need to understand the difference between the operating systems.

Windows 95 Limitations

As yet, Windows 95 doesn't contain a full implementation of the Win32 API. When you call a Win32 API function not supported under the Windows 95 implementation, it provides a stub function which returns an error.

Unicode

One such limitation is Unicode, which Windows 95 doesn't currently support. Calling a Unicode function, however, doesn't crash your application. As I mentioned before, Windows 95 provides stub functions. In the case of Unicode, Windows 95 will return an error message back to the caller for most cases (or APIs). There are, however, a few Unicode functions for which Windows 95 will provide limited implementation. These include:

```
ExtTextOut()
GetCharWidth()
GetTextExtentExPoint()
GetTextExtentPoint()
MessageBox()
MessageBoxEx()
TextOut()
```

In addition, Windows 95 implements the **`MultiByteToWideChar()`** and **`WideCharToMultiByte()`** functions for converting strings to and from Unicode.

Window Management

Remember how they said that 16-bit limitations would go away with Windows 95? Well, I'm here to tell you that some of these limitations still exist. For example, the standard edit control is still limited to a maximum of 64K of data (in the multiline version, it's only 32K for single-line controls).

Some Win32 API functions are still limited to 16-bit values, even though they receive the values via a 32-bit integer. The reason for the limitation is that Windows 95 still needs to store or process the value internally as a 16-bit value. Some API functions even thunk down to a 16-bit API (that might have stuck around from Windows 3.x). An example would be the GDI functions (more on GDI later). These limitations and restrictions are still around, due to the fact that Windows 95 has to be more backward compatible with Windows 3.x and DOS than Windows NT.

In Windows 95, the `wParam` parameter in list box messages, such as `LB_INSERTSTRING` or `LB_SETITEMDATA`, is limited to a 16-bit value. One effect of this limit is that list boxes cannot contain more than 32,767 items. Although the number of items is restricted, the total size of the items in a list box, in bytes, is limited only by available memory.

Although the limit of available window and menu handles has grown from the Windows 3.x days, the new limit is still lower than Windows NT. Windows 95 is restricted to 16,364 window handles and 16,364 menu handles.

Windows 95 now makes use of values at `WM_USER + 0` to about `WM_USER + 99`. Anything above `WM_USER + 100` is safe for your applications to use as private messages. I personally prefer to use `RegisterWindowMessage()` to retrieve a unique message ID from the system.

Graphics Device Interface (GDI)

Windows 95 uses a 16-bit world coordinate system and restricts *x* and *y* coordinates for text and graphics to the range ±34,816. Windows NT uses a 32-bit world coordinate system and allows coordinates in the range ±2,097,152. If you pass full 32-bit coordinates to text and graphics functions in Windows 95, the system truncates the upper 16 bits of the coordinates before carrying out the requested operation.

Windows 95 doesn't support world transformations that involve shearing or rotation. In addition, OpenGL, the standard 3D graphics API under Windows NT, is not available for Windows 95 at the time of writing. However, I understand it's in beta and will probably be available by the time you read this book.

There are a couple of limitations under Windows 95 for pens and pen styles. It doesn't support the dashed or dotted pen styles, such as `PS_DASH` or `PS_DOT`, in wide lines. The `BS_DIBPATTERN` brush style is limited to an 8-by-8 pixel brush.

As far as brushes are concerned, Windows 95 doesn't support brushes from bitmaps or device-independent bitmaps (DIBs) that are larger than 8-by-8 pixels. Although larger bitmaps can be passed to the `CreatePatternBrush()` or `CreateDIBPatternBrush()` function, only a portion of the bitmap is used to create the brush.

The Kernel

Since the first version of Windows, the kernel has always handled things like file I/O, error handling, date functions, memory management and couple of other duties. In Windows 95, this has not changed. However, there are a couple of differences in the way that Windows NT and Windows 95 operate and accomplish these tasks.

If your application is calling either `FileTimeToDosDateTime()` or `DosDateTimeToFileTime()`, you can run into trouble if you're not aware of the differences for the return values. In Windows NT, these functions allow dates up to 12/31/2107. In Windows 95, these functions allow dates up to 12/31/2099.

Deleting files with `DeleteFile()` while the file is still open will fail under Windows NT and will succeed under Windows 95. However, be aware that it's always a bad idea to delete a file while it's open, since you might cause bad

side effects across the rest of the system.

Memory-mapped Files

When you start to use memory-mapped files in your application, you'll find that there are a couple of differences between Windows NT and Windows 95. I won't explain how to create memory-mapped files, since there are already so many books that cover this topic in much more detail than I can provide in this section. Just remember that the name space used for memory-mapped files, is the same as that used for events, semaphores and mutex, and it's not possible to have objects with the same name in the same name space. If you attempt to create an object of one type (such as a semaphore) with the same name as another object of another type (such as an event), you'll get an error and the creation function will fail.

In Win32, you can open a disk file as a memory-mapped file and allow Windows to page the file blocks in and out for you as you access different parts of the file. Both Windows NT and Windows 95 limit the size of a file mapping by the available disk space. In Windows NT, the size of a mapped view of an object is limited to the largest contiguous block of unreserved virtual memory in the process performing the mapping (at most, 2 GB minus the virtual memory already reserved by the process). In Windows 95, it's limited to the largest contiguous block of unreserved virtual memory in the**shared virtual arena**.

The shared virtual arena on Windows 95 is the area shared by certain components (such as 16-bit Windows-based applications) and non-overlapping memory mapped views. Under Windows 95, these views are mapped to the 2-3 gigabyte address range (which is exactly where the shared virtual arena is located).

The arena will be at most 1 GB, minus any memory in use by other components of Windows 95 which use the shared virtual arena (16-bit Windows-based applications). Each mapped view will use memory from this arena, so this limit applies to the total size of all non-overlapping mapped views for all applications running on the system.

In Win32, you can also specify a larger size to **`CreateFileMapping()`** than the actual file being mapped. Under normal conditions, Win32 will grow the file to match the size specified. However, NT will fail if you specify **`PAGE_WRITECOPY`** as the **`fdwProtect`** parameter to **`CreateFileMapping()`**, whereas Windows 95 will not.

When you map a file to a view (as it's called), the view is created in the process space of the running application. Under NT, the address falls within the range 0 – 2 GB. In Windows 95, it's between the range of 2 – 3 GB.

When you call **`MapViewOfFileEx()`**, you must specify the **`lpvBase`**. On Windows NT, not specifying such a value will cause **`MapViewOfFileEx`** to fail. In Windows 95, the address is rounded down to the nearest integral multiple of the system's allocation granularity. To determine the system's allocation granularity, call **`GetSystemInfo()`**.

You can always fall back to using **`MapViewOfFile()`** instead of the extended version, which means that you don't have to specify the base address. The operating system will determine the address of the view for you. There are a couple of reasons why you might want this to happen.

You might think that you can use the same address for all mapping of a view from all applications wishing to share memory with each other, but, on Windows NT, this might backfire. The specified virtual address range may not be free in all of the processes involved and the mapping could fail for one of the processes.

In Windows 95, all views to the same file object are mapped to the same address by default. Therefore, it's useless to try and map it yourself, since the operating system will do it for you. When you're attempting to map the first view at a predetermined address, that address may already be in use by other components of Windows 95 which use the shared virtual arena.

There are a few more items that we need to cover as we talk about using memory-mapped files under Windows 95. First of all, as you know, one of the parameters to the **`MapViewOfFile()`** and the **`MapViewOfFileEx()`** functions is the **`dwOffsetHigh`** parameter. If you're using it with Windows 95, you should set this parameter to zero, since you shouldn't specify which byte to map onto the view because the byte that you would specify could potentially

map onto something being used already by another view.

The **CreateFileMapping()** function normally receives two parameters **dwMaximumSizeHigh** and **dwMaximumSizeLow**. These parameters should be set to zero under Windows 95 so that the maximum size of the file-mapping object will be the same size as that of the file specified. Another parameter to note is the **fdwProtect** argument. This parameter is used to pass flags to the function. There are two flags that are the not supported under Windows 95: **SEC_IMAGE** and **SEC_NOCACHE**.

When two or more processes in Windows 95 use a view that has been marked with the **PAGE_WRITECOPY** protection flag, they are allowed to view the changes made to the data from another process. In other words, when one process makes changes to the data in the view, the other application immediately sees the changes. However, because the data is marked as **PAGE_WRITECOPY**, the changes are not written to disk.

In Windows NT, this scenario works a little differently. When a process viewing a named data map chooses to write to the data, the process is given a separate copy of the data. Other processes are not effected, since they don't see the changes to the data. The disk file isn't effected at all.

Multithreading

One of the major design changes implemented in Win32 is the asynchronous input model, whereby a single program can have multiple threads of execution, each thread receiving its own message queue. The effects of this are seen in APIs related to mouse capture, the active window and querying window focus.

The Win32 implementation of the **Get** APIs (**GetFocus()**, **GetActiveWindow()**, **GetCapture()**) query information solely on the current thread. Now, in Win32, it's possible for **GetFocus()** to return **NULL**. This occurswhen the thread that issues the **GetFocus()** doesn't own the window with focus. This same problem affects the **GetActiveWindow()** and **GetCapture()** APIs. If a thread attempts to **SetFocus()** to a window that it didn't create, the thread's focus status is set to **NULL**. The thread that owns the window specified by **SetFocus()** will have its focus status set.If a thread attempts a **SetActiveWindow()** on a window not created by itself, the thread issuing the API will have its active window status set to **NULL**. The thread that owns the window specified in **SetActiveWindow()** will have its active window status set.

Any 16-bit Windows code that assumes the **Get** APIs always return a valid window handle will have to be modified for Win32.

The Registry

When you begin to port your application to Windows 95 or Windows NT, your code is not the only thing you have to worry about. You'll now have the pleasure of working with the system's **registry**. Although Windows 3.x had the concept of a system registry, Win32's system registry is far more extensive and powerful.

The system registry is a place provided by the operating system to act as a database of information where the system, applications, or users can place persistent information. This information has life across sessions and can be accessed programmatically or via a tool provided by the system(**Regedt32.exe** in Windows NT or **Regedit.exe** in Windows 95). You can also use the registry to store temporary data (although I'd prefer to use other tools for this sort of stuff).

What do You Place in the Registry?

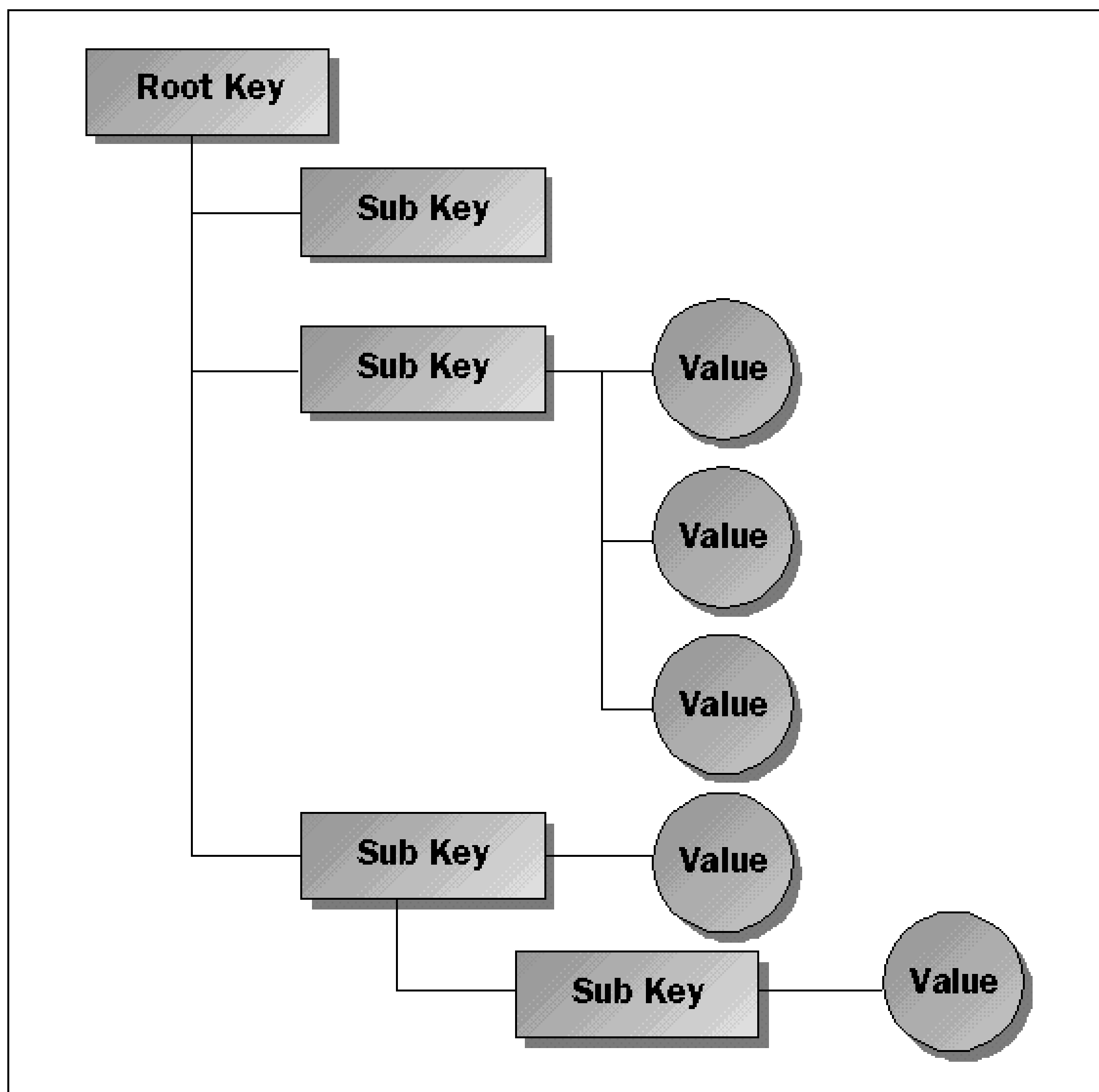
Windows 3.x had no central place to store configuration information. Although it had a registry, it was limited, so was only used by the operating system for internal use.

Now the rules have changed. Any information that would have been place into **`.ini`** files under Windows 3.x should now be placed in the registry for Win32-based applications. Most Windows 3.x applications kept the majority of their configuration information in private **`.ini`** files, although some information was also placed in the system's **`Win.ini`** and **`System.ini`** files.

Applications also created group (**`.grp`**) files which contain location and icon information for the application(s). All of these files must be maintained using various methods and functions, which can add to confusion and problems. The solution is to have one central place where all of the information can be placed and maintained in an organized manner.

The registry provides a hierarchical view of the information. It's organized using nodes, where each node is called a **key** and each key can contain values and/or other subkeys. Subkeys can also have other subkeys within them. The registry has six main keys which are known as the **root keys**. The root keys provide an entry point for other keys.

Each key can have a default value and other named values associated with it. Note that the key doesn't need to have any value at all. The figure below shows how the hierarchy is organized in the registry:



To use or modify the registry, you need to provide the handle of an open key. But how do you provide the handle of an open key if you haven't opened the key yet? This is like asking, "Which came first, the chicken or the egg?". For this very reason, Win32 automatically opens the six root keys and the handle of these keys is known at compile time. These predefined root keys are stored in the Win32 SDK header files and are listed here:

Key	Description
HKEY_CLASSES_ROOT	Mostly used to provide OLE support. All OLE class IDs are stored under this key. Type libraries, file viewers and Windows 95 shell extensions are also stored under this key.
HKEY_LOCAL_MACHINE	Contains information about the system's local state. Information includes the computer hardware, drivers, I/O ports and other operating system software components. This information is used for whatever user is logged on.
HKEY_USERS	This key contains a subkey, called **.Default**, which is used to create a user's profile for users with no relevant profile on the system. In addition to the **.Default** subkey, **HKEY_USERS** also contains all the user profiles for the users that have logged on to the system.
HKEY_CURRENT_USER	Contains the profile of the user who is currently logged on. If the profile is available across the machines that the user can possibly log on to, the user is guaranteed to always have the same user interface settings. This key can be seen as containing the information necessary to maintain environmental settings such as application preferences, screen colors, and security access permissions.
HKEY_CURRENT_CONFIG	This key contains hardware-specific information pertaining to the current set of hardware plugged into the computer. This key is new to Windows 95.
HKEY_DYN_DATA	The information maintained in this key can change dynamically. It plays an active role in the plug & play implementation. This key is new to Windows 95.

Working with Keys and Paths

Although you can open a subkey from a root key, then open a subkey within the first subkey, there's a better way to perform this task. Using a path (very similar in concept to disk-based paths), you can open a key three or four levels down, directly. A path is a string containing the hierarchy of keys ending with the key that you wish to open. The keys are separated in the string with a slash (\) character. This is a path to the information stored for the Microsoft Access key in the **HKEY_CURRENT_USER** key:

```
\SOFTWARE\Microsoft\Access
```

By passing this path to the **RegOpenKeyEx()** function along with a handle to a key (**HKEY**) for **HKEY_CURRENT_USER**, you would receive a **HKEY** back. You can then use the **HKEY** in any subsequent calls to the Win32 registry API functions.

Using the Registry API

You should normally replace any calls to the **.ini** functions with the registry functions. Under Windows NT, you can get Windows NT to do it automatically for you at the system level. This works out great if you need to use one code base for both Windows 3.x and Windows NT. To achieve this, you must tell Windows NT the name of the **.ini** file you would have created and place the name in the **HKEY_LOCAL_MACHINE\Software\Microsoft\WindowsNT\CurrentVersion\IniFileMapping** key. For example, on my machine the Clock application that ships with Windows NT has a subkey under this key, called **Clock.ini**. This key has a value of **Software**

Microsoft\Clock. I then looked under the **HKEY_CURRENT_USER\SOFTWARE\MICROSOFT\CLOCK** key and found the following values:

```
"Maximized: 0"
"Options: 0, 1, 0, 0, 0, 0"
"Position: 0, 0, 804, 476"
```

Windows 95 currently has no provision for achieving this, but if you use MFC, you can have this done for you. You must call a function, named **CWinApp::SetRegistryKey()**, in your **InitInstance()** before calling any MFC **.ini** functions. This will cause MFC to redirect any call to the **Get**/**SetPrivateProfile**()family of members of **CWinThread** to the registry equivalents. The table below gives a brief description of the registry API functions. For a more descriptive explanation of the API, see your books online or Win32 SDK reference manuals.

Registry API	Description
RegCloseKey()	Releases the handle of the given key.
RegConnectRegistry()	Establishes a connection to a predefined registry handle on another computer.
RegCreateKeyEx()	Creates a specified key.
RegDeleteKey()	Deletes the specified key.
RegDeleteValue()	Removes a named value from a specified key.
RegEnumKeyEx()	Enumerates the subkeys of a specified key.
RegEnumValue()	Enumerates the values for a supplied open key.
RegFlushKey()	Writes all the attributes of the given open key immediately (if any information is still in the cache).
RegOpenKeyEx()	Opens a specified key for subsequent operations.
RegQueryInfoKey()	Retrieves information about the characteristics of a key.
RegQueryValueEx()	Retrieves the type and the data associated with an unnamed value in a registry key.
RegSetValueEx()	Stores data in a value field of a specified registry key.

Dynamic-link Libraries

Applications aren't the only modules that need to be ported to Win32. Dynamic link libraries (DLLs) are also good candidates for porting to Win32. Before you start the porting process for DLLs, there are a couple of things that you should know about.

Memory Consumption

Although DLLs have not changed much since they were first introduced into Windows development, they have been changed slightly to take advantage of some new features offered by Win32, such as memory management, multitasking and separate process address spaces.

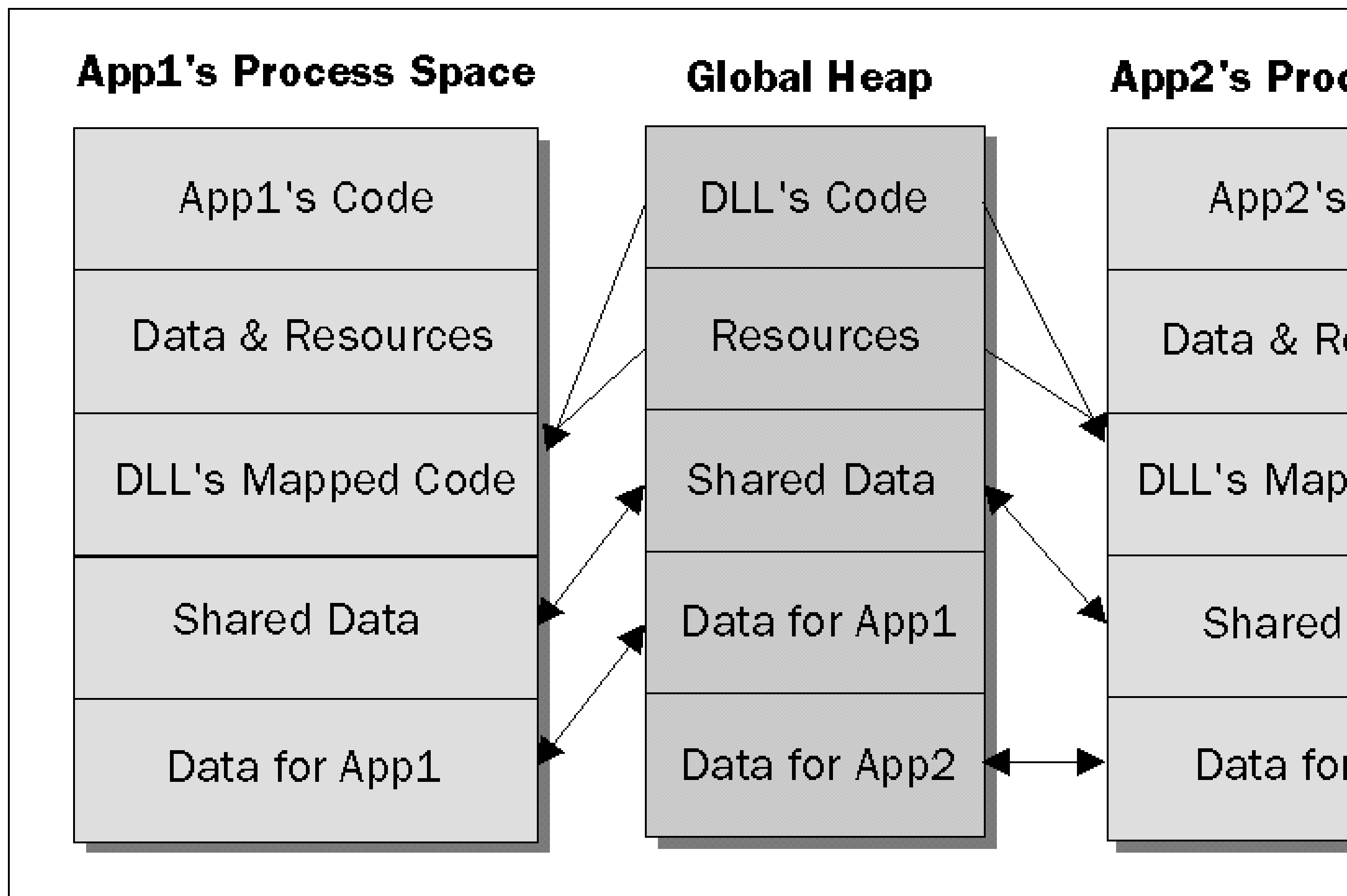
Since all memory owned by an application (including code) resides in the linear address space owned by the process, it's not possible to share data with other applications. This causes Win32 to map a DLL into each process space that must use that DLL. In essence, this causes the DLL to become reentrant, since each process can call the DLL at virtually the same time (a.k.a. multitasking) from any thread.

The DLL isn't really copied to the address space of each process. Rather, it's loaded once into the global heap and mapped to the address space of each application that loaded the DLL.

Global memory and global C++ objects (objects created globally in your application and stored in the data segment) are treated differently in Win32. They aren't shared by default, as they were in Windows 3.x. A copy of the data is actually copied for each process which loads the DLL. The memory is allocated from the global heap, but is marked as non-sharable memory. It's simply mapped to the process space of each application. There's a way that you can share data allocated within the DLL amongst the different processes. This involves marking the data as **`SHARED`** data and dictating how you want to share the data (**`READ`** or **`WRITE`**).

If you decide to share data with several processes at the same time, it's imperative that you manage the shared data very carefully. Since Win32 now has multitasking, two threads can access a piece of shared memory at the same time. For this very reason, Win32 now supports synchronization objects which can be used to synchronize access to shared memory.

The following figure illustrates how the DLL and memory is mapped into the address space of two different processes:



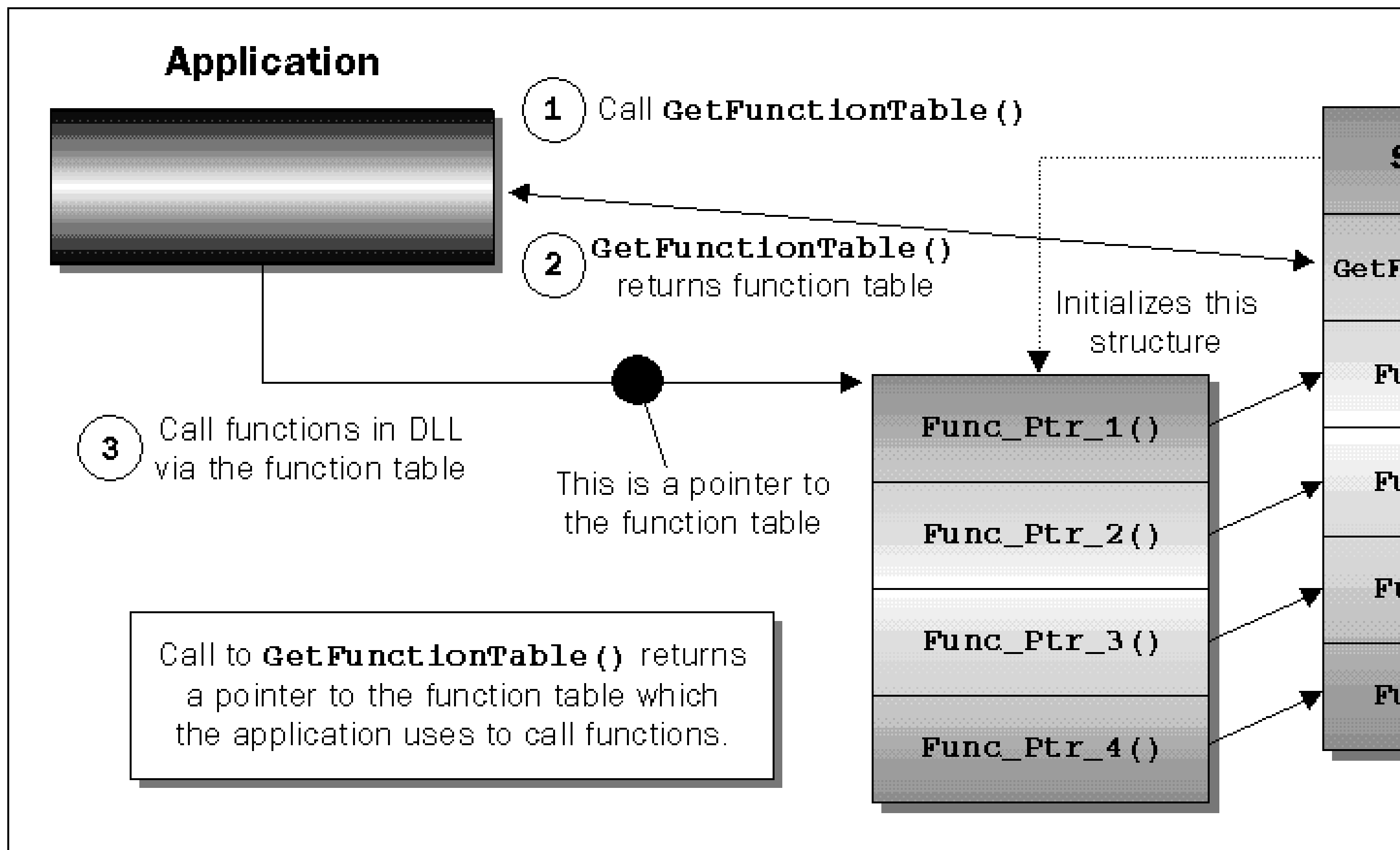
Loading DLLs

As in Windows 3.x, when an application needs to communicate with a DLL, it first needs to have the DLL loaded. You can load a DLL in two ways: implicitly and explicitly.

The implicit loading of a DLL takes place when you link to a library file created from the DLLs tokens. The tokens are the pieces of information that describe the location of the functions in the DLL, as well as the DLL itself. Most developers use Visual C++ to generate a library file containing information (the tokens) which allow applications to call functions in the DLL. The functions simply act as stubs until run time, when the functions can then be linked in dynamically.

An application can also load a DLL explicitly, which means that the application calls **`LoadLibrary()`** at run time. This allows the application to link to any functions that are located in the DLL which the application is calling. Loading a DLL like this allows you greater power, since the application can control when the DLL is loaded or thrown out of memory. The downside is that you must call **`GetProcAddress()`** to retrieve the function's address in the DLL.

You could, however, allow the DLL to set up the addresses to the functions in a structure using function pointers (or a VTABLE in C++) and return a pointer to the structure containing the function pointers via a function you call in the DLL. This procedure is illustrated in the following figure:



Calling functions using this procedure is very much how OLE allows applications to communicate via interfaces. The interfaces are actually these function table structures (or VTABLEs in C++).

Win32 Entry Point

In Win32, **`LibMain()`** and **`WEP()`** (Windows 3.x's entry points) have gone away. Instead, Win32 DLLs can implement a new function, called **`DllEntryPoint()`**, for both cases. The function receives a parameter, called **`dwReason`**, which will contain one of four values: **`DLL_PROCESS_ATTACH`**, **`DLL_PROCESS_DETACH`**, **`DLL_THREAD_ATTACH`** and **`DLL_THREAD_DETACH`**. Your application first determines why it's being called (startup or shutdown), then, if it's being shutdown, it can proceed to allocate or deallocate resources.

Keep in mind that the **`DllEntryPoint()`** function for a DLL is called once for every process that loads the DLL with **`DLL_PROCESS_ATTACH`** as the **`dwReason`** parameter. In response to this, the DLL should initialize the data that the application will use, since the application will end up with its own copy of the data in the DLL. Similarly, when you need to free resources, you should free them once per process, since memory will be allocated on behalf of each process.

Since DLLs are reentrant, the entry point is called for each new thread that loads the DLL with **`DLL_THREAD_ATTACH`** as the reason (except for the first thread which sends **`DLL_PROCESS_ATTACH`**). The DLL will receive **`DLL_THREAD_DETACH`** when the thread lets go of the DLL or the thread is terminated. If the thread is the primary thread of the process, the DLL will instead receive **`DLL_PROCESS_DETACH`**.

If a thread loads a library more than once, the DLL's entry point will only be called once (not once per load, only once per thread). On the other hand, the DLL count will be incremented, so you should free the library as many

times as you load it. The system will automatically decrement the reference count on the DLL (the number of times that the thread loaded the DLL) when a thread is terminated.

Exports and Imports

Functions that are currently declared with the **`__export`** keyword must be changed to be declared with **`__declspec(dllexport)`** in Win32. You no longer have to explicitly declare an exported function in the **`.def`** file. The **`__declspec`** keyword will automatically export the function for you.

You must convert any functions that were being imported with the **`__import`** keyword to use the **`__declspec(dllimport)`** keyword. Using this keyword is not essential, but if you do use it, you'll get improved performance.

You will also get compiler warnings about certain sections in your **`.def`** file that have changed for Win32 development. For example, the **`EXETYPE`** keyword has been dropped and is no longer necessary.

Summary of Changes

Building a DLL has actually become easier in Win32. There is no separate startup module. The DLL startup code is handled for your DLL by the code that is linked to your DLL.

Global objects defined in the DLL are duplicated for each process. The DLL manages this by initializing and calling the constructor for the objects automatically.

The entry point functions have been reduced to one function instead of several (as it was in Windows 3.x), and you can name the startup function anything you want, as long as you include the appropriate parameters and return type for the function.

Importing and exporting of functions has been simplified by the use of the **`_declspec`** keyword and there's no need to place functions into a **`.def`** file.

Remember that memory is no longer shared by default. Shared memory can still be achieved, but you must manage things more carefully if you choose this option.

COM and OLE

In this chapter, we'll examine COM and OLE. You'll see how MFC supports this object technology through its classes and macros and how this allows you to implement advanced features in your own applications. MFC includes support for many OLE features that we'll be looking at in this chapter, including:

- Aggregation
- Structured Storage
- Uniform Data Transfer
- OLE clipboard operations
- Drag-and-drop

If you're not already an OLE expert, you may have thought that you could keep holding out on learning OLE forever, or at least until you really needed it for implementing compound document or automation support. Well, I'm here to tell you that you can't put it off any longer.

You may have a basic knowledge of how MFC communicates with COM so that you can build your own OLE objects, but if you don't have a **comprehensive** insight into precisely how MFC operates on your behalf, you may have a hard time changing MFC's behavior or tracking down bugs if something goes wrong. You might be asking yourself, "Why would I need to modify MFC's behavior if it provides everything I need?" Well, if it does provide everything you need, that's fine, but MFC can't provide classes for everything. There will be times when you need to provide your own interfaces to handle things that MFC doesn't do very well (or doesn't do at all). Programming for the Windows 95 shell (covered elsewhere in this book) is just one example.

COM Basics

COM is a protocol and a binary standard that allows software objects to communicate at run time without any prior knowledge of what the other object can do. Since COM is a binary standard, objects written in any language can communicate and interact with each other, regardless of whether the objects are in the same process, in separate processes on the same machine or communicating across a network.

A COM object is made up of a collection of **interfaces** (to COM, an interface is simply a table of function pointers). In C++, it's very easy to create a table of function pointers by using the vtable that is automatically created when a class possesses **`virtual`** functions. In C++, an interface turns out to be nothing more than an abstract class with a bunch of pure **`virtual`** functions. Now, as you know, you can't have a C++ object instantiated from an abstract class, so to expose the interface in a useful way, you must derive a new class from the interface (or abstract class) and implement every single one of the functions before you can instantiate an object.

COM allows one object to communicate with another without knowing everything about its facilities by defining a standard interface called **`IUnknown`**. **`IUnknown`** contains three functions: **`QueryInterface()`**, **`AddRef()`** and **`Release()`**. **`AddRef()`** and **`Release()`** are used to manage the lifetime of an object. **`QueryInterface()`** is used by a caller to request a pointer to a specific interface. If the object doesn't support that interface, the function will return an error code. In this way, the caller can find out which of the features that it's interested in are supported by the other object and use only the ones that are supported.

Every COM object must implement at least the **`IUnknown`** interface. In fact, every other interface must

also contain the three functions that make up **IUnknown**. This means that a client can call **QueryInterface()** on any interface in a server COM object to get a pointer to any other interface on that object.

MFC COM Fundamentals

To see how MFC can help us create COM objects, let's start by considering how we might do it *without* MFC's help. Suppose we wanted to create a simple COM object with a single interface. We might code it something like this:

```
class MyDataObject : public IDataObject
{
public:
   MyDataObject();

   STDMETHOD_(ULONG, AddRef)();
   STDMETHOD_(ULONG, Release)();
   STDMETHOD(QueryInterface)(REFIID iid, LPVOID* ppvObj);

   STDMETHOD(GetData)(LPFORMATETC, LPSTGMEDIUM);
   // ...other functions specific to IDataObject
   STDMETHOD(EnumDAdvise)(LPENUMSTATDATA*);

   DWORD m_dwRef;   // Used for reference counting.
};
```

Here, you can see that we can just derive a class from the interface. We'd have to implement the reference counting and **QueryInterface()** functions ourselves, which is tedious, but the real problem is that most objects have to expose more than one interface, making reference counting and interface lookup more complicated.

Using Nested Classes

If we wanted to support more than one interface in our object, we could use nested classes to group the interfaces into a single unit. Since each interface must implement the standard **IUnknown** functions (**QueryInterface()**, **AddRef()** and **Release()**) we need to devise a method for handling reference counting and interface querying that will work for the object as a whole, as well as each of its interfaces. One way to do this would be to derive the main class from **IUnknown**:

```
class CComObject : public IUnknown
{
public:
   CComObject();

   STDMETHOD_(ULONG, AddRef)();
   STDMETHOD_(ULONG, Release)();
   STDMETHOD(QueryInterface)(REFIID iid, LPVOID* ppvObj);

   DWORD m_dwRef;

   class CDataObject : public IDataObject
   {
   public:
      CComObject* m_pParent;
      STDMETHOD_(ULONG, AddRef)();
      STDMETHOD_(ULONG, Release)();
      STDMETHOD(QueryInterface)(REFIID iid, LPVOID* ppvObj);
      // Define all IDataObject members here.
   } m_dataObject;
```

```
   class COleObject : public IOleObject
   {
   public:
      CComObject* m_pParent;
      STDMETHOD_(ULONG, AddRef)();
      STDMETHOD_(ULONG, Release)();
      STDMETHOD(QueryInterface)(REFIID iid, LPVOID* ppvObj);
      // Define all IOleObject members here.
   } m_oleObject;
};
```

The **IUnknown** functions in the main class must be responsible for maintaining the reference count for the whole object and for returning pointers to any interface through **QueryInterface()**. We still have to provide implementations for **AddRef()**, **Release()** and **QueryInterface()** for the nested classes, but these can simply delegate to the functions in the main class, as shown:

```
STDMETHODIMP CComObject::CDataObject::QueryInterface(REFIID iid,
                                                     LPVOID* ppvObj)
{
   // Delegate to the main object
   return m_pParent->QueryInterface(iid, ppvObj);
}

STDMETHODIMP CComObject::QueryInterface(REFIID iid, LPVOID ppvObj)
{
   \\ All interface queries are handled here
   if (iid == IID_IDataObject)
      *ppvObj = &m_dataObject;
   else if (iid == IID_IOleObject)
      *ppvObj = &m_oleObject;
   else
   {
      *ppvObj = NULL;
      return E_NOINTERFACE;
   }

   return S_OK;
}
```

AddRef() and **Release()** would act in a similar way. Since all the functions delegate to one **IUnknown** implementation, reference counting and interface lookup is made much easier. Of course, this relies on **m_pParent** being correctly set in the constructor for the main class:

```
CComObject::CComObject()
{
   m_dataObject.m_pParent = this;
   m_oleObject.m_pParent = this;
}
```

How Does MFC do it?

MFC works in a similar way, but offers a simpler and more complete solution by providing a class and a number of macros to make your life easier. Two of these macros are **BEGIN_INTERFACE_PART()** and **END_INTERFACE_PART()**, which are used to simplify the declaration of nested classes. They are used as follows:

```
class CComObject : public CCmdTarget
{
public:
   CComObject();
```

```
    BEGIN_INTERFACE_PART(DataObject, IDataObject)
      STDMETHOD(GetData)(LPFORMATETC, LPSTGMEDIUM);
      // ...Other members
    END_INTERFACE_PART(DataObject)

    BEGIN_INTERFACE_PART(OleObject, IOleObject)
      STDMETHOD(SetClientSite)(LPOLECLIENTSITE);
      // ...Other members
    END_INTERFACE_PART(OleObject)

    DECLARE_INTERFACE_MAP()
};
```

The preprocessor will expand the macros into the following:

```
class XDataObject : public IDataObject
{
public:
   STDMETHOD_(ULONG, AddRef)();
   STDMETHOD_(ULONG, Release)();
   STDMETHOD(QueryInterface)(REFIID iid, LPVOID* ppvObj);
   STDMETHOD(GetData)(LPFORMATETC, LPSTGMEDIUM);
   // ...Other members
} m_xDataObject;
friend class XDataObject;
```

As you can see, the macros declare the **IUnknown** functions for the nested class. They also make the class a **friend** of the controlling class, so that the nested objects can see any **private** members of the controlling class. Remember that only **friend**s can see your **private** parts!

To get MFC's support for reference counting and interface lookup, you must derive your COM object class from the **CCmdTarget** class and use an **interface map**. Interface maps are similar in concept to message maps and dispatch maps, except they maintain the glue between an interface ID (IID) and the data member of a COM object which implements the specified interface.

CCmdTarget has built-in support for **IUnknown**. It contains functions for performing **AddRef()**, **Release()** and **QueryInterface()**. The **QueryInterface()** function built into **CCmdTarget** will use the interface map to locate the appropriate member for the interface's implementation. For example, if I wanted to use MFC's support for creating COM objects to implement the COM class specified above, I would create the class as follows:

```
class CComObject : public CCmdTarget
{
public:
   CComObject();

   BEGIN_INTERFACE_PART(DataObject, IDataObject)
     STDMETHOD(GetData)(LPFORMATETC, LPSTGMEDIUM);
     STDMETHOD(GetDataHere)(LPFORMATETC, LPSTGMEDIUM);
     STDMETHOD(QueryGetData)(LPFORMATETC);
     // ...Other members
   END_INTERFACE_PART(DataObject)

   BEGIN_INTERFACE_PART(OleObject, IOleObject)
     STDMETHOD(SetClientSite)(LPOLECLIENTSITE);
     STDMETHOD(GetClientSite)(LPOLECLIENTSITE*);
     STDMETHOD(SetHostNames)(LPCOLESTR, LPCOLESTR);
     // ...Other members
   END_INTERFACE_PART(OleObject)

   DECLARE_INTERFACE_MAP()
};
```

```
// This goes in the implementation (.cpp) file
BEGIN_INTERFACE_MAP(CComObject, CCmdTarget)
   INTERFACE_PART(CComObject, IID_IDataObject, DataObject)
   INTERFACE_PART(CComObject, IID_IOleObject,  OleObject)
END_INTERFACE_MAP()

STDMETHODIMP_(ULONG) CComObject::XDataObject::AddRef()
{
   METHOD_PROLOGUE(CComObject, DataObject)
   return (ULONG)pThis->ExternalAddRef();
}

STDMETHODIMP_(ULONG) CComObject::XDataObject::Release()
{
   METHOD_PROLOGUE(CComObject, DataObject)
   return (ULONG)pThis->ExternalRelease();
}

STDMETHODIMP CComObject::XDataObject::QueryInterface(
   REFIID iid, LPVOID* ppvObj)
{
   METHOD_PROLOGUE(CComObject, DataObject)
   return (HRESULT)pThis->ExternalQueryInterface(&iid, ppvObj);
}
```

In the implementation file, you must use the **BEGIN_INTERFACE_MAP()** and **END_INTERFACE_MAP()** macros to let the **CCmdTarget**-derived class know of the interfaces that the controlling object will support. In between these macros, entries are made for each interface. The entries are used by **CCmdTarget** to look up the interface pointer when a **QueryInterface()** call is made. For example, if a client application calls **QueryInterface()**, asking for **IID_IDataObject**, a pointer to the **m_xDataObject** member of the controlling object will be returned. You don't need to include an entry for **IID_IUnknown**, since MFC will return the first entry in the map. In our case, **IID_IDataObject** was placed as the first entry in the map, so when the client asks for **IID_IUnknown**, MFC will return **m_xDataObject**'s **this** pointer.

Although you don't need to declare **AddRef()**, **Release()** and **QueryInterface()** for the nested classes (since this is done automatically by the **BEGIN_INTERFACE_PART()** macro), you do need to provide an implementation for them. This is very easy, since all you have to do is retrieve a pointer to the controlling object and call its **IUnknown** members:

```
STDMETHODIMP_(ULONG) CComObject::XDataObject::AddRef()
{
   METHOD_PROLOGUE(CComObject, DataObject)
   return (ULONG)pThis->ExternalAddRef();
}
```

This looks a bit different from the way we did it without MFC support, but the principle is the same. **METHOD_PROLOGUE()** simply creates a local variable, called **pThis**, which contains a pointer to the main object (equivalent to **m_pParent** in the earlier example). **ExternalAddRef()**, **ExternalRelease()** and **ExternalQueryInterface()** are the **CCmdTarget** functions which implement the standard **IUnknown** functions **AddRef()**, **Release()** and **QueryInterface()**. (We'll see why they have **External** in their name in the section on aggregation later in this chapter.)

When you use the **BEGIN_INTERFACE_MAP()** macro, as well as specifying the controlling COM class, you must specify the base class. This allows you to derive from an existing MFC/OLE class and add more functionality to the class with new interfaces.

In other words, let's say I wanted to use the functionality of the MFC **CDataSource** class (which already implements the **IDataObject** interface) from my own class which I plan to expose as a COM class. I could derive my class from **CDataSource**, implement the interfaces I wish to add (such as

IPersistStorage or IOleObject) in my new class, and expose the class with its own CLSID. Then, any client can simply create an instance of my COM class and access all the interfaces (including the ones in CDataSource).

You can't, however, derive from one of the interfaces implemented by an MFC class. In other words, I can't derive a class from the nested XDataObject class implemented within the CDataSource class.

Instantiating a COM Object

The question is, how do you create a COM object and get a pointer to an interface on it in the first place? The answer is usually to call CoCreateInstance(), which (if all goes well) will create the object you specify and return you a pointer to the interface that you require.

To access an object via CoCreateInstance(), you'll need a globally unique identifier (GUID) for the class of object that you want OLE to create for you. The OLE libraries will then use that GUID to look in the registry and find out more information about the class (if it exists), such as the location of the server executable or DLL that contains the code which actually creates the object (and therefore, the interfaces). The GUID that identifies an object class is called a **class identifier** (or **CLSID** for short).

Just as we have GUIDs for object classes, we also have GUIDs for the interfaces, called **interface identifiers** (**IID**). IIDs are used to identify the interfaces that we request from a COM object.

There are several ways to create a GUID for your own custom COM classes and interfaces. One is to use the Guidgen.exe application which is shipped with the Win32 SDK, as well as VC++. Another way to generate them is to use the OLE API function CoCreateGUID().The last is to call Microsoft and request 256 unique GUIDs. Because GUIDs use a great algorithm and a 128-bit structure to store the identifier, they're pretty much guaranteed to always be unique.

Using GUIDs and Class Factories

Fortunately, when you use ClassWizard to generate a class derived from CCmdTarget, you have the option of telling ClassWizard that the class should be OLE Automation Creatable by type ID which means ClassWizard will generate (among other things) two lines of code that look like this:

```
// In the header file:
DECLARE_OLECREATE(CComObject)

// In the implementation file:
// {15F3C485-30D8-11CF-97E6-444553540000}
IMPLEMENT_OLECREATE(CComObject, "COMOBJECT", 0x15f3c485, 0x30d8, 0x11cf, 0x97, 0xe6, 0x44,
0x45, 0x53, 0x54, 0x0, 0x0)
```

These two lines will ensure that the object has a **class factory** and that objects can be created from the object class. Class factories are what actually create the instances of the OLE objects provided by the server that OLE loads (when CoCreateInstance() is called by the client).

When an application calls OLE to get an interface on a newly created data object, OLE finds and loads the server by using the information stored in the registry for the server. This information can be registered in several ways. One way is to call COleObjectFactory::UpdateRegistryAll() to register all the class factories. UpdateRegistryAll() is a static function within that class which updates the registry with the appropriate information including the program ID, class ID and the full path to the in-process server.

COleObjectFactory::UpdateRegistryAll() is usually executed as part of the server application's **InitInstance()** function if the server can also be run as a stand-alone executable. If it's an in-process server, it will usually export a function, called **DllRegisterServer()**, that calls **COleObjectFactory::UpdateRegistryAll()**. This function can be called by `Regsvr32.exe`, an application that takes the path to an in-process server and calls **DllRegisterServer()** on it. You can find `Regsvr32.exe` in your `\Msdev\Bin` directory.

Servers can also be registered, via a registration (`.reg`) file which can be merged with the registry by double-clicking on it. **.reg** files are simple text files. The following is an example of a registration file (this file is used in the aggregation example later in this chapter):

```
REGEDIT4

[HKEY_CLASSES_ROOT\CLSID\{591E69E7-63F0-11CF-B337-444553540000}]
@="Data Object Server"
[HKEY_CLASSES_ROOT\CLSID\{591E69E7-63F0-11CF-B337-444553540000}\InprocServer32]
@="c:\\msdev\\projects\\aggregation\\dataserv\\debug\\dataserv.exe"
[HKEY_CLASSES_ROOT\CLSID\{591E69E7-63F0-11CF-B337-444553540000}\ProgID]
@="DATASERV.DATAOBJ"

[HKEY_CLASSES_ROOT\STORSERV.STORAGEOBJ\CLSID]
@="{591E69E7-63F0-11CF-B337-444553540000}"
```

Aggregation

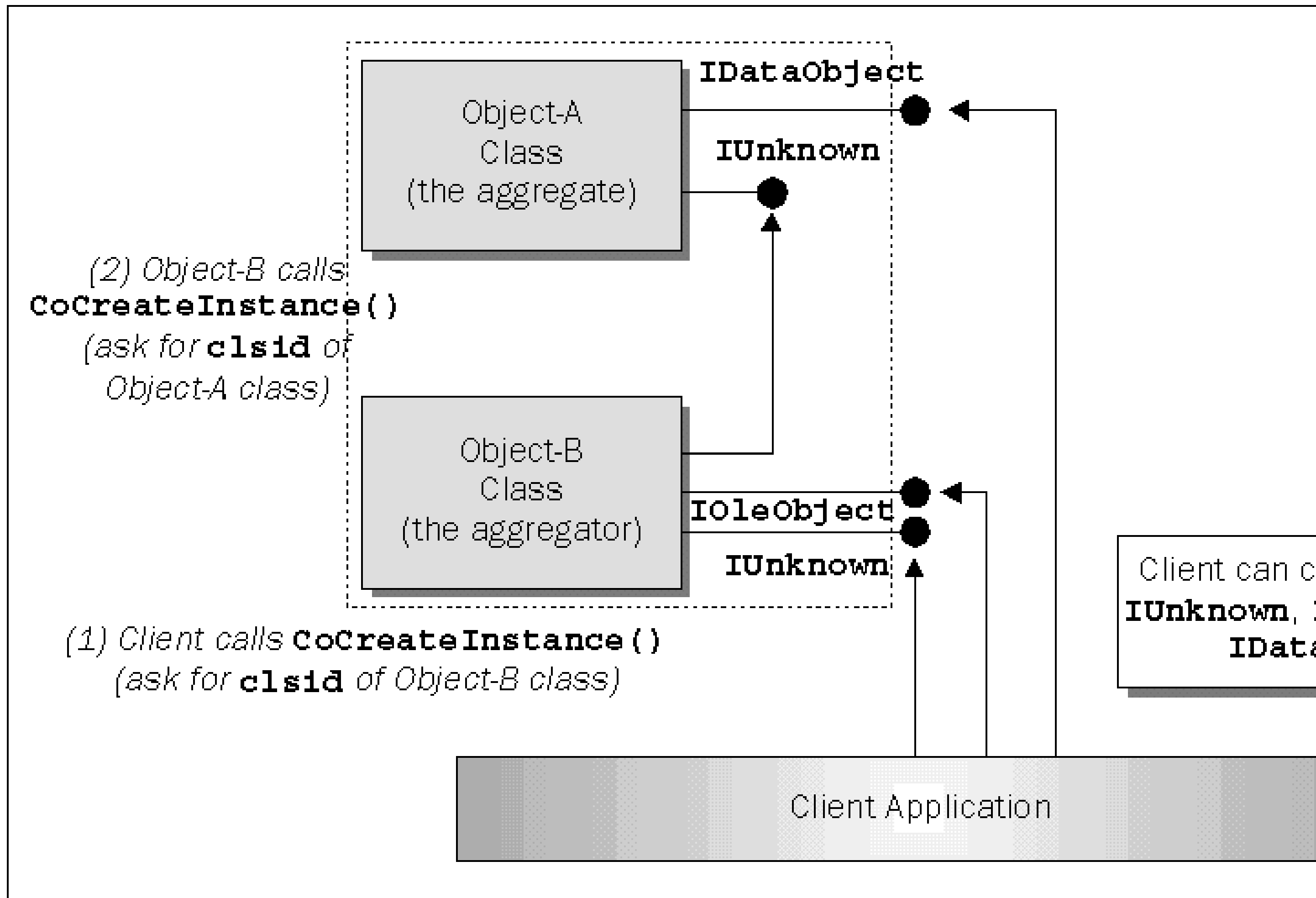
You can see that using `CCmdTarget` and an interface map frees you from the tedium (and potential mistakes) of implementing your own reference counting and interface lookup functions. You can simply delegate all the work to the functions provided by `CCmdTarget`. This becomes even more useful when you consider **aggregation**, a much misunderstood technique.

If you've ever wished that you could use the functionality of some object as if it were part of your own object then you have already started to dream about aggregation. In short, aggregation is the ability to use the implementation of another object as if it were your own, and what's more, it's fast.

You're probably already familiar with a related technique—**containment**. With containment, you add functions to your class' interface that mimic the functions of another object. The implementations of these functions simply call the corresponding functions on a contained member of the other object. This technique gives you exact control over which functions you expose and how they can be called, but it's slow in terms of execution speed (an extra function call is made for each function in the contained object) and it's also laborious to duplicate all the functions of the other object if it's rich and complex.

Aggregation solves both of these problems by providing clients with direct access to entire interfaces of the object you wish to use. You can select which interfaces are available to the client, but once they have a pointer to that interface, they can call the functions on it directly, as if the client had created the object themselves. Thus, aggregation is faster than containment and, if you don't need to limit the functionality of the aggregated object too much, it's a better solution. It does, however, require that the object you wish to exploit was written ready to be aggregated. We'll see why as we examine exactly how aggregation works.

Let's say that your department is going to create an application and management wants you to use a great COM object (which we'll call Object-A) developed in another department for which you don't have any source code. Object-A contains an implementation of `IDataObject`, but has no support for becoming an embeddable OLE document. This is where aggregation comes in. By creating the code for another COM object (Object-B) that implements all of the other necessary interfaces, you can use and expose the `IDataObject` interface from Object-A and the `IOleObject` interface from Object-B as a single COM object class. Then, all the client application has to do is create an instance of Object-B (which in turn creates an instance of Object-A, although this is invisible to the client).



When you're implementing aggregation, there are always two sides to the story. There's the object that's being aggregated (the **aggregate**, Object-A in our case) and there's the object that is aggregating other objects (the **aggregator**, Object-B in our case). The aggregator's `IUnknown` is referred to as the **outer unknown** because this is the only `IUnknown` that is available to the client. As far as the client is concerned, the `IUnknown`s of any aggregated objects are hidden inside the virtual object represented by the dashed line in the diagram. It doesn't know that more than one object is involved.

For an object to become an aggregate, it must adhere to a number of rules, all of which are designed to ensure that the three following principles are implemented satisfactorily:

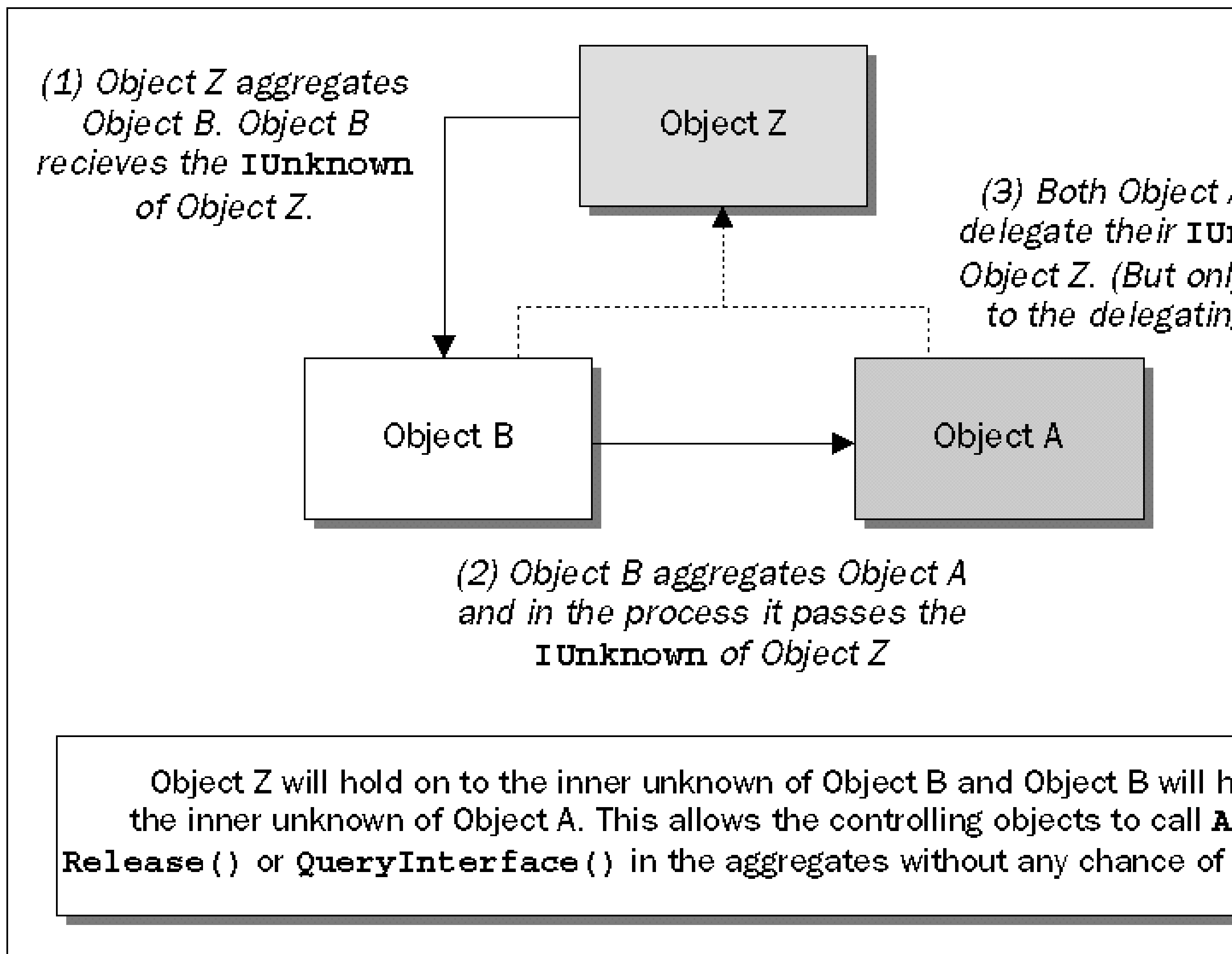
- The use of aggregation should be transparent to the client.
- The aggregator has complete control over the lifetime of the aggregate.
- The aggregator has complete control over which of the aggregate's interfaces are exposed to the client.

The first and foremost rule in aggregation is that the aggregate must let the aggregator handle any calls to `AddRef()`, `Release()` or `QueryInterface()` made on its exposed interfaces. Clearly, this makes sense when we consider the second and third of the principles above; `AddRef()` and `Release()` control an object's lifetime and `QueryInterface()` controls an object's functionality, so the aggregator must handle these functions. It makes even more sense that only the aggregator can handle calls to `QueryInterface()`

when you realize that an aggregated object has no way of knowing what interfaces the aggregator might expose and, therefore, no way of returning pointers to them.

An aggregate simply **delegates** any calls made to these functions to the `IUnknown` interface of the aggregator. This is known as the **controlling** or **outer unknown**. The aggregate will be passed a pointer to the outer unknown when it is created via `IClassFactory::CreateInstance()` or `CoCreateInstance()`. In fact, it's the presence of this pointer (in a parameter that would otherwise be `NULL)`,that tells the object that it is being aggregated and allows it to change its behavior accordingly.

Since aggregation is transparent to all clients, an aggregator might itself be aggregated, whether knowingly or unknowingly. There's no limit to the levels of aggregation that can take place. There's no easy way to tell whether an object that is about to be aggregated already uses aggregation, and, in fact, there is no need to know. All aggregated objects must delegate their `AddRef()`, `QueryInterface()` and `Release()` functions to a single `IUnknown` to ensure that there's only a single object to control the functionality and lifetime of the object. This is handled by requiring all objects to pass their outer unknowns to the creation function. It's only the outermost object that will be passing its own `IUnknown` pointer to any objects it aggregates. You can see this situation in the figure below:



You can see that the lifetime and capabilities of the object as a whole are controlled from a single object and that the client has no idea about whether aggregation is being used at all.

Of course, the aggregator must have a way to **`AddRef()`**, **`Release()`** and **`QueryInterface()`** the aggregate without it delegating back to the outer unknown, otherwise there would be an endless loop. This problem is solved by simply ensuring that these functions on the **`IUnknown`** interface never delegate, whereas the same functions on any other interface do. The aggregator must ask for the **`IUnknown`** interface when it creates an aggregate so that it can control its lifetime and functionality. In fact, if an aggregator attempts to pass an outer unknown pointer when it creates an object, but fails to request the **`IUnknown`** interface, the class factory responsible should fail to return a valid pointer and should return an error instead.

In addition, the aggregate's class-factory should ensure that the first request for an interface doesn't delegate an **`AddRef()`** call to the aggregator. Lucky for you, MFC already implements all of this work on behalf of the class factory.

Aggregates in MFC

With MFC's help, creating an object that can be aggregated is easy.
If you're already using a **CCmdTarget**-derived class and the class factory supplied by **DECLARE_OLECREATE()**, all you have to do is call **EnableAggregation()** from the object's constructor. MFC will take care of the rest, including, as we've just mentioned, returning the correct pointer on object creation and storing the outer **IUnknown** pointer to which calls must be delegated.

Now, maybe it's becoming apparent why the functions we called from the **IUnknown** functions in our interfaces were called **ExternalQueryInterface()**, **ExternalAddRef()**, and **ExternalRelease()**. It just so happens that these functions delegate to the outer unknown if the object is being aggregated, otherwise they just call the inner **IUnknown** we considered before.

There are also internal versions of these functions called **InternalAddRef()**, **InternalRelease()**, and **InternalQueryInterface()**. **InternalAddRef()** and **InternalRelease()** work on the inner unknown, regardless of whether or not the object is being aggregated. The interface lookup in **InternalQueryInterface()** is handled internally with no delegation, but it still calls **ExternalAddRef()** if an interface is found, causing the outer unknown's reference count to be incremented. You can, of course, call these functions directly if you need to be sure that your requests aren't delegated, but pretty much all of the time you're going to want to call the **External** versions

Now you can see how MFC gives you aggregates for free, let's see how it makes your life easier when you want to create an aggregator.

Aggregators in MFC

The first thing you'll need to do if you want to create an aggregator is to create some data members in the controlling class of the aggregator for each **IUnknown** pointer that you expect to get back from the aggregates. You'll also need to initialize these members to **NULL** in the aggregator's constructor.

As we've already mentioned, you need to call **CoCreateInstance()** to create the aggregates, passing it the outer unknown pointer for your object and the data member that will be used to hold the returned **IUnknown** pointer. When you call **CoCreateInstance()**, you should pass it the value returned from **CCmdTarget::GetControllingUnknown()**, in case the aggregator is also being aggregated by another object. By calling **GetControllingUnknown()**, you allow MFC to pass the appropriate **IUnknown**. If the aggregator is not being aggregated, the function will return the **this** pointer of the controlling object, otherwise it will return the **IUnknown** of the aggregating object.

To make your life particularly easy, **CCmdTarget** possesses a function called **OnCreateAggregates()** where you should place your creation code. This is called by the framework when the aggregator is first created.

With the aggregates created, the only thing left is to tie the interfaces in the aggregate into your object's **QueryInterface()** function. Since all it took to hook our own object's interfaces up to **QueryInterface()** was an entry in the interface map, it would be great if we could do the same for our aggregates interfaces. This would allow the aggregator's **QueryInterface()** to check the interface map and call the aggregate's inner unknown if it needs to.

In fact, we can use MFC's **INTERFACE_AGGREGATE()** macro to tell the **CCmdTarget**-derived class about any objects that we're aggregating. This macro needs to be placed in between the **BEGIN_INTERFACE_MAP()** and **END_INTERFACE_MAP()** macros after any **INTERFACE_PART** entries. You must add an entry for each object that you're aggregating. The **INTERFACE_AGGREGATE()** macro requires that you pass it the name of the **CCmdTarget**-derived class, and a pointer to the inner **IUnknown** of the aggregate.

Your source code will look something like this:

```
CComObject::CComObject()
{
   m_lpAggregate = NULL;
}

BEGIN_INTERFACE_MAP(CComObject, CCmdTarget)
   INTERFACE_PART(CComObject, IID_IDataObject, DataObject)
   INTERFACE_PART(CComObject, IID_IOleObject, OleObject)
   INTERFACE_AGGREGATE(CComObject, m_lpAggregate)
END_INTERFACE_MAP()

BOOL CComObject::OnCreateAggregates()
{
   HRESULT hr = ::CoCreateInstance(CLSID_ObjectA,
       GetControllingUnknown(), CLSCTX_ALL, IID_IUnknown, &m_lpAggregate);

   if (FAILED(hr))
      return FALSE;

   return TRUE;
}
```

This lets **CCmdTarget** search through the map for any requested interfaces. If the interface isn't provided in the controlling object, **CCmdTarget** will pass the request on to the inner unknown of any aggregates in the list, searching through them until the interface is queried successfully or the list is exhausted.

This is great if you want to allow any interface request not handled by your own interfaces to be passed on to an aggregate, but, if you want to be more selective, there's another way. When an MFC COM object's **QueryInterface()** function is called, it will give you first crack at determining the interface pointer to return, since it calls **CCmdTarget::GetInterfaceHook()** before using the interface map. If you override this function, you have the option of returning an interface to an aggregate yourself.

Note that if you override **GetInterfaceHook()**, you shouldn't place the **IUnknown** of the aggregate for which you want to expose a limited set of interfaces in the interface map. In other words, don't call **INTERFACE_AGGREGATE()** for the aggregate.

```
LPUNKNOWN CComObject::GetInterfaceHook(const void* iid)
{
   HRESULT hr;
   LPVOID  lpInterface;

   // Allow only one interface from the aggregate.
   if (*(IID*)iid == IID_ISomeInterface)
      hr = m_lpAggregate->QueryInterface(*(IID*)iid, &lpInterface);
   else
      return NULL;

   return FAILED(hr) ? NULL : lpInterface;
}
```

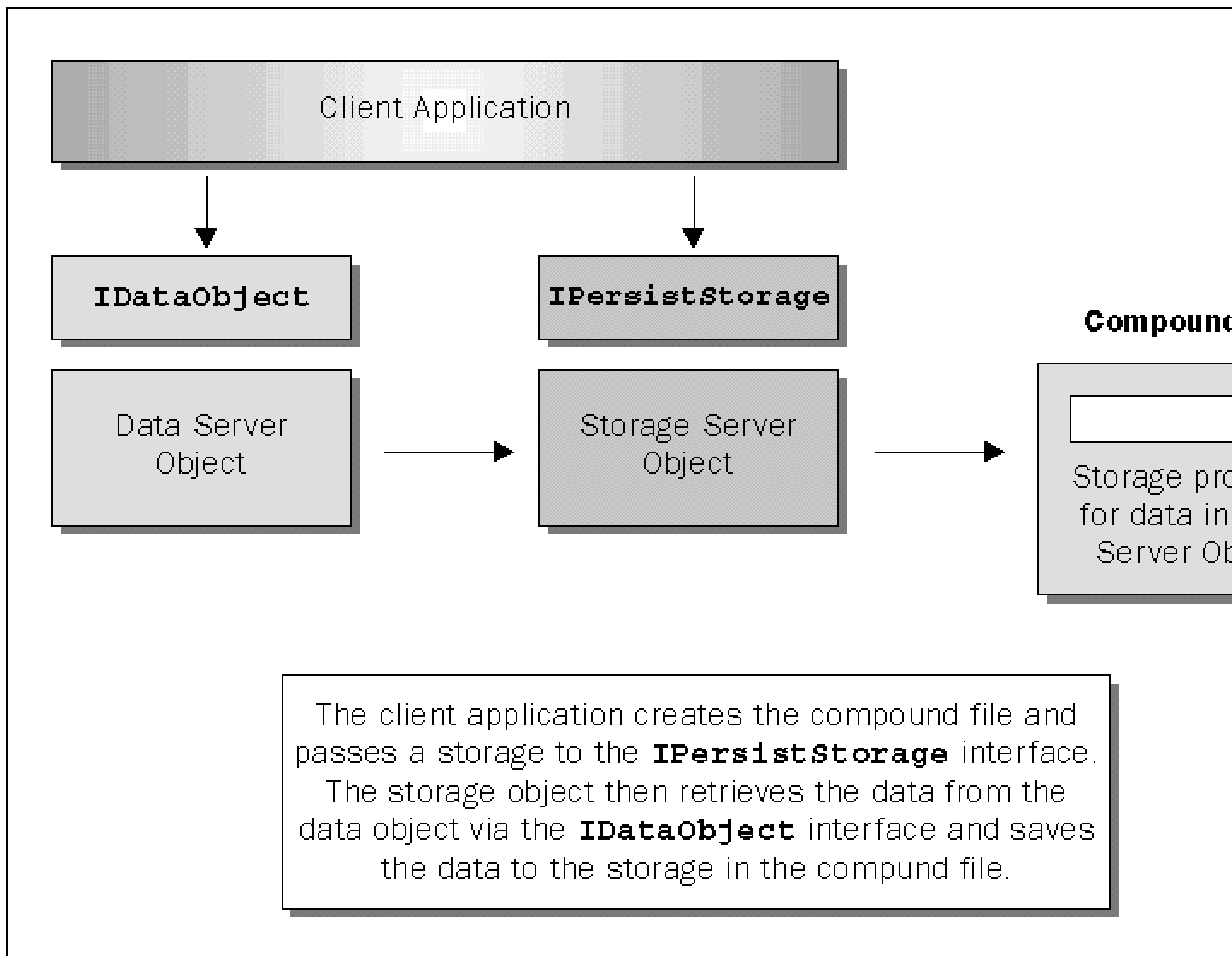
Aggregation through Example

Let's wrap this section up with an example. I'll demonstrate aggregation by implementing a client and two servers with a COM object in each. We'll call the servers the *data server* (since it will expose an **`IDataObject`** interface) and the *storage server* (since it exposes **`IPersistStorage`**).

The data server has its own user interface in the form of a dialog box and is implemented as an in-process server (DLL), so it can't run stand-alone. Although the data server can provide data, it has no support for file I/O and, therefore, cannot be told to save its data persistently. The storage server has been designed to handle this task.

The storage server has been implemented as a local server (`.exe`) and its objects act as aggregators to objects in the data server. The storage server adds the ability to store the data kept in the data server by implementing the **`IPersistStorage`** interface in its objects.

The client application will create an object from the storage server (and implicitly one from the data server as well) and will provide a compound file for the object to store its data. Although the data server will provide the data, the actual storage of the data will be handled by the storage server.

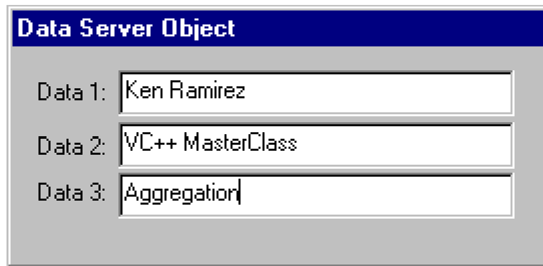


Don't worry too much about what the different interfaces do and how the persistent storage works as we'll cover that through the course of this chapter. For the moment, just accept this as an example of aggregation.

The Data Server

The data server implements COM objects called data objects. Data objects provide two interfaces: **`IUnknown`** and **`IDataObject`**. A client can ask for either one of these interfaces via **`CoCreateInstance()`**, **`IClassFactory::CreateInstance()`**, or **`IUnknown::QueryInterface()`**. However, as we've seen, if the object is being aggregated, the aggregator must always ask for **`IUnknown`**.

Internally, the data object creates a dialog box and displays it. This dialog box gathers information from the user via three edit controls. You can see the dialog box below:



The information is then returned to a client when the client calls `IDataObject::GetData()`, which can return the data in one of three different formats: metafile, bitmap or text. Later, when we meet the storage server and the client, you'll see that the client will ask for the data as a bitmap to display it within its view, and the storage server will ask for the data as text to save it into an `IStorage` provided by the client.

Besides gathering the information from the edit controls when the user types new information, the dialog box class that I create in the data server also has the responsibility of rendering the data in all of the different formats when a client calls `IDataObject::GetData()`, and displaying the data in the edit controls when a client sends data to the data object via the `IDataObject::SetData()` function (`IDataObject` is a standard Microsoft interface that we'll cover later in this chapter).

There are three functions which render the data: `RenderBitmap()`, `RenderMetafilePict()` and `RenderNative()`. `RenderNative()` returns the data as one block of memory containing the data for all three strings obtained from the edit controls. The strings are null-terminated within the memory block so that they can be separated later on (if necessary).

The `SetData()` function within the dialog box class takes a block of memory (again, the strings are null-terminated within the block of memory) and rips it apart into three individual strings. The strings are then placed into the edit controls for the user to see.

These functions are called from the `IDataObject` implementation of the data object. The details of `IDataObject` are discussed later in this chapter (see the section on Uniform Data Transfer for more information).

As I said before, when an application calls OLE to get an interface on a newly created data object, OLE finds and loads the server by using the information stored in the registry for the server. I provided three methods to register the server. You can either use `Regsvr32.exe` (as explained above) or the registration file that I provided, called `Dataserv.reg` (which was also shown above). Alternatively, the server will be registered by a custom build step when you build it.

The only aggregation-specific item in the data server is the call to `EnableAggregation()` in the `CCmdTarget`-derived class' constructor. Everything else is handled by `CCmdTarget`.

Any client that wants to create a data server object needs only to call `CoCreateInstance()` and pass it the CLSID of `CLSID_DataObject`. I placed this CLSID, along with the storage object's CLSID, in a file called `Aggrguid.h`. The client can then ask for a pointer to the `IDataObject` interface and start to call members to set or get data from the object.

The Storage Server

The storage server implements objects called **storage objects**. These play the role of the aggregator and they each create an instance of a data object for aggregation. I implemented the storage server as a local server (`.exe`) because I wanted to show two ways of implementing COM objects, within an in-process server and a local server.

When you're writing your own servers, you might be faced with the decision of determining where to place your code. Do you place it in an in-process server or in a local server? There are several factors that you'll have to consider before making that decision. For example, does your server ever have to run as a stand-alone application (in which case, it has to be a local server), or will it always run on behalf of a client (in which case, you can make it an in-process server). In-process servers execute and communicate faster with clients because they need no marshaling support, but, if you have to write the server as a local server, you'll have to provide the marshaling support yourself (which is not the easiest or the most interesting thing to do).

This server can be registered using a registration file or by running the local server briefly to allow it to register itself. Note that if you choose to run the server, MFC adds an entry to the registry that causes the server to have an in-process handler and this entry causes problems later when the client attempts to create an instance of the server. Therefore, you should remove the **`InprocHandler32`** entry from the registry for the server. I'd choose to save myself the headaches and use the registration file. The registration file for the storage server is called `Storserv.reg`, and here's what it looks like:

```
REGEDIT4

[HKEY_CLASSES_ROOT\CLSID\{246135C9-67F6-11CF-B337-444553540000}]
@="Storage Object Server"
[HKEY_CLASSES_ROOT\CLSID\{246135C9-67F6-11CF-B337-444553540000}\LocalServer32]
@="c:\\msdev\\projects\\aggregation\\storserv\\debug\\storserv.exe"
[HKEY_CLASSES_ROOT\CLSID\{246135C9-67F6-11CF-B337-444553540000}\ProgID]
@="STORSERV.STORAGEOBJ"

[HKEY_CLASSES_ROOT\STORSERV.STORAGEOBJ\CLSID]
@="{246135C9-67F6-11CF-B337-444553540000}"
```

A storage object implements the **`IPersistStorage`** interface (which will be discussed in greater detail later in this chapter). It also creates an aggregate object from the data server for the sole purpose of exposing the data object's interfaces as if they were implemented by the storage object. When a client application creates a storage object, the application can also ask for an interface pointer for the **`IDataObject`** interface. Even though the storage object doesn't implement the interface, it can safely return it since it acts as the aggregator to an object that *does* implement the **`IDataObject`** interface.

How did I manage this wizardry? Easy—I used MFC's macros and classes to my advantage, to implement and expose the **`IPersistStorage`** interface. First of all, I declared a data member, **`m_lpAggregate`**, in the class which would hold the inner **`IUnknown`** pointer returned from the aggregate's class factory when the object is initially created. I also knew that I would be calling members of the **`IDataObject`** interface when the storage object is told to load or save its data via the **`IPersistStorage`** interface. I declared a data member to hold on to the **`IDataObject`** pointer which is immediately retrieved from the data object once it's created and its **`IUnknown`** pointer has been received.

Here's an extract of the class definition for the data object:

```
class CStorageServObj : public CCmdTarget
{
// .
// . Other pieces of code are not shown here for simplicity.
// .
```

```
// Overrides
public:
   virtual void OnFinalRelease();
   BOOL OnCreateAggregates();

// Implementation
protected:
   LPUNKNOWN      m_lpAggregate;
   LPDATAOBJECT   m_lpDataObject;

// .
// . More code
// .
};
```

Note that the members are initialized to **NULL** in the class's constructor to prevent premature usage of the **m_lpAggregate** data member within MFC.

To make MFC aware that there's an aggregate object whose **QueryInterface()** should be called for any interface request, not implemented within the storage object, I used an **INTERFACE_AGGREGATE()** macro within the **BEGIN_INTERFACE_MAP()** and **END_INTERFACE_MAP()** macros in the implementation file:

```
BEGIN_INTERFACE_MAP(CStorageServObj, CCmdTarget)
   .
   . Other Interface entries
   .
   INTERFACE_AGGREGATE(CStorageServObj, m_lpAggregate)
END_INTERFACE_MAP()
```

The next thing is to provide the override for the **CCmdTarget::OnCreateAggregates()** function where I create the aggregate and retrieve its **IDataObject** interface:

```
BOOL CStorageServObj::OnCreateAggregates()
{
   HRESULT hr;

   hr = ::CoCreateInstance(CLSID_DataObject, GetControllingUnknown(),
      CLSCTX_INPROC_SERVER, IID_IUnknown, (LPVOID*)&m_lpAggregate);

   hr = m_lpAggregate->QueryInterface(IID_IDataObject,
  (LPVOID*)&m_lpDataObject);

   m_dwRef--;

   return TRUE;
}
```

By this point, you should be familiar with the elements of the **CoCreateInstance()** call. To let the aggregate know of its aggregator, the storage object must pass its outer **IUnknown** pointer to **CoCreateInstance()**. As I mentioned earlier in the chapter, this is easily done by calling **CCmdTarget::GetControllingUnknown().**

The reason for decreasing the reference count of the aggregator has to do with the **QueryInterface()** call to the data object's inner unknown. When the storage object calls the aggregate's **QueryInterface()** asking for the **IDataObject** interface, it will delegate its **AddRef()** (which occurs if the interface is found) to the aggregator (which is the storage object). This will happen even though the aggregator is calling the inner unknown. As a result, we need to make sure that we reduce the reference count by one to make up for the extra count on storage object. If we don't, the object will live forever.

Finally, we have one more function to discuss: **OnFinalRelease()**. In the data server, this function was

no big deal because all it did was call the base class' implementation, which simply called **delete** on the object's **this** pointer. However, in the storage server, this function plays more of a role. Here is the implementation:

```
void CStorageServObj::OnFinalRelease()
{
   // When the last reference for an object is released
   // OnFinalRelease is called.  The base class will automatically
   // delete the object.

   // Before calling the IDataObject::Release(), we need to pump up our
   // own count, since the aggregate will delegate a Release() to us.
   // We need to pump by two, because if we added only one, we'd end
   // in this function again, and again, and again.  Well, you get
   // the idea.
   m_dwRef += 2;

   // Release the interface and set the pointer to NULL.
   RELEASE_INTERFACE(m_lpDataObject);
   RELEASE_INTERFACE(m_lpAggregate);

   // .
   // . Other code here
   // .

   CCmdTarget::OnFinalRelease();
}
```

OnFinalRelease() is called when the reference count of the object has reached zero. Within the implementation of **OnFinalRelease()**, I'd like to release the pointers to the inner **IUnknown** and the **IDataObject** interfaces of the aggregate. The problem is that the aggregate will delegate its **IDataObject::Release()** call to its outer unknown, which is the aggregator's **IUnknown** implementation, causing **OnFinalRelease()** to be called over and over and over (well, you get the point, I hope).

We can't avoid having the aggregator's **Release()** function called, but we can prevent the infinite calls to **OnFinalRelease()** by increasing the reference count back up to two. The reason I set it to two, and not to one is because if we only set it to one, the reference count would be decreased to zero again, causing the **OnFinalRelease()** function to be called as well.

Although the storage object's purpose is to provide an **IPersistStorage** implementation, the discussion of this interface is deferred until later in this chapter (see the section on Structured Storage and Compound Files).

The Client

The client plays the easiest role of all, simply using the services of the two servers. It never knows that it's communicating with two servers. It creates an instance of a COM class it thinks supports three interfaces (**IUnknown**, **IDataObject** and **IPersistStorage**) by calling **CoCreateInstance()** and asking for an instance of **CLSID_StorageObject**:

```
HRESULT hr = ::CoCreateInstance(
   CLSID_StorageObject,      // Class identifier
   NULL,                     // We aren't creating an aggregate
   CLSCTX_LOCAL_SERVER,      // Context for running executable code
   IID_IUnknown,             // Ask for IUnknown
   (LPVOID*)&m_lpUnknown);   // Store interface pointer here.
```

Once the first interface pointer is retrieved, the client can then ask for **IDataObject** or **IPersistStorage**.

The following is a description of the steps that occur in order to obtain the appropriate interface pointer.

Achieving an `IPersistStorage` from the initial `IUnknown`:

1. The client calls `QueryInterface()` asking for the `IPersistStorage` interface on the `IUnknown` pointer it obtained from `CoCreateInstance()`.
2. The aggregator checks its interface map for a match and finds it.
3. The client receives the `IPersistStorage` interface from the storage object (which is the object it originally created).

Achieving an `IDataObject` from the initial `IUnknown:`

1. The client calls `QueryInterface()`, asking for the `IDataObject` interface on the `IUnknown` pointer it obtained from `CoCreateInstance()`.
2. The aggregator checks its interface map for a match and doesn't find it.
3. Since the storage object (which is also the aggregator) doesn't implement `IDataObject`, MFC next checks the aggregate in the map and calls its `QueryInterface()` function.
4. The aggregate checks its own interface map and finds the interface to return.
5. The aggregate then calls the `AddRef()` function of its controlling unknown (on the aggregator), increasing the reference count in the storage object, not the data object.
6. The client receives the `IDataObject` interface.

Achieving an `IPersistStorage` from the `IDataObject`:

1. The client calls `QueryInterface()` on the `IDataObject` pointer asking for the `IPersistStorage` interface.
2. Since the `IDataObject` interface is implemented by the data object (which is the aggregate), the function will delegate to the outer unknown (the `IUnknown` of the aggregator).
3. The `QueryInterface()` call will end up in the storage object's controlling unknown where it will find the interface in the interface map.
4. The client receives the `IPersistStorage` interface from the storage object (even though the `QueryInterface()` was called on the `IDataObject` interface which lives in the data object).

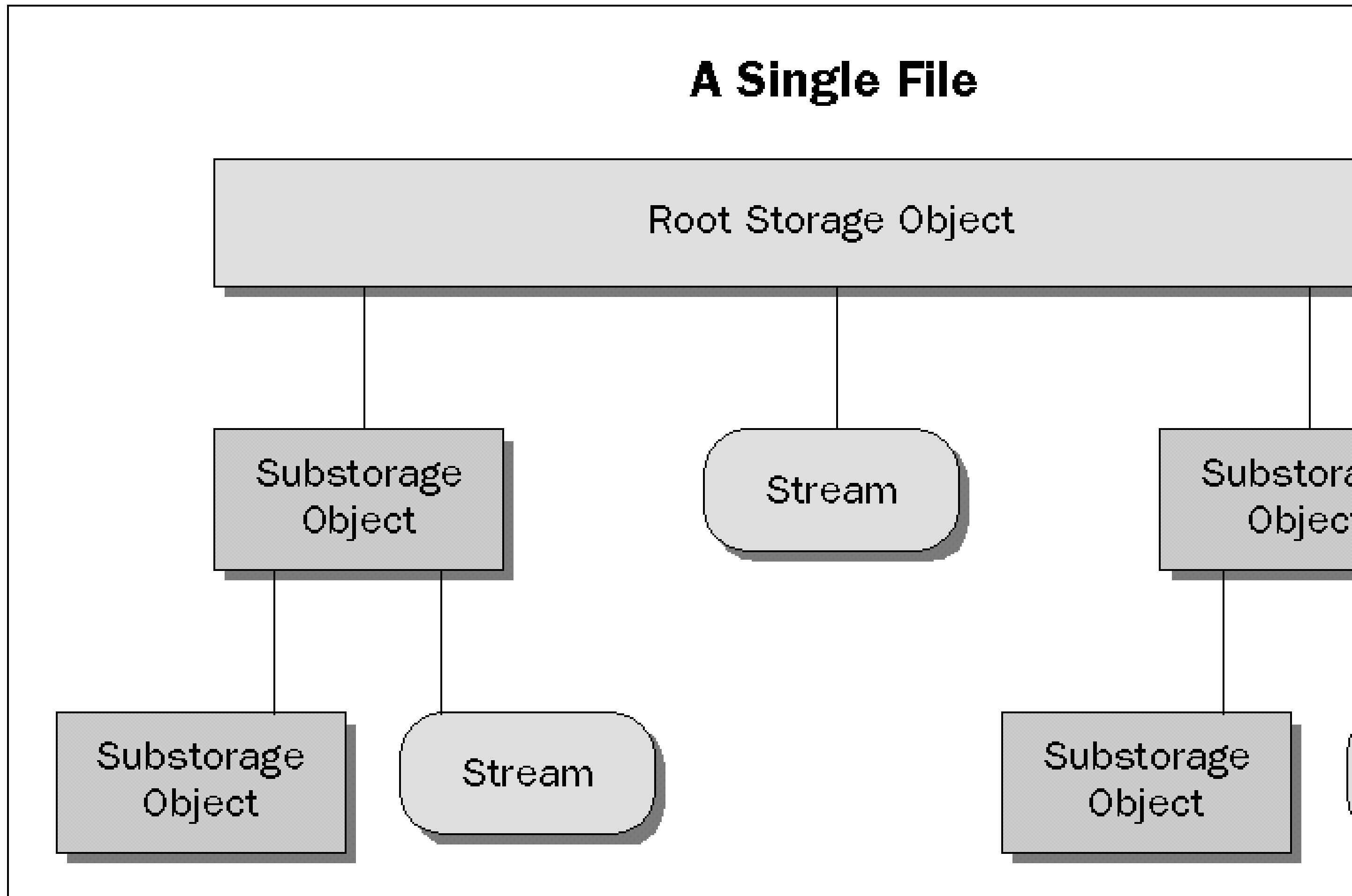
Achieving an `IDataObject` from the `IPersistStorage:`

1. The client calls `QueryInterface()` on the `IPersistStorage` pointer asking for the `IDataObject` interface.
2. The storage object checks its interface map for a match and doesn't find it.
3. Since the storage object (which is also the aggregator) doesn't implement `IDataObject`, MFC next checks the aggregate in the map and calls its `QueryInterface()` function.
4. The aggregate checks its message map and finds the interface to return.
5. The aggregate then calls the `AddRef()` function of its controlling unknown (on the aggregator). Therefore, the reference count is incremented in the storage object, not the data object.
6. The client receives the `IDataObject` interface on the data object (which is the aggregate).

There's not much more to this fascinating story except for the actual details concerning the usage and implementation of `IDataObject`, `IPersistStorage`, and compound documents. We'll come back to these details in the sections to follow.

Structured Storage and Compound Files

There is an OLE technology, known as **Structured Storage**, which is used to persistently save objects and data to a file with a hierarchical structure much like that of your file system. This technology uses a system of **storages** and **streams**. Storages are comparable to the directories of a file system; they can hold substorages and streams of data. Streams are comparable to the files of your file system; they can hold data, with the structure of that data determined by the application. Here you can see this 'file system within a file' concept:



The primary implementation of structured storage (provided by Microsoft) is called **Compound Files**. The idea is to replicate the structure of a file system within a single file. Note that Compound Files is just one possible implementation of structured storage. You could simply implement the necessary interfaces yourself to provide your own implementation of structured storage.

The use of Compound Files is most widely associated with OLE Documents. It's ideal since it allows each part of a compound document made up of linked or embedded objects to store itself in a single file. The document's container passes each of the embedded or linked objects a storage in the file in which to store itself via an interface called `IPersistStorage`.

If Compound Files only allowed the structured storage of information in a single file, it would be pretty good, but, in fact, it does much more than that, which makes it pretty awesome. First of all, it allows incremental saving of data (which increases performance), it allows several applications to share the same files (for both reading and writing), and it allows buffering of data (or transaction support). With transaction support, we could write several changes to a buffer which are not actually written to disk until we commit the changes.

We'll discuss these topics a bit further in this section, but first we'll discuss how we open and access a compound file. We'll then take a look at the interfaces used for dealing with structured storage and compound files. Finally, we'll discuss the interfaces that we can implement in our component objects to make them able to save themselves to a storage.

Using Structured Storage

You can create or open a compound file by using the OLE API functions **`StgCreateDocFile()`** and **`StgOpenStorage()`** respectively. There's also a function, called **`StgIsStorageFile()`**, which allows you to determine whether a file is in fact a compound file before you open it as one.

When you create or open a compound file, the system gives you back a pointer to an interface, called **`IStorage`**. **`IStorage`** contains the necessary methods for accessing or creating other substorages or streams.

Streams use an interface called **`IStream`** that allows you to read or write data to and from a stream. With all of this in mind, you can start to see how the embedded OLE objects in a compound document save themselves. The container simply passes an **`IStorage`** interface pointer to the objects and allows them to do their work inside of the storages.

The topmost storage is known as the **root storage** and it allows you to access another interface on it, known as the **`IRootStorage`** interface. This interface's only purpose is to allow you to switch the underlying file to another file in your file system. The contents of the storage object, and everything under it, are simply copied to the newly named file. The filename can't already exist. This is a great feature to use under low-memory conditions where no memory can be used to save any changes. Sometimes, when you attempt to save the data for a stream, you can get a low memory error. This is the perfect time to make use of the **`IRootStorage`** interface.

When you use structured storage, you'll find there's an interface which **`IStorage`** and **`IStream`** make use of to communicate with the underlying storage facility (whether it's a disk files, internal memory, or a database). This interface is called **`ILockBytes`**, and is an implementation of a byte array which treats data as a series of bytes. This interface is used primarily by the root storage to abstract the management of the underlying storage.

IStream and COleStreamFile

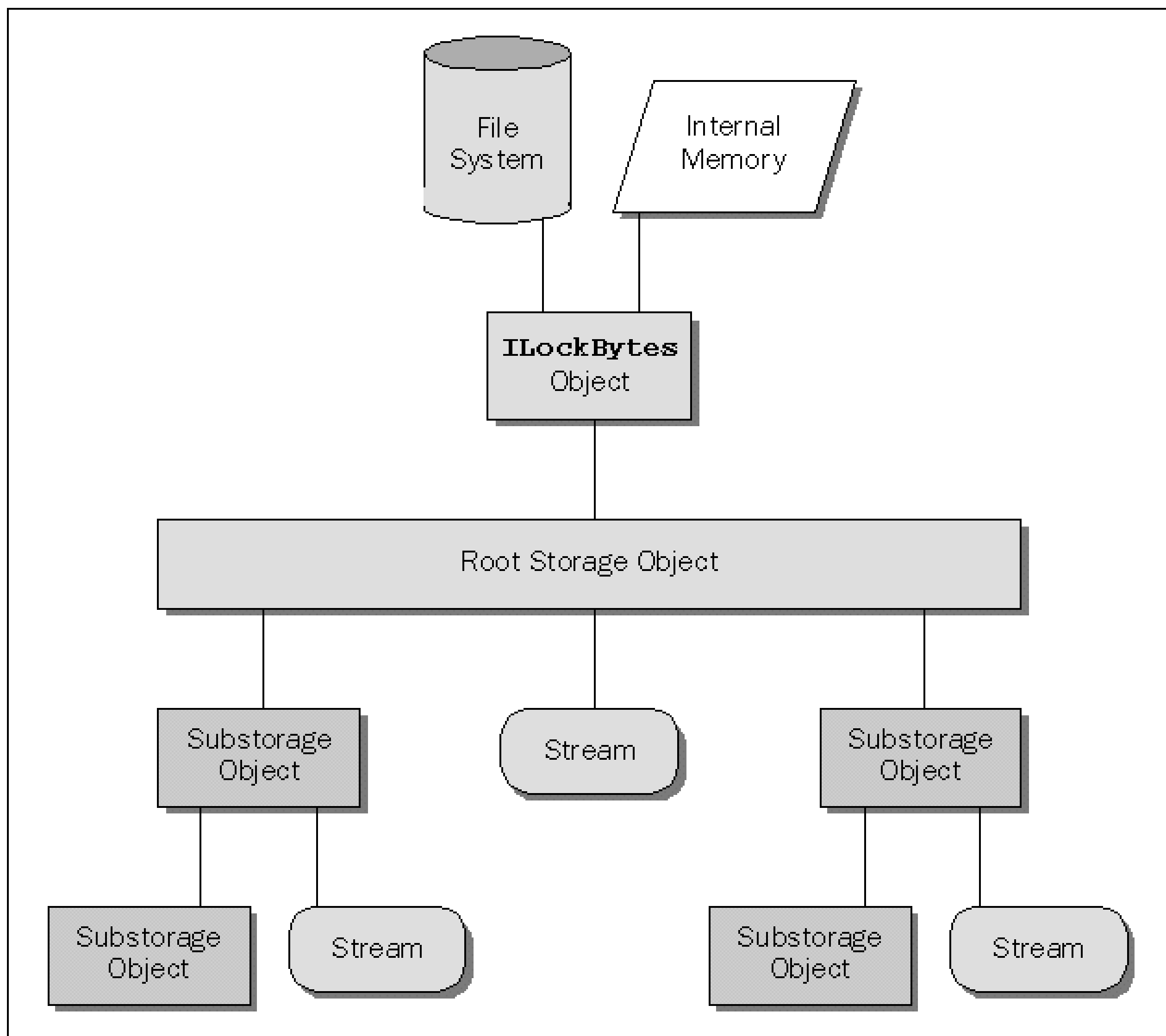
If you're like me and write most of your code using MFC, you've certainly dealt with some of MFC's file I/O classes, such as **`CFile`**, **`CArchive`**, or **`CMemFile`**. If you've already written some MFC code that uses regular files and performs serialization from MFC based objects, you'll want to know more about a class named **`COleStreamFile`** which derives from **`CFile`**. **`COleStreamFile`** has a member to hold an **`IStream`** pointer and allows you to act upon the associated stream as if it was a file. This means that you can

replace the use of the `CFile` objects with `COleStreamFile` objects and continue to use the rest of the code exactly as it stands.

This is the route that MFC takes with its implementation of compound documents. The `COleDocument` class derives from `CDocument` and replaces the use of `CFile` objects with `COleStreamFile` objects. If you turn your current application into an OLE aware application, it makes it very easy to deal with compound files, since you can leave your current implementation of the `Serialize()` member function intact.

Microsoft's Implementation

Microsoft provides two implementations of structured storage; one for disk files and one for memory. The memory implementation allows you to create compound files in memory and save them to anywhere you like.



When you're using structured storage in memory you can make the embedded component objects think

that they're talking to disk-based files when they're really talking to RAM. You can do this by using several OLE API functions, such as `CreateILockBytesOnHGlobal()`, `StgCreateDocfileOnILockBytes()` and `StgOpenStorageOnILockBytes()`. Together, these functions allow you to create an `ILockBytes` pointer on some global memory, create a compound file on the `ILockBytes` pointer (which gives you back an `IStorage` pointer) and continue to use the `IStorage` pointer, just as you would any other `IStorage` pointer, so you can create or access other substorages or streams. We'll make use of this in the next chapter, where you'll see how to save embedded objects to a database.

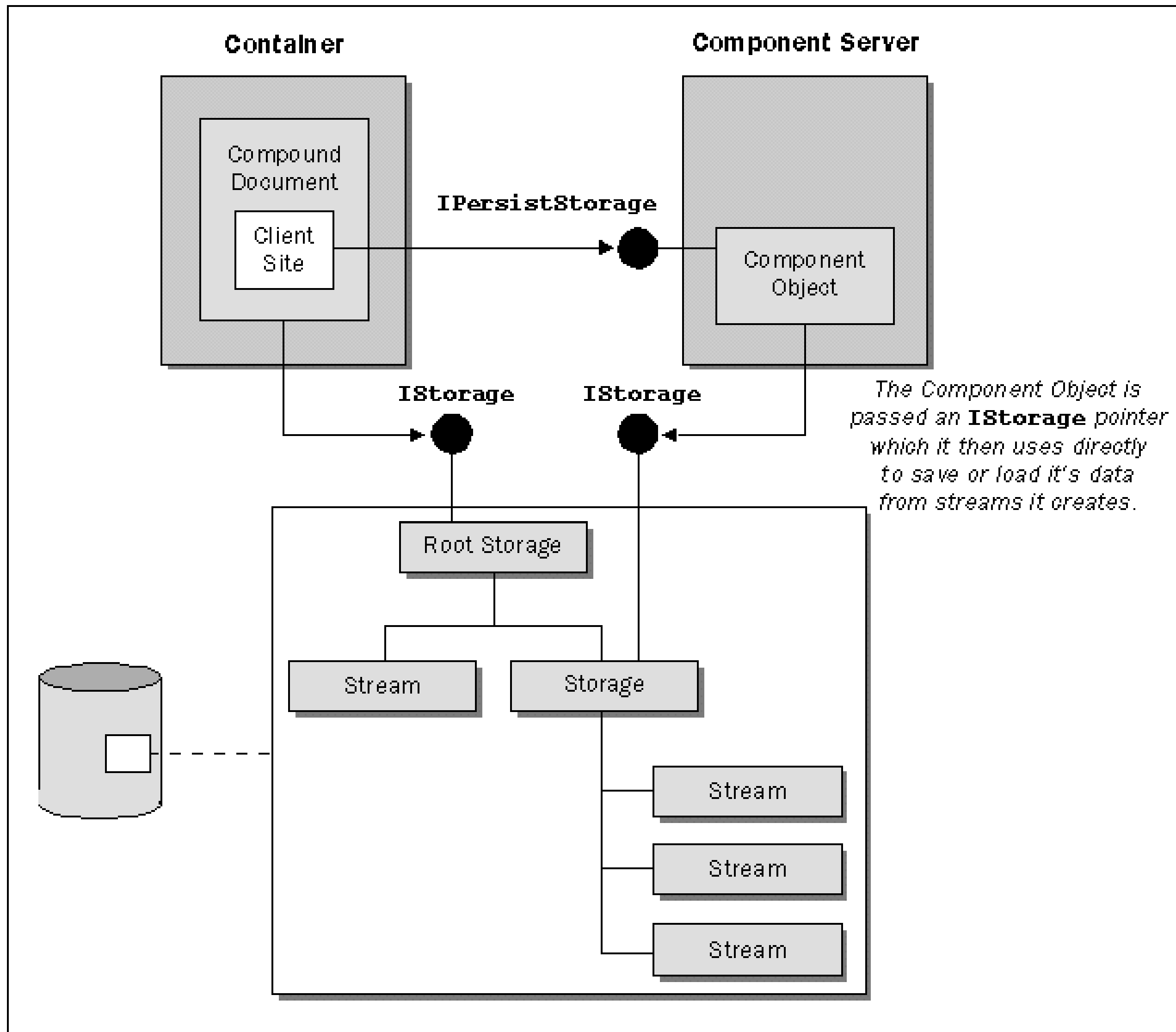
The Persistent Object

When you share data between two applications, there are times when you might know the structure of an object. If you tell a server application to give you the data, you can deal with it accordingly, since you know how the data is arranged. This is known as **structured data**.

However, there are other times when you might not know how the data is arranged, but still need to save or load an object. Having OLE objects embedded in a compound document is just one example of this. How do you do this if you don't know how the data is structured? OLE solves the problem by defining a protocol along with a set of interfaces that allow an object to save or load itself.

This support is provided by the `IPersist`*xxx* interfaces, where *xxx* stands for `File`, `Storage` or `Stream`. If an object supports one of these interfaces, you know that you can pass the object a file, storage or stream respectively and ask it to save itself to it. The `IPersistFile` interface allows an object to save its data to a file in your file system. The `IPersistStorage` allows an object to create streams within a given storage and save its data however it sees fit. Finally, the `IPersistStream` interface allows an object to save its data to a given stream.

For example, the `IPersistStorage` interface must be implemented by content objects embedded into a compound document if they are to be saved in a compound file provided by the container.



In the earlier example that demonstrated aggregation, involving the client and the two servers, we also made use of structured storage and compound files in a way that is different from the usual implementation that MFC provides. The current MFC implementation handles compound files for OLE embedding documents and servers, but in the example, all I wanted to do was get the data from the data server to a compound file and back again. In this section, you'll see how I did it.

The Client Implementation

The client application provides a compound file and creates a storage that the storage object could use to save the data however it sees fit. Because the client application uses the MDI model, a document in the client application can be created in one of two ways: the user can choose File/Open... or File/New from the menu. When a new document is created, the constructor creates a COM instance of `CLSID_StorageObject`. This object implements `IPersistStorage`, an interface that allows a client to tell the object to save itself to a storage the best way it knows how.

The client provides the object with a pointer to a storage created in a compound file. When the client creates or opens any storages, it's given a pointer to an **IStorage** interface. The client can then call one of two functions in the object's **IPersistStorage** interface and pass it the **IStorage** pointer. **InitNew()** is called if the client is initializing the object from scratch. **Load()** is called when the client wants to initialize the object from data that was saved during a previous session.

Persistent Object Initialization

Since a storage object is always available for every document created, I needed to initialize the storage object as soon as it had been created (passing it the **IStorage** object). There are two places to handle the initialization process. The first is **OnNewDocument()**, which is called if the user chooses to create a new document. The second is **OnOpenDocument()**, which is called when the user chooses to open an existing document file. This is what the **OnNewDocument()** function looks like:

```
BOOL CClientDoc::OnNewDocument()
{
   if (!CDocument::OnNewDocument())
      return FALSE;

   LPSTORAGE lpStorage;
   LPPERSISTSTORAGE lpPersistStorage;
   HRESULT hr;

   hr = m_lpUnknown->QueryInterface(IID_IPersistStorage,
      (LPVOID*)&lpPersistStorage);
   if (FAILED(hr))
      return FALSE;

   hr = StgCreateDocfile(NULL, STGM_DIRECT | STGM_READWRITE
      | STGM_CREATE | STGM_DELETEONRELEASE | STGM_SHARE_EXCLUSIVE,
      0, &lpStorage);
   if (FAILED(hr))
   {
      RELEASE_INTERFACE(lpPersistStorage);
      return FALSE;
   }

   m_lpStorage = lpStorage;

   hr = lpPersistStorage->InitNew(lpStorage);

   RELEASE_INTERFACE(lpPersistStorage);
   if (FAILED(hr))
      return FALSE;

   return TRUE;
}
```

The function begins its work by asking the storage object for its **IPersistStorage** interface (which will receive an **IStorage** pointer briefly after the storage has been created). It then creates a compound file on the file system, using the **StgCreateDocfile()** OLE API function. Notice that we don't pass it a file name. That's because at this point, we don't have a file name. The document title bar probably contains something like untitled or document1 because the user hasn't saved the document yet, so we need to create the compound file with a temporary name. OLE can do this for us if we pass it **NULL** for the file name.

The second thing to notice about the way the file is created is a flag named **STGM_DELETEONRELEASE**. This flag allows OLE to delete the file as soon as the **IStorage** pointer is released. This works out great because, when the user chooses to save the file for the first time, we'll receive a path name in the **OnSaveDocument()** function. We can then use the path name to create a new compound file, copy the

content of the original temporary file into the new files and release the original temporary file (which causes the file to be deleted).

The only unresolved matter is, how do we know in **OnSaveDocument()** that we should create a new document and copy the data to the new document, as opposed to simply saving a document to the same file obtained when the file was opened via File/Open... That's easy. We keep some type of flag around that we can use to determine the outcome. I created a data member to keep the path name around when **OnOpenDocument()** is called. However, since it's initialized to **NULL** in the constructor, I can compare the cached path name against the passed in path name in **OnSaveDocument()** to determine whether to save the file to a new file or the existing one. The same technique works for a File/Save As... operation, because the path names will be different then too.

When the user chooses File/Open..., the **OnOpenDocument()** function is called. Here's my code for this function:

```
BOOL CClientDoc::OnOpenDocument(LPCTSTR lpszPathName)
{
   USES_CONVERSION;

   if (!CDocument::OnOpenDocument(lpszPathName))
      return FALSE;

   // Save name for comparing later in OnSaveDocument.
   m_strPathName = lpszPathName;

   LPSTORAGE lpStorage;
   LPPERSISTSTORAGE lpPersistStorage;
   HRESULT hr;

   hr = m_lpUnknown->QueryInterface(IID_IPersistStorage,
      (LPVOID*)&lpPersistStorage);
   if (FAILED(hr))
      return FALSE;

   hr = StgOpenStorage(T2OLE(lpszPathName), NULL, STGM_DIRECT
      | STGM_READWRITE | STGM_SHARE_EXCLUSIVE, NULL, 0, &lpStorage);
   if (FAILED(hr))
   {
      RELEASE_INTERFACE(lpPersistStorage);
      return FALSE;
   }

   m_lpStorage = lpStorage;

   hr = lpPersistStorage->Load(lpStorage);
   RELEASE_INTERFACE(lpPersistStorage);
   if (FAILED(hr))
      return FALSE;

   return TRUE;
}
```

After gathering the **IPersistStorage** pointer from the storage object, I open the compound file and get an **IStorage** pointer in return. I then take that storage pointer and pass it to the persistent interface by calling its **Load()** function. Note that you can only call **Load()** or **InitNew()** once. If you call one of these functions, you're not allowed to call the other.

Saving a Persistent Object

If the user wants to manipulate the data across sessions, they'll have to choose File/Save or File/Save As...,

which will result in a call to a function named **OnSaveDocument()**. I coded the function as follows:

```
BOOL CClientDoc::OnSaveDocument(LPCTSTR lpszPathName)
{
   USES_CONVERSION;

   LPSTORAGE lpStorage;
   LPPERSISTSTORAGE lpPersistStorage;
   HRESULT hr;
   BOOL bSameAsLoad;

   hr = m_lpUnknown->QueryInterface(IID_IPersistStorage,
      (LPVOID*)&lpPersistStorage);
   if (FAILED(hr))
      return FALSE;

   if (m_strPathName != lpszPathName)   // Do we have a SaveAs case.
   {
      hr = StgCreateDocfile(T2OLE(lpszPathName), STGM_DIRECT
         | STGM_READWRITE | STGM_CREATE | STGM_SHARE_EXCLUSIVE
         , 0, &lpStorage);
      if (FAILED(hr))
      {
         RELEASE_INTERFACE(lpPersistStorage);
         return FALSE;
      }

      m_lpStorage->Release();
      m_lpStorage = lpStorage;

      bSameAsLoad = FALSE;
      m_strPathName = lpszPathName;
   }
   else
      bSameAsLoad = TRUE;

   hr = lpPersistStorage->Save(m_lpStorage, bSameAsLoad);
   if (SUCCEEDED(hr))
   {
      hr = lpPersistStorage->SaveCompleted(bSameAsLoad ? NULL
         : m_lpStorage);
   }

   RELEASE_INTERFACE(lpPersistStorage);
   if (FAILED(hr))
      return FALSE;

   SetModifiedFlag(FALSE);

   return TRUE;
}
```

As usual, I get the **IPersistStorage** interface from the object. The interface pointer will be used when it's time to tell the object to save itself to the provided storage. Next, I compare the two path names and, if they don't match, I need to create a new compound file, copy the contents from the original file to the new file and close the old one. Either way, I need to tell the object whether it's saving itself to the original storage that was passed to it in **OnNewDocument()** or **OnLoadDocument()**.

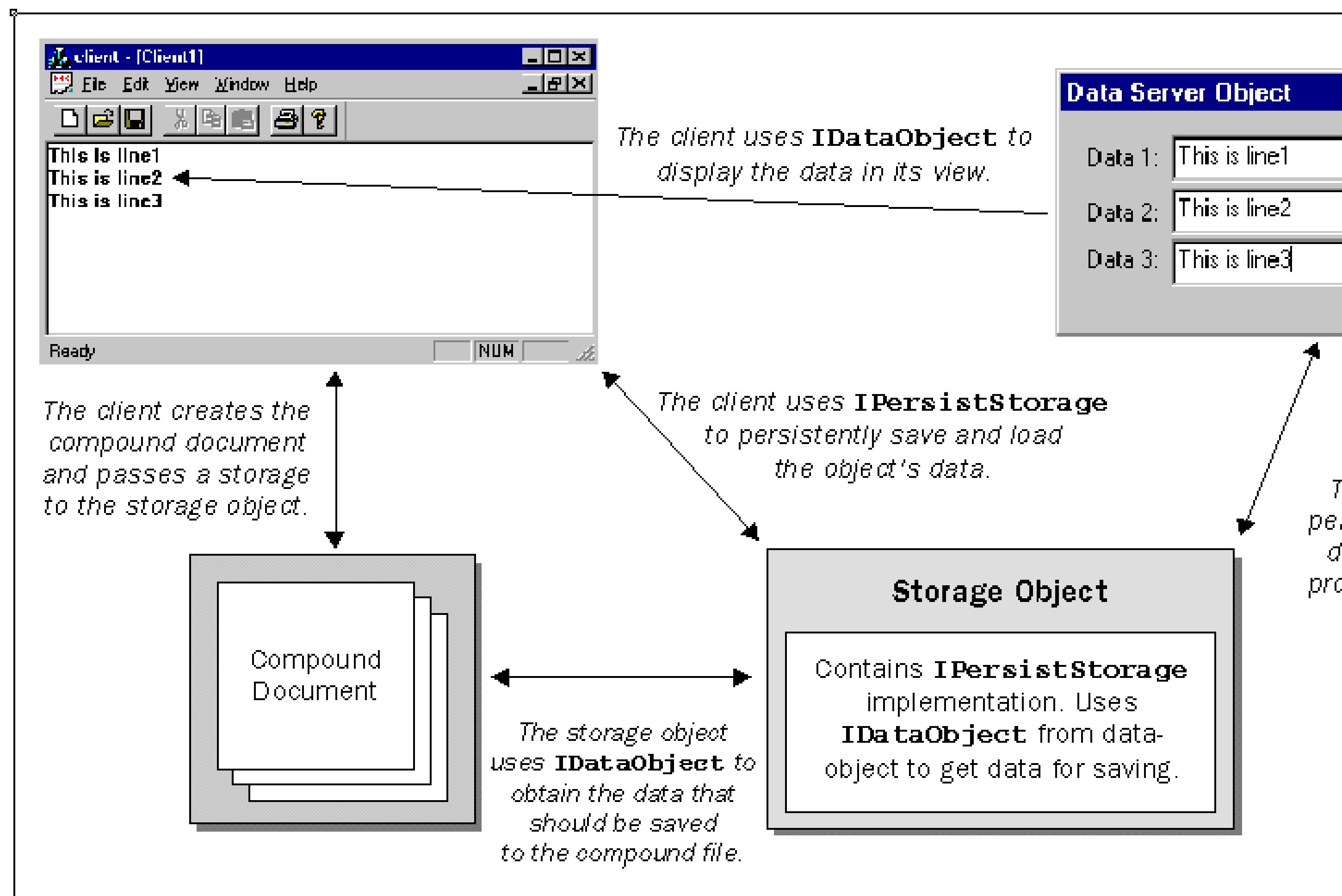
When an object's **IPersistStorage::Save()** function returns, the object is not allowed to write any incremental saves it might have otherwise performed until the **IPersistStorage::SaveCompleted()** function is called. The reason for this is that the object might have just saved itself to a different storage when **Save()** was called. If this is the case, the object is not allowed to hold on to the storage for incremental saves until **SaveCompleted()** is called. At that point, the object has the opportunity to release the old storage it was holding on to and cache the new storage that it's being passed.

But why do we have to call two functions to perform a save instead of simply calling one? Remember that I mentioned that compound files support transactions. A container application might save its data incrementally by calling all of the objects' `IPersistStorage::Save()` functions. Once done, it calls `IStorage::Commit()` on the storages used by the objects. Finally, it would then call the objects' `IPersistStorage::SaveCompleted()` to tell the them that the operation has been completed.

Now, let's say that the application is performing incremental saves based on a timer (say, every five minutes). The user might change their mind and wish to revert the changes. If the application calls `IPersistStorage::Save()` and then calls `IPersistStorage::SaveComplete()` with no call to `IStorage::Commit()`, the application can save the data to a buffer without committing it, until the user is ready to commit the data. (We'll discuss the code for `IPersistStorage::Save()` and `IPersistStorage::SaveCompleted()` in the next section.)

The Storage Server Implementation

It's the storage object which implements the `IPersistStorage` interface. The implementation I provided is fully featured and makes use of the aggregate data object to get and set the data seen through the data object's dialog box and the client's view.



To safely get back all of the information to fill the dialog box provided by the data object (and the view provided by the client), the storage object needs to save three pieces of information: the amount of memory it will need to allocate for the data (when it's told to reload the data in later sessions), the data itself and information about the type of data being stored in the storage provided.

The Storage Object's Constructor

The first thing I did in the header file was to define the interface that the COM object has to implement with the following code:

```
BEGIN_INTERFACE_PART(PersistStorage, IPersistStorage)
   STDMETHODIMP GetClassID(LPCLSID); // Defined in IPersist

   STDMETHODIMP IsDirty(void);       // Members of IPersistStorage
   STDMETHODIMP InitNew(LPSTORAGE);
   STDMETHODIMP Load(LPSTORAGE);
   STDMETHODIMP Save(LPSTORAGE, BOOL);
   STDMETHODIMP SaveCompleted(LPSTORAGE);
   STDMETHODIMP HandsOffStorage(void);
END_INTERFACE_PART(PersistStorage)
```

Once the definition has been created, the implementation must be provided. Let's start with the members I initialized in the constructor:

```
CStorageServObj::CStorageServObj()
{
   EnableAggregation();
   AfxOleLockApp();

   // IPersistStorage Support
   m_szUserType = _T("Data/Storage Object");
   m_lpStorage = NULL;
   m_lpAggregate = NULL;
   m_lpDataObject = NULL;
   m_psState = PSSTATE_UNINIT;
}
```

OLE allows us to place a marker inside of a storage so that we can later check for this marker and determine whether the storage does in fact have data we understand. This is done with an OLE API function named **`WriteFmtUserTypeStg()`**. The function allows us to save vital information, such as the clipboard format the data is in (this can also be a registered clipboard format), and a unique string that is used to identify the type of object stored in the storage. For example, in our case, we're saving information gathered from the data server, but it's the storage server that is performing the saving and loading. Therefore, I named the type of object being stored, **`"Data/Storage Object"`**. Later in this section, you'll see when we call the **`WriteFmtUserTypeStg()`** and how we read the information back using the **`ReadFmtUserTypeStg()`** API function.

I also needed to keep track of the state of the **`IPersistStorage`** implementation at all times, so I created a member, called **`m_psState`**, to hold the current state. The states include:

> **`PSSTATE_UNINIT`**, which means the object has not had its **`InitNew()`** or **`Load()`** function called yet.
> **`PSSTATE_SCRIBBLE`**, which means that one of its initialization functions has been called and the object can feel free to incrementally save any information it sees fit.
> **`PSSTATE_ZOMBIE`**, which this means that the object's **`Save()`** function has been called and the object should stop performing incremental saves until its **`SaveCompleted()`** function is called.

PSSTATE_HANDSOFF, which means that the object's **HandsOffStorage()** function has been called and it has released all of its streams and the primary storage it was holding on to. The object must wait until its **SaveCompleted()** function is called again before it can hold onto a storage again.

As the client calls different functions of the **IPersistStorage** interface, the object will change states. All of these values are defined as an enumeration in the file `Ole2ext.h` that is provided with the sample code.

Initializing through IPersistStorage

When the client wants to initialize either a newly created storage object or an existing storage object (obtained from a compound file), it will call **InitNew()** or **Load()**, respectively, using the object's **IPersistStorage** interface.

The **InitNew()** function must create any streams it will need before its **Save()** function is called. This is sort of a safety mechanism that almost guarantees that the object can save its data under low memory conditions. If the streams are already there, all you have to do is copy the data from memory to the streams and you're done. This assumes that the streams are pre-allocated with enough memory to hold the data (and OLE does provide a means for doing this, using **IStream::SetSize()**). In my case, I didn't know ahead of time how much memory I would need, since the strings could be any size, so I couldn't pre-allocate the streams. That's life!

I also took the opportunity to call **WriteFmtUserTypeStg()** to save the object's information into the storage (OLE does the work for you) and set the object's state to **PS_SCRIBBLE**. Here's what the code looks like without error handling (you can find the complete code on the CD):

```
STDMETHODIMP CStorageServObj::XPersistStorage::InitNew(LPSTORAGE
   lpIStorage)
{
   USES_CONVERSION;

   METHOD_PROLOGUE(CStorageServObj, PersistStorage)

   pThis->m_SizeStream.CreateStream(lpIStorage, _T("SIZE"));
   pThis->m_ContentsStream.CreateStream(lpIStorage, _T("CONTENTS"));

   WriteFmtUserTypeStg(lpIStorage, CF_TEXT, T2OLE(pThis->m_szUserType));

   lpIStorage->AddRef();
   pThis->m_lpStorage = lpIStorage;
   pThis->m_psState = PSSTATE_SCRIBBLE;

   return NOERROR;
}
```

The **Load()** function has to open the existing streams within the provided storage, load the data from the streams and pass the data to the data object (which is aggregated by the storage object, as we saw earlier in the chapter). The following function contains the code for the **Load()** function with the exception of the error checking code and the transferring of data to the data object using the **IDataObject** interface. We'll discuss the **IDataObject** implementation when we cover Uniform Data Transfer. Here's what the **Load()** function looks like:

```
STDMETHODIMP CStorageServObj::XPersistStorage::Load(LPSTORAGE lpIStorage)
{
   METHOD_PROLOGUE(CStorageServObj, PersistStorage)

   HGLOBAL hData;
```

```
    LPTSTR lpData;
    ULONG cb;

    pThis->m_SizeStream.OpenStream(lpIStorage, _T("SIZE"));
    pThis->m_ContentsStream.OpenStream(lpIStorage, _T("CONTENTS"));

    pThis->m_SizeStream.SeekToBegin();
    pThis->m_SizeStream.Read(&cb, sizeof(ULONG));

    hData = GlobalAlloc(GMEM_SHARE|GMEM_MOVEABLE, cb);
    lpData = (LPTSTR)GlobalLock(hData);

    pThis->m_ContentsStream.SeekToBegin();
    pThis->m_ContentsStream.Read(lpData, cb);
    GlobalUnlock(hData);

    pThis->m_lpStorage = lpIStorage;
    pThis->m_psState = PSSTATE_SCRIBBLE;
    lpIStorage->AddRef();

    // .
    // . Code for transferring data goes here.
    // .

    return NOERROR;
}
```

Saving through IPersistStorage

The OLE specification states that when a client saves an object, it must tell the object if the object is being saved to the same storage as the one it was loaded from or if it's being saved to a brand new storage.

The object would obviously have to save itself to a new storage if the user chose the File/Save As... command from the menu, or if the user originally created a new document using the File/New... choice and is now saving the document for the first time. Therefore, the object would have to create the necessary streams on the new storage before it can save its data. However, the object doesn't release its original storage until the **SaveCompleted()** function is called, since the object might very well be instructed to continue to use the same storage the next time **Save()** is called. In other words, the object doesn't always release the storage. The object might be told to save itself to a new storage when the **Save()** function is called, and then later it might be told to continue using the original storage when the **SaveCompleted()** function is called.

The object does not have to release the storage or streams at all, if it's saving itself to the same storage that it originally initialized itself from (even when **SaveCompleted()** is called).

The client has to call **Save()** before it calls **SaveCompleted()**. Inside of **Save()**, the object checks for a different storage and acts appropriately. If it's a new storage, it creates the necessary streams, saves its data and closes the old streams down, while creating pointers to new streams in the new storage. If it's the original storage, it simply saves its data and returns. In either case, it has to set its state to **PS_ZOMBIE** before returning back to the caller. Here's what the **Save()** function looks like without error handling:

```
STDMETHODIMP CStorageServObj::XPersistStorage::Save(LPSTORAGE lpIStorage,
    BOOL bSameAsLoad)
{
    USES_CONVERSION;

    METHOD_PROLOGUE(CStorageServObj, PersistStorage)

    ULONG cb;
    HRESULT hr;
```

```
    LPTSTR lpData;

    // These variables allow generic usage of streams.
    COleStreamFile* pSizeStream;
    COleStreamFile* pContentsStream;

    // These variables will be used if bSameAsLoad == FALSE.
    COleStreamFile NewSizeStream;
    COleStreamFile NewContentsStream;

    if (bSameAsLoad)
    {
       // Assign to generic pointers.
       PSizeStream = &(pThis->m_SizeStream);
       pContentsStream = &(pThis->m_ContentsStream);

       // Seek to beginning for writing.
       pSizeStream->SeekToBegin();
       pContentsStream->SeekToBegin();
    }
    else
    {
       // Create new streams in new storage.
       NewSizeStream.CreateStream(lpIStorage, _T("SIZE"));
       NewContentsStream.CreateStream(lpIStorage, _T("CONTENTS"));

       // Assign to generic pointers.
       pSizeStream = &NewSizeStream;
       pContentsStream = &NewContentsStream;

       WriteFmtUserTypeStg(lpIStorage, CF_TEXT,
                           T2OLE(pThis->m_szUserType));
    }

    // .
    // . Code for obtaining data from data object goes here.
    // .

    // Lock the data and write the size and data to the streams.
    cb = GlobalSize(stm.hGlobal);
    lpData = (LPTSTR)GlobalLock(stm.hGlobal);

    pSizeStream->Write((LPVOID)&cb, sizeof(ULONG));
    pContentsStream->Write((LPVOID)lpData, cb);

    // Release the global data returned by the aggregate.
    GlobalUnlock(stm.hGlobal);
    ReleaseStgMedium(&stm);

    // Must Release the new streams on exit of function.
    if (!bSameAsLoad)
    {
       NewSizeStream.Close();
       NewContentsStream.Close();
    }

    // Set access mode to ZOMBIE.
    pThis->m_psState = PSSTATE_ZOMBIE;
    return NOERROR;
}
```

When the client calls **`SaveCompleted()`**, the OLE specification states that the client needs to pass in a storage *only* if the object saved itself to a different storage than the one it was originally initialized from. If this is the case, the object must release the streams and the storage it's holding on to and should hold onto the new storage passed to **`SaveCompleted()`**. It should then reopen its streams appropriately. Finally, the object has to reset its state to **`PS_SCRIBBLE`**. This is exactly what I did in my implementation of **`SaveCompleted()`**:

```
STDMETHODIMP CStorageServObj::XPersistStorage::SaveCompleted(LPSTORAGE
   lpIStorage)
{
   METHOD_PROLOGUE(CStorageServObj, PersistStorage)

   // Did we write to a new storage?
   if (NULL != lpIStorage)
   {
      // Stop all operations on current streams IMMEDIATELY!
      pThis->m_SizeStream.Abort();
      pThis->m_ContentsStream.Abort();

      // Open streams in new storage.
      pThis->m_SizeStream.OpenStream(lpIStorage, _T("SIZE"));
      pThis->m_ContentsStream.OpenStream(lpIStorage, _T("CONTENTS"));

      // Reassign storage.
      RELEASE_INTERFACE(pThis->m_lpStorage);
      lpIStorage->AddRef();
      pThis->m_lpStorage = lpIStorage;
   }

   // Change state back to scribble.
   pThis->m_psState = PSSTATE_SCRIBBLE;
   return NOERROR;
}
```

The last function to look at is **HandsOffStorage()**. After a client calls **Save()**, it might want to guarantee that the object will not write any data to its storage. The client can call **HandsOffStorage()** and, at that point, the object must release all of its pointers to any streams and the storage. The client might later call **SaveCompleted()** to give control of a storage back to the object. This function is very easy to code. Here's my implementation:

```
STDMETHODIMP CStorageServObj::XPersistStorage::HandsOffStorage(void)
{
   METHOD_PROLOGUE(CStorageServObj, PersistStorage)

    // Can only be in scribble or zombie mode otherwise there must
    // be a bug in the client
    if (PSSTATE_UNINIT == pThis->m_psState
       || PSSTATE_HANDSOFF == pThis->m_psState)
        return E_UNEXPECTED;

   // Shut down streams.
   pThis->m_SizeStream.Close();
   pThis->m_ContentsStream.Close();

   RELEASE_INTERFACE(pThis->m_lpStorage);

   pThis->m_psState = PSSTATE_HANDSOFF;
   return NOERROR;
}
```

The other functions in the **IPersistStorage** interface are trivial to code, so you shouldn't have a problem understanding the code that I provided in my implementation.

Uniform Data Transfer

There might be times when an application exposes its data to another application in a structured manner. In other words, both applications understand the format of the data. Since there are a number of ways in which data might be exchanged between two applications (by drag-and-drop or via the clipboard, for example), the underlying OLE technology that allows this transfer to take place is known as **Uniform Data Transfer**.

Data objects exposed by servers implement an interface named **`IDataObject`**, which allows potential clients to access or write data to and from the objects using uniform data transfer in a specified clipboard format. In the case of embedded component objects, the data is usually never understood directly by the containers (except for some intermediate clipboard formats such as **`CF_EMBEDDEDOBJECT`**). OLE however, understands clipboard formats, such as **`CF_METAFILEPICT`** or **`CF_DIB`**, which OLE uses to get presentation data from the objects.

In the aggregation example we've been looking at in this chapter, I implemented Uniform Data Transfer (UDT) in the data server. This involved having the data server's objects include support for an interface named **`IDataObject`**. The interface has several functions, of which two are necessary for transferring information back and forth.

`IDataObject::GetData()` allows a caller to retrieve data in a specified format. If the server supports the format, the data will be returned, otherwise an error is returned. The format is specified using a structure named **`FORMATETC`**. In the old days, data was simply passed across applications using global memory. This **`FORMATETC`** structure provides a more flexible way of asking for data, since it allows the caller to specify the format the data should come back in, information about the device for which the data is to be rendered (global memory, disk files, streams, or storages), the aspect of the data (whether it should be the complete content, a portion, or an icon representation) and the type of medium that should be used for transferring the data. The medium can be global memory, a file, a stream, a storage, a metafile picture, or a bitmap. This is what the structure looks like:

```
typedef struct  tagFORMATETC {
   CLIPFORMAT cfFormat;
   /* [unique] */ DVTARGETDEVICE __RPC_FAR *ptd;
   DWORD dwAspect;
   LONG lindex;
   DWORD tymed;
}FORMATETC;
```

When the client calls **`GetData()`**, it must provide a pointer to a **`STGMEDIUM`** variable. The data object will then use this pointer to return the requested data:

```
typedef struct tagSTGMEDIUM {
   DWORD tymed;
   union {
       HBITMAP hBitmap;
       HMETAFILEPICT hMetaFilePict;
       HENHMETAFILE hEnhMetaFile;
       HGLOBAL hGlobal;
       LPOLESTR lpszFileName;
       IStream* pstm;
       IStorage* pstg;
       } u;
   IUnknown* pUnkForRelease;
}STGMEDIUM;
```

The first field determines the type of medium used for returning the data. The second field is a union of the available types of medium, the data could possibly come back in. You can provide your own registered clipboard format and return or get data in a format known only to the server and client.

The **pUnkForRelease** pointer is used in situations where the data returned to a client is being shared with different applications and must not be freed until the last application has stopped using the data. When an application no longer needs to use the data specified, it simply calls **IUnknown::Release()**, using the **pUnkForRelease** pointer. The data is then freed when the reference count of the associated **IUnknown** reaches zero.

In the client's view class, you'll notice that **OnDraw()** contains code similar to this:

```
STGMEDIUM   stm;
FORMATETC   fe;

fe.cfFormat = CF_BITMAP;
fe.ptd = NULL;
fe.dwAspect = DVASPECT_CONTENT;
fe.lindex = -1;
fe.tymed = TYMED_GDI;

pDataObject->GetData(&fe, &stm);
```

It initializes the **FORMATETC** structure, requesting for data to come back as a bitmap, and calls the **IDataObject::GetData()** function. The storage server also contains similar code, which looks like this:

```
FORMATETC fe;
STGMEDIUM stm;
SETFORMATETC(fe, CF_TEXT, TYMED_HGLOBAL);

pThis->m_lpDataObject->GetData(&fe, &stm);
```

This time, I used a macro to initialize the **FORMATETC** structure. The storage object requests the data to come back in the format of **CF_TEXT** using an **HGLOBAL** as the medium.

When the data server receives the **GetData()** call, it has to determine whether it supports the data format being requested and return it using the stipulated medium. The following code is extracted from the data object:

```
STDMETHODIMP CDataServObj::XDataObject::GetData(LPFORMATETC pFE,
   LPSTGMEDIUM pStm)
{
   METHOD_PROLOGUE(CDataServObj, DataObject)
   CDataServDlg* pDlg = &(pThis->m_DataServDlg);

    UINT uCF = pFE->cfFormat;

    // Check the aspects we support.
    if (!(DVASPECT_CONTENT & pFE->dwAspect))
        return DATA_E_FORMATETC;

    pStm->pUnkForRelease = NULL;

    // Go render the appropriate data for the format.
    switch (uCF)
    {
      case CF_METAFILEPICT:
         pStm->tymed = TYMED_MFPICT;
         return pDlg->RenderMetafilePict(&pStm->hGlobal);

      case CF_BITMAP:
```

```
            pStm->tymed = TYMED_GDI;
            return pDlg->RenderBitmap(&pStm->hGlobal);

         case CF_TEXT:
            pStm->tymed = TYMED_HGLOBAL;
            return pDlg->RenderNative(&pStm->hGlobal);
      }

       return DATA_E_FORMATETC;
   }
```

Although we made use of UDT in the data server, storage server and the client for the simple purpose of sharing data, there are other uses for UDT which allow us to share data in other ways. This is the next topic of discussion.

Other Uses for UDT

Windows' multitasking ability means it's desirable to share data between applications. First came the **clipboard**. The clipboard allows us to copy data from one application into a common place and then paste the data to another application.

Next came DDE, which allowed us to share data with other applications programmatically. You could request data from an application and establish links that acted as notifications of data changes.

Although these technologies have their advantages, they also inherited many disadvantages. For example, the only way to pass a large bitmap around was to use global memory. There was no provision for passing it via files on disk (which is where it will eventually end up anyway).

This is the reason Microsoft came up with UDT. Although UDT was originally created to support OLE documents, it also plays a major role in **clipboard transfers** and **drag-and-drop**.

Using UDT with the Clipboard

Using OLE's support for the clipboard, applications can now place a pointer to an `IDataObject` interface on the clipboard and the user can paste the object into another application. If one of the applications doesn't support OLE's method of using the clipboard, they can still get back data in the expected format. However, an application that understands and implements OLE clipboard support can extract the data using the `IDataObject` interface. Providing that the object is capable of becoming an embedded object, this allows the application to query the object for different formats and reactivate the server that originally created it .

The non-OLE application has to communicate directly with the clipboard at all times. If the object originally placed on the clipboard supports embedding, the server information is not attached to the data pasted into the non-OLE application.

The application providing the object to be placed on the clipboard is known as the **data source application**. It begins the process of placing the data on the clipboard by calling `OleSetClipboard()`, passing it a pointer to the `IDataObject` interface. `OleSetClipboard()` calls `AddRef()` on the `IDataObject` pointer it received. The source application can then release the object by calling its `IUnknown::Release()` to free the application from any further responsibility for the object. At this point, the OLE clipboard will be the only one holding a reference to the `IDataObject`.

Note that OLE uses a form of delayed rendering for the data offered by the object. This means that the

data is never placed on the clipboard until an application places a request to get a copy of the data from the clipboard. If the source application needs to leave the data on the clipboard after the application has been closed, it must call **`OleFlushClipboard()`** before closing down.

When **`OleSetClipboard()`** is called, OLE actually places a newly created object containing an **`IDataObject`** interface on the clipboard. This allows it to hold on to the original object for other clipboard requests. When a request comes in, OLE calls the original data object's **`GetData()`** function and copies the data into the newly created object. It then sends the **`IDataObject`** pointer of the newly created object to the destination application. If the destination application doesn't understand OLE, it gets a copy of the actual data with no **`IDataObject`** support. When the created object is sent to an application, OLE creates another one to place on the clipboard again.

Before OLE makes the data available to other applications, it first iterates through all of the clipboard formats that the object supports and places them on the clipboard. If a client application requests the current clipboard data in a particular format, OLE can check the available formats against the requested one. The data will always be available via global memory as the medium. The good news is that OLE will transfer the data upon request, back to its original storage medium when an OLE application gets the data object from the clipboard.

When an application that implements OLE's support for the clipboard wants to paste data from the clipboard, it calls **`OleGetClipboard()`**. OLE then sends the application the **`IDataObject`** sitting in the clipboard.

If the data in the clipboard was placed there by an application that doesn't support OLE, the OLE-enabled destination application will get back a synthesized **`IDataObject`**, which means that the object might not contain accurate information about the creator of the object. This would make it difficult for the object's server to become activated for updating the data.

Enough theory, let's find out how MFC makes all this easier.

Supporting the Clipboard Using MFC

MFC contains two classes that support transferring data objects via the clipboard. The first class, **`COleDataSource`**, implements a COM object class with an implementation of the **`IDataObject`** interface. The second class is called **`COleDataObject`** and simply provides a data member to hold a pointer to an **`IDataObject`** interface obtained from the clipboard, or via a drag-and-drop operation, along with some member functions to manipulate the contained **`IDataObject`** member.

The application that wishes to place data on the clipboard uses the **`COleDataSource`** class to do so. The class has functions to cache the data and place it on the clipboard. When the source application is ready to place the data on the clipboard, it will need to allocate memory for the data to place on the clipboard (by calling **`GlobalAlloc()`** or something similar), create an instance of **`COleDataSource`** on the heap, cache the data using one of the cache functions available and call **`COleDataSource::SetClipboard()`**. The following code demonstrates this:

```
HGLOBAL hGlobal = // Call some function which returns the data
COleDataSource* pds = new COleDataSource;

pds->CacheGlobalData(CF_TEXT, hGlobal);
pds->SetClipboard();
```

On the destination side, the application that wants to get an **IDataObject** pointer from the clipboard will need to create an instance of **COleDataObject**. It can then call its functions to attach to the clipboard and get the data from the associated data object. The functions **AttachClipboard()** and **GetGlobalData()** (or **GetData()**, which is another variant) will do the job:

```
COleDataObject dobj;
HGLOBAL hGlobal;

dobj->AttachClipboard();
hGlobal = dobj->GetGlobalData(CF_TEXT);
```

You can use these two classes to make your current applications OLE-aware and still continue to support the clipboard with very little code on your part. The **COleDataObject** class even has support for querying the object to find out whether the data can be returned in a format that your application can work with. Check it out. You'll find that it'll be worth your time.

Supporting Drag-and-drop

Although sharing data using the clipboard is great, it does expect require the user to perform a number of steps. They have to select the data in the source application, cut or copy the data to the clipboard, select the destination application and, finally, paste the data into its client area.

Several applications have been using a home-brewed version of **drag-and-drop** within their windows for some time now. The problem is that each implementation has always been different. Therefore, in order to drag-and-drop between applications, a standard had to be developed.

This standard now lives in OLE as the Drag-and-drop technology. All that an application has to do is support one of two interfaces, depending on whether it's acting as the source or the destination.

The first interface is called **IDropSource** and is implemented by the source application. It contains two functions: **QueryContinueDrag()** and **GiveFeedBack()**. **QueryContinueDrag()** is used by OLE to determine whether the drag-and-drop operation should continue or be canceled. **GiveFeedBack()** is called continuously to set the mouse to the appropriate cursor. The mouse is changed, depending on the state of the drag-and-drop operation. There are several states: **DROPEFFECT_NONE**, **DROPEFFECT_MOVE**, **DROPEFFECT_COPY**, **DROPEFFECT_LINK** and **DROPEFFECT_SCROLL**.

The second interface is called **IDropTarget** and is implemented on the destination side. A window wishing to receive drag-and-drop objects must be registered with OLE as a drop target. When the window registers itself, it calls a function named **RegisterDragDrop()**, passing it two parameters. The first parameter is the window handle and the second is a pointer to the **IDropTarget** interface implemented by the target window. OLE keeps a map in memory (similar to the figure below) which maps the window handle to the **IDropTarget** and calls the functions of the interface at different points, depending on what the user is doing.

HWND	IDropTarget
B10	0x00571E30
250	0x00487C31
.	.
.	.
.	.
15C	0x00A37442
B4C	0x0023D436

The **IDropTarget** interface contains four functions which must be implemented by the destination application: **DragEnter()**, **DragOver()**, **DragLeave()** and **Drop()**. **DragEnter()** is called when the mouse moves into the window associated with the drop target. Once the mouse is inside the window, **DragOver()** is called for every move the mouse makes within the window. If the mouse exits the window, **DragLeave()** is called. Finally, if the object is dropped within a registered target window, **Drop()** is called.

When the source application is ready to initiate a drag-and-drop operation, it needs to create an object which supports the **IDataObject** interface. Next, it must call **DoDragDrop()**, passing it the **IDataObject** pointer, along with the **IDropSource** pointer implemented by the source application. The function will not return until the drag-and-drop operation has been completed or canceled.

Supporting Drag-and-drop Using MFC

MFC contains two classes that wrap up the implementation of **IDropSource** and **IDropTarget**. These classes are called **COleDropSource** and **COleDropTarget** respectively.

Initiating drag-and-drop operations in MFC involves using the **COleDataSource** class as we did for supporting the clipboard. The difference is that we call **COleDataSource::DoDragDrop()** instead of **SetClipboard()**. The following is an example of the code that the source application would have to implement to begin a drag-and-drop operation:

```
HGLOBAL hGlobal = // Call some function which returns the data
COleDataSource ds;

ds.CacheGlobalData(CF_TEXT, hGlobal);
DROPEFFECT dropEffect = ds.DoDragDrop(DROPEFFECT_COPY |
   DROPEFFECT_MOVE, NULL, NULL);
// Last two parameters would default to NULL if I didn't pass them.

if ((dropEffect & DROPEFFECT_MOVE) == DROPEFFECT_MOVE)
   DeleteSelectedData();       // Call function to delete selected data
```

When the source application calls `COleDataSource::DoDragDrop()`, it has the option of passing a pointer to a `COleDropSource` object as the final parameter. If you're happy with the current implementation of `IDropSource` provided by MFC, you won't have to override the class or even create an instance of the class. MFC will automatically create an instance inside of `COleDataSource::DoDragDrop()`. The default implementation has enough functionality to perform a decent job for the drag-and-drop operation, so unless you wish to display different cursors for it, you shouldn't need to override the class.

On the target side, the application must create an instance of the `COleDropTarget` class and pass it the `this` pointer of the `CWnd`-derived class used for the target window. This is done by calling a function named `COleDropTarget::Register()`.

Internally, `COleDropTarget::Register()` calls `RegisterDragDrop()`, passing it the appropriate information. The code to register the target should look like this:

```
class CDerivedView : public CScrollView
{
   // ...Other code
   COleDropTarget m_dropTarget;
   // ...Other code
};

class CDerivedView::OnCreate()
{
   m_dropTarget.Register(this);
}
```

If the window passed to `Register()` is a `CView` (or one of its derived classes), the drop target object will attempt to call the view's functions with the same name as the target. In other words, when the target's `OnDragEnter()` function is called, it will call the view's `OnDragEnter()` function. This allows you to override one class instead of two. The view will receive an object of type `COleDataObject` with the associated `IDataObject` pointer, contained within the object.

When the target application is ready to close down, it needs to call `COleDropTarget::Revoke()`. This tells OLE to remove the entry from its map. If you don't call this function, OLE will continue to look for it and really bad scary things might happen! Note that the source and target application can be the same, allowing you to use drag-and-drop operations for moving objects between different windows of the application, or even within the same window.

Once the target has been registered properly, the only thing left to do is override the appropriate members in the view class and wait for someone to initiate a drag-and-drop operation. The following code shows how the target view could handle the overrides:

```
DROPEFFECT CDerivedView::OnDragEnter(COleDataObject* pDataObject,
   DWORD dwKeyState, CPoint point)
{
   return OnDragOver(pDataObject, dwKeyState, point);
}

DROPEFFECT CDerivedView::OnDragOver(COleDataObject* pDataObject,
   DWORD dwKeyState, CPoint point)
{
   DROPEFFECT de;

   if ((dwKeyState & MK_CONTROL) == MK_CONTROL)
      de = DROPEFFECT_COPY;
   else
      de = DROPEFFECT_MOVE;
```

```
    return de;
}
```

Inside of **`OnDragEnter()`**, you can allocate any resources or initialize members before a drop is performed. In the above code, I need to perform the same tasks as in **`OnDragOver()`**, so I just include the code in one of the functions and call it from the other. All I did was to check if the control key is being held so that when the **`IDropSource::GiveFeedBack()`** function is called in the source application, it will receive the current state of the drag-and-drop operation. Basically, the drop source needs to know what kind of drop would occur if the mouse was released at that point in time. The only one who can answer that question is the target. For this reason, I send back an appropriate drop effect from the **`OnDragEnter()`** and **`OnDragOver()`** functions.

The last thing to see is the handling of the **`OnDrop()`** function. This would, of course, look different for your application because it really depends on what it intends to do with the **`COleDataObject`**:

```
BOOL CDerivedView::OnDrop(COleDataObject* pDataObject,
    DROPEFFECT dropEffect, CPoint point)
{
    CDerivedDoc* pDoc = GetDocument();
    HGLOBAL hGlobal = pDataObject->GetGlobalData(CF_TEXT);

    pDoc->PasteData(hGlobal);   // Place data into document.
    pDoc->UpdateAllViews(NULL);

    return TRUE;
}
```

In the above example, I simply fetched the data from the **`COleDataObject`** and called a function in the document to paste data into the document.

Summary

Once you know how to do it, implementing drag-and-drop operations is relatively simple. If you use MFC, the task becomes even easier. Even if you don't use any other OLE functionality in your application, clipboard support and drag-and-drop can add a lot of value to your application.

Now that you've seen some of the fundamental COM and OLE technologies, including aggregation, structured storage and UDT, you'll be prepared for the next chapter where we look at some further uses of OLE, particularly in the context of OLE document containers and servers.

OLE Containers and Servers

Combining multiple data formats in a single document has always been tricky, but it's something that users have always tried to do. Since day one, Windows has allowed us to run multiple applications side-by-side and, with the advent of the clipboard, users could cut and paste pieces of data from different applications to form a whole, combining images and text into one document to produce a truly compound document.

Although the flexibility that the clipboard offered was welcomed, there were some problems. For example, let's say that you used your favorite painting application to create some pictures which will eventually be pasted in your word processing application to complete a document. The document is composed of its own data (the text you type in) and the pictures pasted into the document. But in what form is the word processing application using the pictures and how does the application know how to draw them?

When it gets the data from the clipboard, the word processing application has two choices. It can either directly support the native format of the image (meaning the format that the painting application uses to copy and paste its own data), or it can treat the data as one of the generic clipboard formats (such as a metafile or device dependent bitmap). If the application chose the first choice, it has to be programmed to understand the format for every single painting application in existence (and then some). Keep in mind that I've only mentioned painting applications. If the application wants to support other kinds of application, it would have to support the formats of those applications as well.

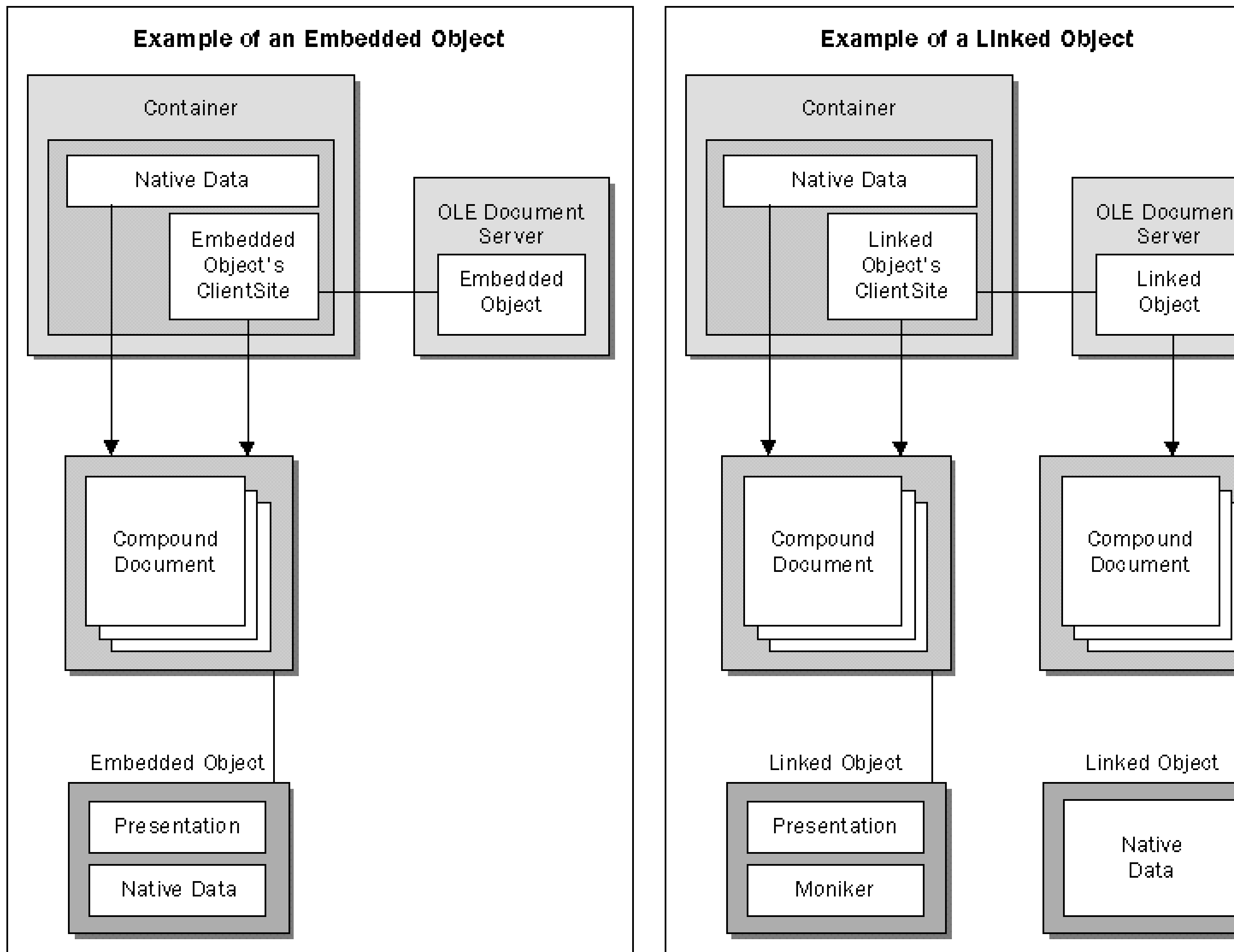
The second choice seems more reasonable, since we could treat data that the user has placed on the clipboard in a very generic manner, but the data loses its roots when you do this. Once it's in a metafile or bitmap format, there's no way to get it back to its original format for further editing. This brings us to the third choice that applications have today: the **OLE Documents** technology.

OLE Documents allows an application to act as a **container**. Containers support the concept of an OLE compound document by providing a page on which we can layout our content objects (or components as we will often refer to them). The components consist of more than just the data of the other application; they also contain information that links them back to the application that supports the data natively. The application that creates the data natively and exposes it to other applications through the OLE Documents interfaces is known as the component **server**.

In our example above, the painting application is our component server, and the word-processing application is the container. The container allows the user to type in the text of the document (which is what the word processing application does best), and then bring in pictures as components which can be placed into the document in one of two ways, either **linking** or **embedding**.

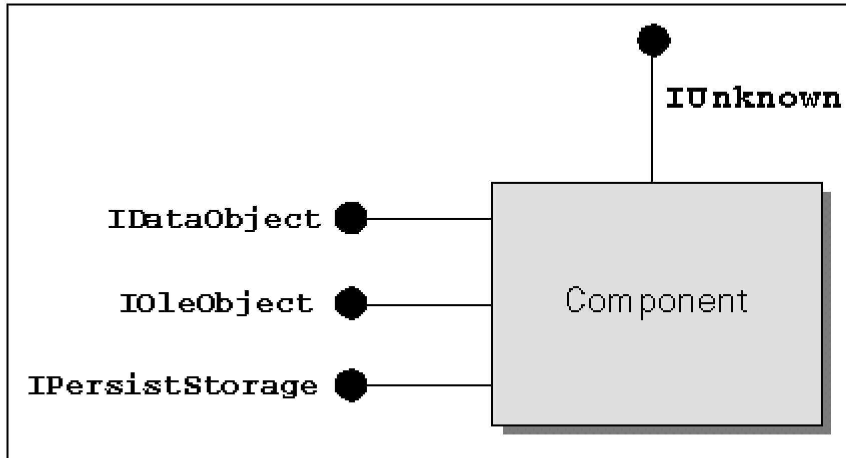
An **embedded object** maintains its native data in the same compound file as the container application. A **linked object** maintains its native data in a different file, but it stores a **moniker** in the container's compound file, along with **presentation data**. The moniker names the data source and tells the component where to locate the data when the user wants to edit it. The component uses the presentation data in order to draw itself.

The component (or content object) is assigned a storage to use for saving its streams of data (we discussed streams and storages in the last chapter). One of the component's streams will contain the presentation data, which will be used when the component needs to be drawn. Here, you can see how linked and embedded objects relate to a container:



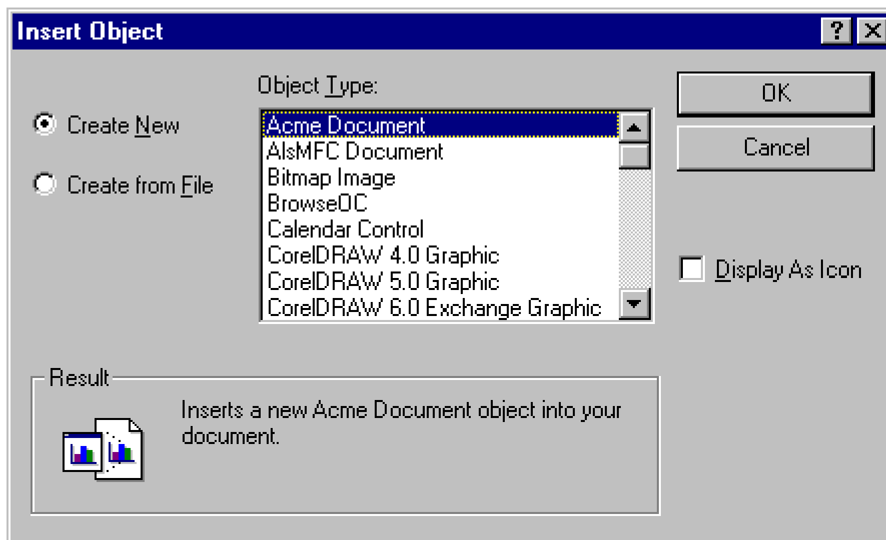
In the beginning, OLE stood for *Object Linking and Embedding***, and really concentrated on the technology now known as** ***OLE Documents*****. Now that OLE covers a wider spectrum of technologies, including OLE Controls and OLE Automation, its original meaning has been lost. It's just OLE.**

There are several interfaces which the component must support so that it can be embedded into a compound document. The figure below shows the interfaces which must be implemented by the component object. In addition to these, the component also needs some interfaces to support drawing itself and managing the presentation cache, but these interfaces are normally supplied by the OLE libraries which provide a generic implementation.



You'll be familiar with the `IDataObject` and `IPersistStorage` interfaces from the last chapter, where we saw them in the context of Uniform Data Transfer and Structured Storage. OLE Documents builds on these foundations, so an understanding of that technology is crucial.

Thanks to these interfaces, containers can communicate with objects in a very generic manner. Even the process of obtaining an object to begin with is done generically. There are many ways to obtain the object; for example, drag-and-drop, via the clipboard, or from the Insert Object common dialog which uses information stored in the registry to offer the user a selection of insertable components.



MFC Support

A large proportion of MFC's support for OLE relates to containers and servers. There are classes for sending information back and forth, using Uniform Data Transfer, there are classes for Structured Storage and Automation, but, without doubt, the biggest bed of code is for OLE Documents. It's a shame that a lot of developers I speak to will tell me that they have no need for OLE Documents and end up using only the OLE Automation and OLE control classes. I feel they haven't looked hard enough at the benefits of

OLE Documents, and by the time you've finished reading this chapter, I hope you'll start to see what I mean.

During the course of the next few sections, I'll show you some interesting and useful ways to apply this technology.

MFC has complete support for OLE embedding and linking documents, but, sometimes, the application that we're writing might just not need everything that Microsoft has provided. In such a situation, you definitely need an understanding of what's going on inside the classes, so that you can make an intelligent decision as to whether you can simply inherit from a class and change certain functionality, or if you'll need to come up with your own concoction from scratch.

A Full Server for OLE Documents

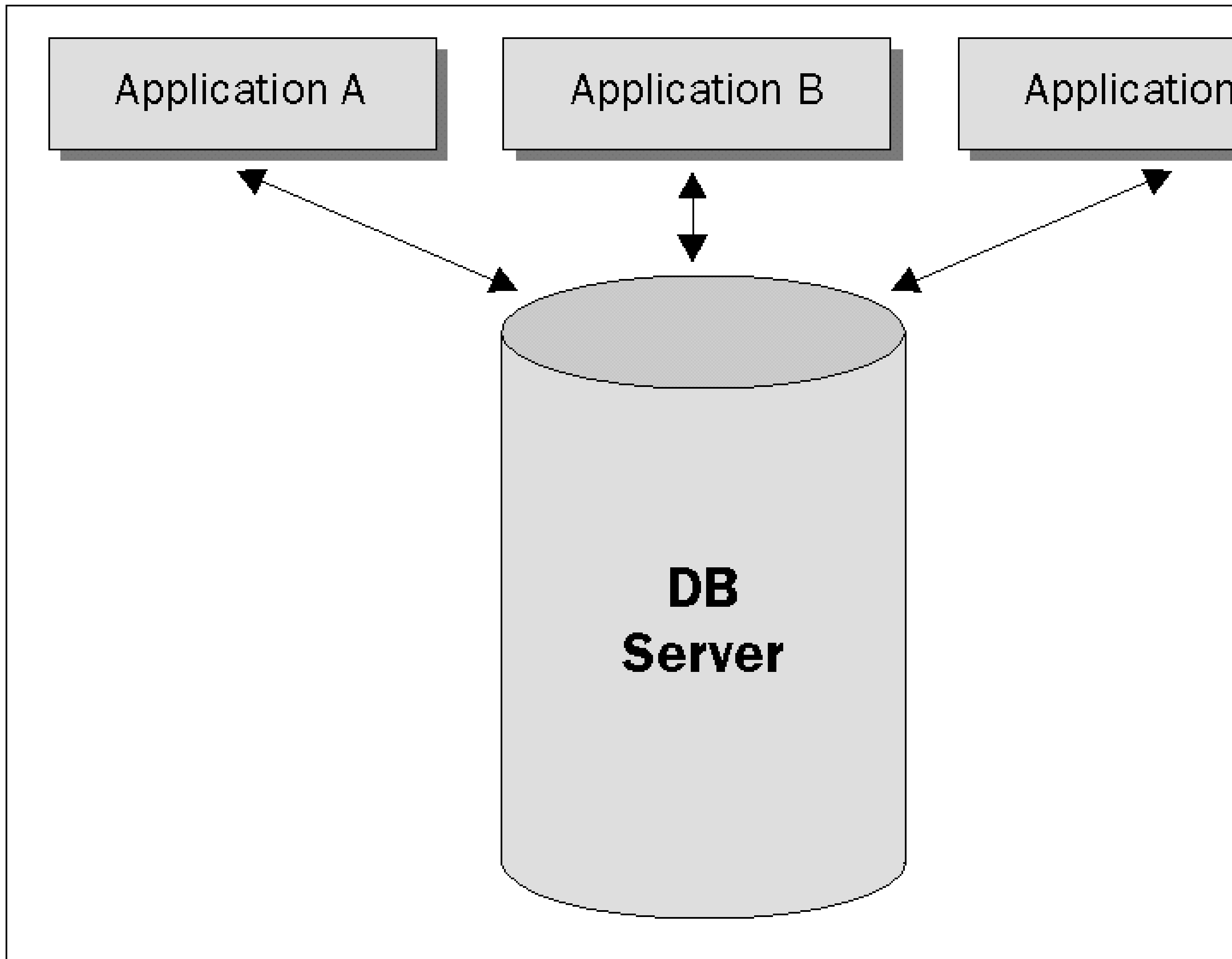
In the previous chapter, you learned about some of the technologies utilized by OLE Documents. Armed with this knowledge, we can start to work with OLE document containers and servers, but one of our primary tasks is to find a real use for this technology.

When I first started learning about the OLE technologies, I always asked myself, "This is great, but where can I use it?" Now that I write about OLE and teach it to others, I'm often asked the same question.

A technology like OLE Documents can be easy to implement in applications like painting programs, word processors, or spreadsheets, but when you're trying to implement it in a corporate environment, supporting a database application, sometimes it's a little more difficult.

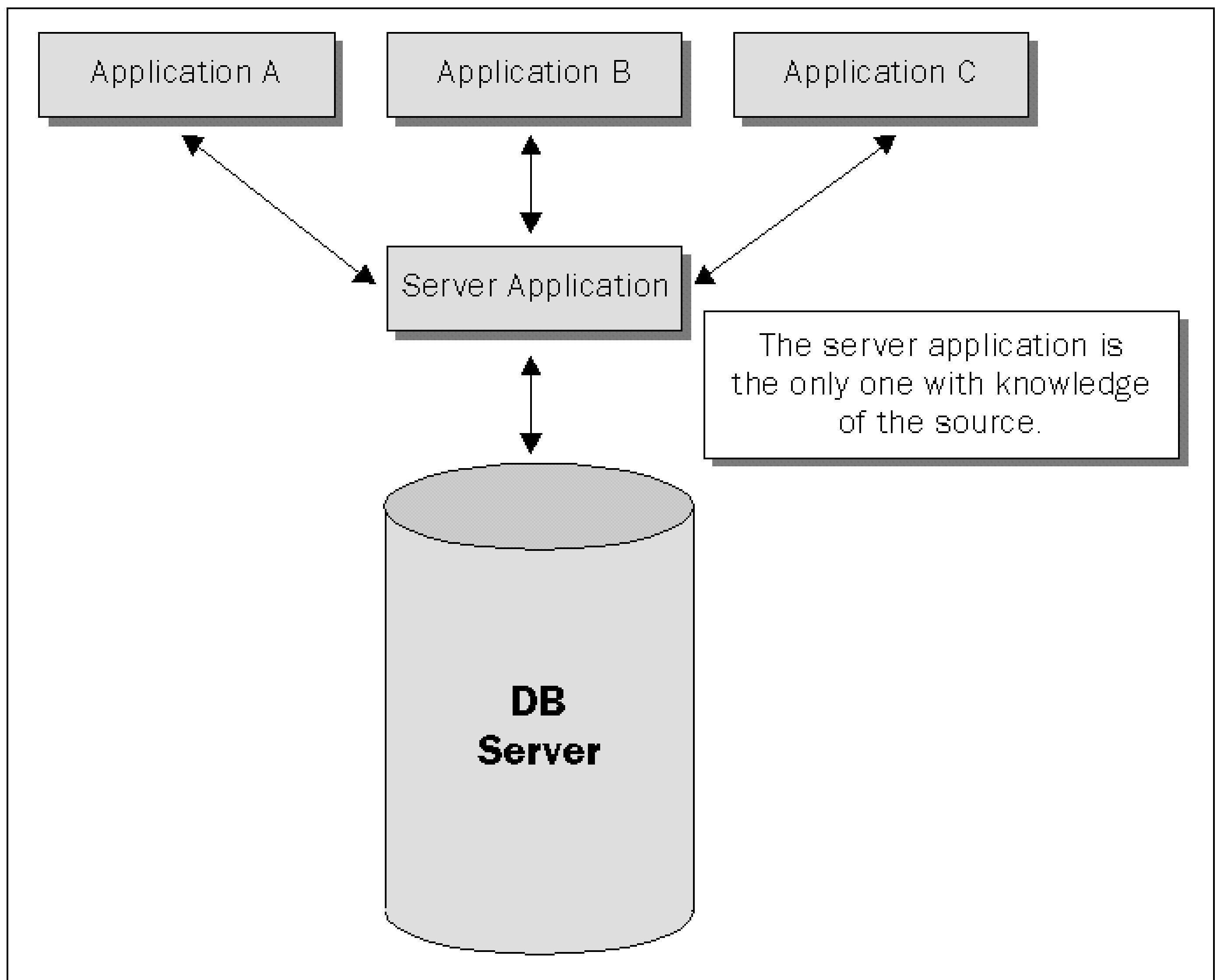
Having said that, though, I've recently started to see the light at the end of the tunnel. Users in a corporate environment often have several applications that need to share a common source of data, stored on a database server somewhere across a local area network. Programmers are often hired to provide a way to retrieve the information and display or manipulate it. The average application must access a database server, gather data from the server, present it to the user, allow the user to manipulate the data, and finally save the data back to the server. If the user has to perform several different tasks, the chances are that they will have to use several applications.

Having to add knowledge of the data source into each application makes them more complicated and also accounts for the many client/server applications that never meet their deadlines.

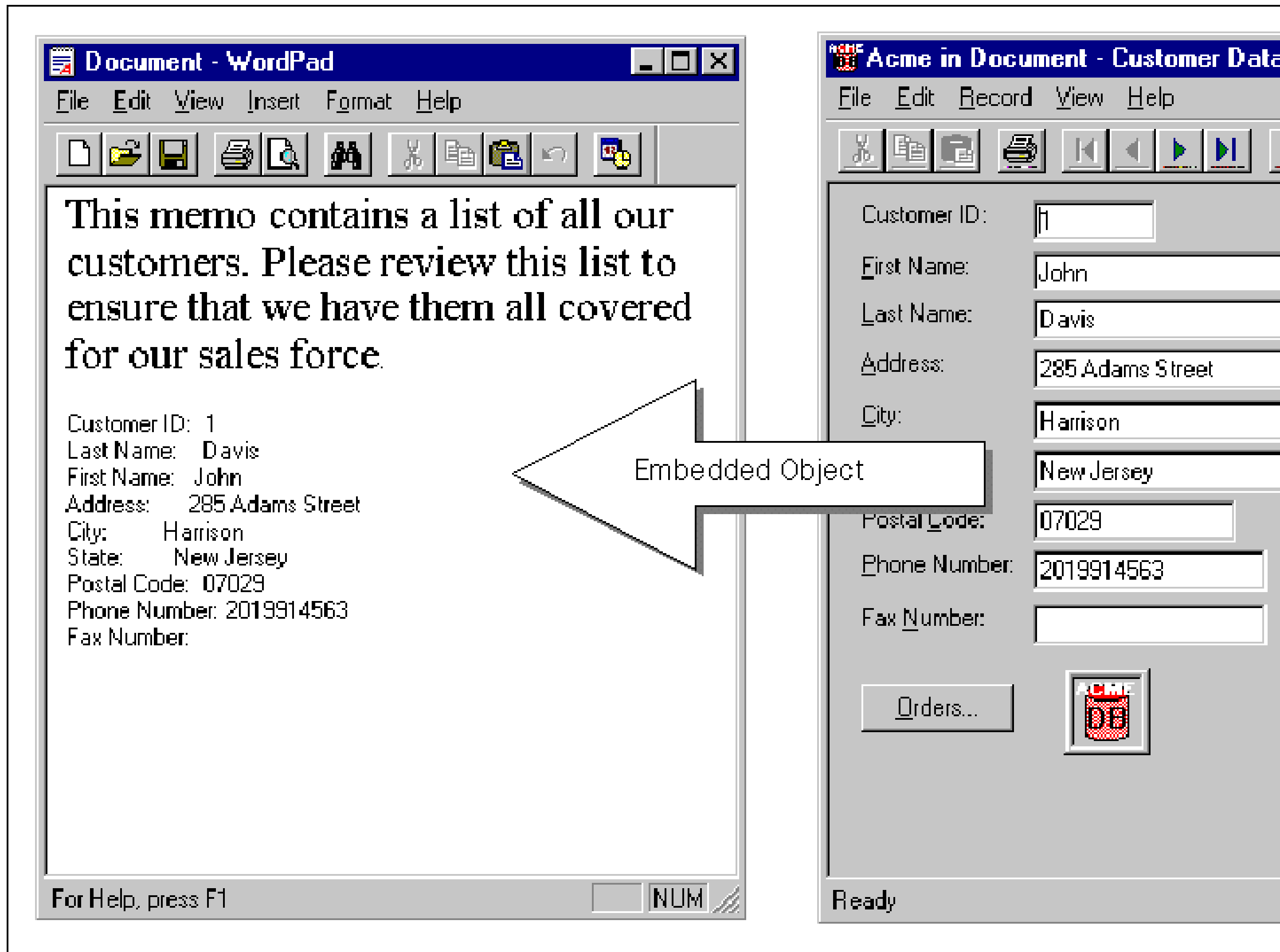


The perfect solution would be to have a single application that contains the knowledge to access and manipulate the data source. This server application would provide an interface to allow other applications to use or contain the information provided by the server in their own documents. Furthermore, if this server application uses OLE Documents technology, rather than a completely custom solution, it won't be limited to use with custom containers; we could use any OLE container application (such as Word or Excel) and still gather information from the database server via the OLE server application.

This would also make it very easy to replace the server application (perhaps with one that accesses the data from a different source) and the container application(s) would have no knowledge that anything has changed. This is component programming at its best.



For example, your users may want to type a letter using Microsoft Word and include all the names stored in a customer database. Furthermore, if the address on the database changes, they'll want this change to be reflected the next time that they print out the letter. We'll see exactly how we can create a server to provide this functionality in the next section.



Adding OLE Server Support to an Existing Application

Turning an existing database-aware application into an OLE server is easy, as you'll see in this section, where I take an existing database application, written using MFC, and turn it into an OLE server. The application uses the MFC ODBC (Open Database Connectivity) classes to communicate with an Access database. Since I'm using ODBC, the underlying database could have been anything (provided I have an ODBC driver for the database), including a LAN-based database such as SQL Server.

The application provides a user interface to a customer database with several customers. The user can select a customer and view the associated order records by pressing the `Orders` button on the main screen.

Basically, I wanted to allow the application's data objects (the customer records) to be embedded into any container application the user chooses (just as in the figure above). The user can then save the container's document to a compound file for later use. If the user wants to change the embedded customer object, they simply double-click on the customer object in the container's window, and the server application is

reopened with the appropriate record selected. The user can then change the record to another customer, and the embedded object in the container reflects this change.

> **You can find the application I started with on the CD in the `Acme` directory. You can also find the application I ended up with on the disc in `AcmeSrv`. Note that to use the examples, you should register the database files as ODBC data sources. There are full instructions for this in the file `AcmeSrv\Instructions.txt`.**

Preparing the Application for OLE Support

The first thing I had to do was add the default OLE support to our non-OLE MFC application, starting with the **`Stdafx.h`** file. Here, I needed to include the header file containing OLE functions and classes, which is the **`Afxole.h`** file:

```
#include <afxole.h>  // MFC OLE classes
```

All OLE applications must initialize the OLE DLLs at startup. You can do this from the top of **`InitInstance()`** with one function call to **`AfxOleInit()`**:

```
// Initialize OLE libraries
if (!AfxOleInit())
{
   AfxMessageBox(IDP_OLE_INIT_FAILED);
   return FALSE;
}
```

I also needed a class-factory; one that creates an instance of the document, view and frame window when it's told to create an object. The class in MFC that will do this for us is **`COleTemplateServer`**. I first added a **`COleTemplateServer`** member, **`m_server`**, to my **`CWinApp`**-derived class and then connected it to the document template inside **`InitInstance()`**:

```
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
   IDR_MAINFRAME,
   RUNTIME_CLASS(CTestSrvDoc),
   RUNTIME_CLASS(CMainFrame),  // main SDI frame window
   RUNTIME_CLASS(CTestSrvView));
pDocTemplate->SetServerInfo(IDR_SRVR_EMBEDDED);
AddDocTemplate(pDocTemplate);

// Connect the COleTemplateServer to the document template.
// The COleTemplateServer creates new documents on behalf
// of requesting OLE containers by using information
// specified in the document template.
m_server.ConnectTemplate(clsid, pDocTemplate, TRUE);
```

Notice the **`clsid`** value passed to the **`ConnectTemplate()`** function. Just like the other code above, this value would usually be created in an application by AppWizard, but since I was adding OLE support to an existing project, I had to add it myself using the **`Guidgen.exe`** application to generate a unique identifier. The **`clsid`**, along with some other information, will need to be placed in the registry so that container applications can use it to create instances of our object. This means that the Insert Object dialog, and therefore the container applications, don't need to have any prior knowledge of our COM object.

When I call **`SetServerInfo()`**, this tells the framework that we're going to use different resources (such as menus or icons) when the application is launched as a server (as opposed to being launched stand-alone). **`IDR_SRVR_EMBEDDED`** identifies the resources. In my case, I provided a different menu (which we'll

see later in this section).

OLE servers written with MFC will expose the document template as the embeddable object, and since we want to have the framework create a document, view and frame for us, we'll use the **clsid** to associate the class factory with our template. This is the **clsid** that I generated with the **Guidgen** application:

```
// {B3FC9600-5AFB-11CF-8208-08000996A1CC}
static const CLSID clsid =
{ 0xb3fc9600, 0x5afb, 0x11cf, { 0x82, 0x8, 0x8, 0x0, 0x9, 0x96, 0xa1,
 0xcc } };
```

Next, we register the class factory with OLE, using a **static** function in the **COleTemplateServer** class:

```
// Register all OLE server (factories) as running.
COleTemplateServer::RegisterAll();
```

Finally, we need to make our server a self-registering application. This means that the application will automatically register itself with the system registry whenever it's run as a stand-alone application. MFC has built-in support for providing the registry with all the appropriate information needed for a server. You simply call one function to perform this registration step:

```
// When a server application is launched stand-alone, it is a
// good idea to update the system registry
m_server.UpdateRegistry(OAT_SERVER);
```

OAT_SERVER identifies the application as a server process.

Changes to the Document Class

The document needs to be able to keep a list of items for servicing containers. When a container application creates an object from our server, the document will create an item which will handle communication with the container application. In MFC, there's a class, called **COleServerItem**, that handles the communication for us and implements several OLE interfaces necessary for communicating with the container application, such as **IDataObject**, **IOleObject**, and so on.

When AppWizard generates an OLE server application for you, it usually creates a header file and an implementation file containing a derived class of **COleServerItem**. However, since I added the OLE stuff to my application after the fact, I had to create the file myself. In fact, I actually ended up creating a temporary project with OLE support, then moving these files from the temporary project's directory to my project's directory. Then, I simply added the files to my project.

I needed to code several minor functions in these files. For starters, the container application will request a rendering of the data, and MFC will oblige by calling the **OnDraw()** function of the server item. There's a catch here. The device context that is provided to the server item is, in fact, a metafile. As we all know, you can't query a metafile for information in the same way that you would a regular device context. That's why I usually create a **CClientDC** with no window. I then use the client device context to query it for information and use the information for writing to the metafile. This will work because both the metafile and the device context are set up to use the same mapping mode, extents and origins.

In the sample application, I needed information about the text that I wanted to write into the metafile, including the vertical height and the horizontal width that the text would take up. I wrote a helper function, called **CalculateSize()**, which would return the information in a **CSize** object. Here's what the

code looks like:

```
CSize CAcmeSrvrItem::CalculateSize()
{
   CAcmeDoc* pDoc = GetDocument();

   CClientDC dc(NULL);
   dc.SetMapMode(MM_ANISOTROPIC);
   dc.SelectStockObject(ANSI_FIXED_FONT);

   CStringArray& data = pDoc->GetAcmeData();

   // Find the widest record.
   int nCount = data.GetSize();
   CSize size = dc.GetTextExtent(data.GetAt(0),
                              data.GetAt(0).GetLength());

   CSize sizeNew;
   if (nCount > 1)
      for (int i = 1; i < nCount; i++)
      {
         sizeNew = dc.GetTextExtent(data.GetAt(i),
                                 data.GetAt(i).GetLength());
         if (sizeNew.cx > size.cx)
            size.cx = sizeNew.cx;
      }

   dc.LPtoHIMETRIC(&size);
   return size;
```

The function determines the height of one string by using the height returned from **GetTextExtent()** then it calculates the width by finding the longest string in the bunch.

CalculateSize() is called from several places in the application when information is needed about the text to be drawn. When the container initially embeds the object into its document, it might want to know how much screen real estate it should try to allocate in its window. The container will call the object's **IOleObject::GetExtent()**, which results in a call to the server item's **OnGetExtent()** function. My implementation calls **CalculateSize()**, multiplies the height returned by the number of strings and returns the result. When you pass sizes across applications, OLE requires these sizes to be in **MM_HIMETRIC** for the sake of uniformity.

Now let's take a look at the **OnDraw()** function in the server item. It sets the mapping mode and extents for the metafile to use, and begins the process of painting the strings one by one. Here's the code:

```
BOOL CAcmeSrvrItem::OnDraw(CDC* pDC, CSize& rSize)
{
   CAcmeDoc* pDoc = GetDocument();

   CStringArray& data = pDoc->GetAcmeData();

   // TODO: set mapping mode and extent
   // (The extent is usually the same as the size OnGetExtent)
   CSize size;
   OnGetExtent(DVASPECT_CONTENT, size);

   pDC->SetMapMode(MM_ANISOTROPIC);
   pDC->SetWindowOrg(0,0);
   pDC->SetWindowExt(size);

   pDC->SelectStockObject(ANSI_VAR_FONT);
   int nCount = data.GetSize();
   int nPos = 0;
   for (int i = 0; i < nCount; i++)
   {
```

```
            pDC->TextOut(0, nPos, data.GetAt(i));
            nPos += m_cyHeight;
        }

        return TRUE;
    }
```

For all the details, study the **Srvritem.h** and **Srvritem.cpp** files. Whenever a server item is created, it requires that you pass it the **this** pointer of the document object that owns the server item. The server item will then add itself to the document's list of server items.

The most important steps in creating the OLE server were to derive my document class from MFC's **COleServerDoc** class and to override the **OnGetEmbeddedItem()** function:

```
COleServerItem* CAcmeDoc::OnGetEmbeddedItem()
{
   // OnGetEmbeddedItem is called by the framework to get the
   // COleServerItem that is associated with the document.  It
   // is only called when necessary.
   CAcmeSrvrItem* pItem = new CAcmeSrvrItem(this);
   return pItem;
}
```

Note that if a document has already created the server item for an embedded object, the frame work won't call **OnGetEmbeddedItem()**, since it holds onto the server item that is returned from **OnGetEmbeddedItem()**.

This function is the workhorse of this whole operation and it's actually how the item server is created to begin with. When the client application chooses to embed a new object into its document, our server's document is called upon to deliver an object. This object must support the appropriate interfaces and data formats for it to be properly embedded into the container application. MFC will call the server document's **OnGetEmbeddedItem()** function to provide the server object.

From within the server application, other modules can make use of the object by calling the document's **GetEmbeddedItem()** function. This code exists to provide a type-safe path to the embedded item, although I don't completely agree with Microsoft's implementation, since it seems to be a re-implementation of a non-**virtual** function.

The function actually calls the **GetEmbeddedItem()** function in the base class, but the version in the base class returns a generic type, **COleServerItem** to be exact. The base class checks to see whether a server item has been created already and simply returns the same item. Otherwise, it calls down to **OnGetEmbeddedItem()** in the derived document class, which creates the server item and returns it. You can see all this in the code below:

```
class CAcmeDoc : public COleServerDoc
{
// Attributes
public:
   CAcmeSrvrItem* GetEmbeddedItem()
      { return (CAcmeSrvrItem*)COleServerDoc::GetEmbeddedItem(); }
   // ...and so on
}

class COleServerDoc : public COleLinkingDoc
{
// Attributes
   COleServerItem* GetEmbeddedItem();
   // ...and so on
```

```
}

COleServerItem* COleServerDoc::GetEmbeddedItem()
{
   // allocate embedded item if necessary
   if (m_pEmbeddedItem == NULL)
   {
      m_pEmbeddedItem = OnGetEmbeddedItem();
      m_pEmbeddedItem->ExternalAddRef();
   }

   return m_pEmbeddedItem;
}

COleServerItem* CAcmeDoc::OnGetEmbeddedItem()
{
   CAcmeSrvrItem* pItem = new CAcmeSrvrItem(this);
   return pItem;
}
```

This brings us to an interesting observation. When a server application is working on an object that has been embedded into a container, each document can only work with one object at a time. Therefore, if the application is using an SDI model, each embedded object must be serviced by a different instance of the server. For an MDI server, each MDI child window must pertain to an embedded object. Try it with any server application that you can think of and you'll see what I mean.

The document's **Serialize()** member function will be called from the item object whenever the container requests that the embedded data object be placed on the clipboard, or when the container calls the object's **IPersistStorage::Save()** function. In the **AcmeSrv** example, I took the liberty of passing the Customer ID for the current record to the container. This way, next time the container asks us to load ourselves (again, via the **Serialize()** member function), we can simply look through our records from the database server for a match and display the appropriate record in our view window. This is what the **Serialize()** function in the document looks like:

```
void CAcmeDoc::Serialize(CArchive& ar)
{
   if (ar.IsStoring())
   {
      ar << m_acmeSet.m_CustomerID;
   }
   else
   {
      long ID;
      ar >> ID;

      // ... Code to locate the appropriate record for displaying.

      UpdateAllViews(NULL);
   }
}
```

Resource Changes

I had to make a couple of changes to the resources. First of all, I needed to add a string to the string table with an ID of **IDP_OLE_INIT_FAILED** and text "OLE initialization failed. Make sure that the OLE libraries are the correct version.". Secondly, I needed to add a new menu resource with the ID of **IDR_SRVR_EMBEDDED** to be used when the user is editing an Acme document embedded in a container. This menu is the same as the existing menu, except that it also has an extra item in the File menu, Update, with an ID of

ID_FILE_UPDATE.

Adding Drag-and-drop Support

The last thing I did to my application was to add drag-and-drop support. If you select the icon on the main window and drag it to a container application, you'll see the powerful drag-and-drop support that MFC provides.

All of the code for the drag-and-drop operation is implemented in the view class. Since all I have on the screen is a picture control displaying an icon, I needed some way to detect when the user held the mouse button down on the icon and dragged it, so that I could begin the drag-and-drop operation. I did this by testing the mouse in the **WM_SETCURSOR** handler, as follows:

```
BOOL CAcmeView::OnSetCursor(CWnd* pWnd, UINT nHitTest, UINT message)
{
   if (message == WM_LBUTTONDOWN &&
      pWnd->m_hWnd == GetDlgItem(IDC_DRAGSOURCE)->m_hWnd)
   {
      m_bBeginDrag = TRUE;
      PostMessage(WM_LBUTTONDOWN, MK_LBUTTON, 0L);
      return TRUE;
   }

   return CRecordView::OnSetCursor(pWnd, nHitTest, message);
}
```

I found that if I wanted to use the MFC classes for drag-and-drop, I had to be within the context of a **WM_LBUTTONDOWN** or a **WM_RBUTTONDOWN** message before calling the **DoDragDrop()** function. So, once I know that I've got a hit, I post a **WM_LBUTTONDOWN** message to the view to simulate a mouse click. I then respond to that **WM_LBUTTONDOWN** message and begin the drag-and-drop operation.

The **COleServerItem** class also has a function called **DoDragDrop()**, which works very similarly to the **COleDataSource** class we saw in the last chapter. As a matter of fact, the server item creates a **COleDataSource** object internally and calls its **DoDragDrop()** function. Here's the code for the **WM_LBUTTONDOWN** override:

```
void CAcmeView::OnLButtonDown(UINT nFlags, CPoint point)
{
   if (m_bBeginDrag)
   {
      CAcmeDoc* pDoc = GetDocument();
      COleServerItem* pSrvrItem = pDoc->GetEmbeddedItem();
      pSrvrItem->DoDragDrop(CRect(0, 0, 1, 1), CPoint(0, 0),
         FALSE, DROPEFFECT_COPY);

      pDoc->SetModifiedFlag(FALSE);
   }
   m_bBeginDrag = FALSE;
   CRecordView::OnLButtonDown(nFlags, point);
}
```

Wrapping up the Server

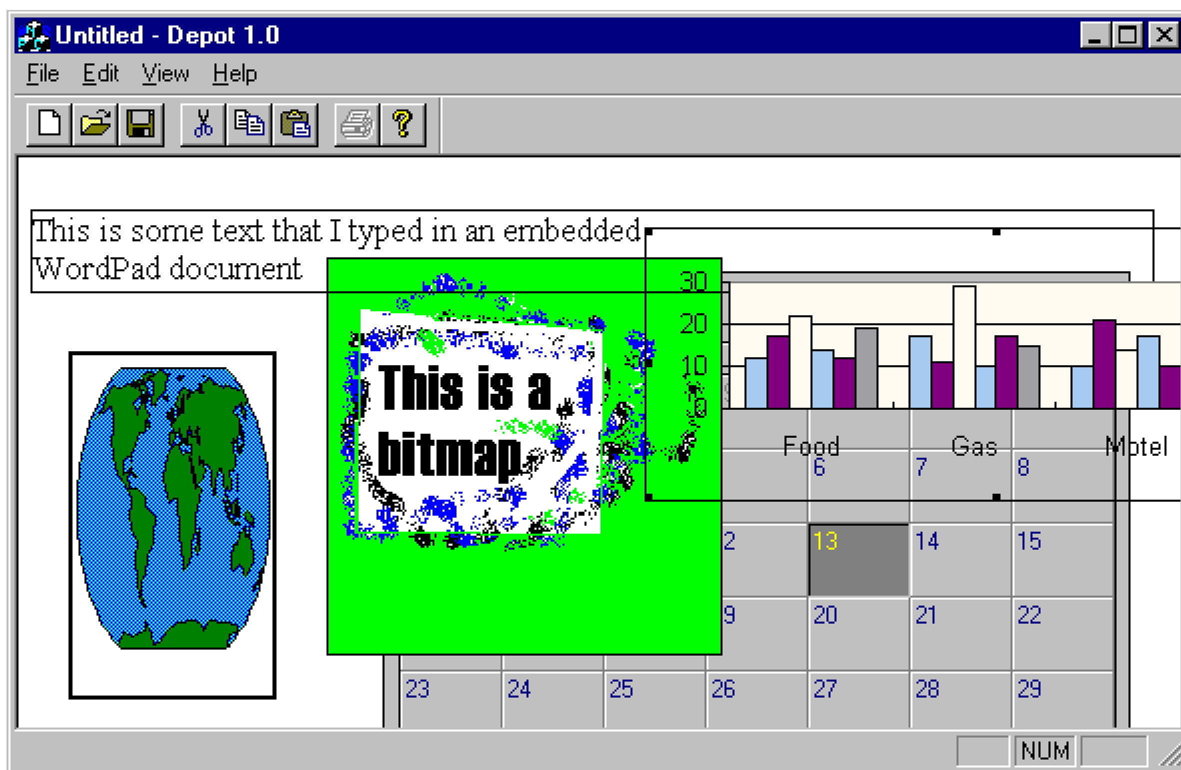
As I mentioned, I've included both versions of the application (both before and after adding OLE support) on the CD. The before picture is in a directory called **Acme**, and the after picture is in **AcmeSrv**. Before you try to insert it into a container application, don't forget to run the application as a stand-alone process to register it as a document server.

When you're ready to insert embeddable objects into a container, a good place to start is WordPad, which ships with Windows 95 as an accessory. You can also find the code for WordPad as part of the samples that ship with MFC. Simply run one of the applications, bring up the Insert Object dialog box and go for it. You'll see the Acme object listed as Acme Document. Also try dragging and dropping an object into the container application just to see that it works.

Implementing a Container

In OLE Documents, the container plays the role of a depot, storing objects regardless of what they are. As far as the container is concerned, the data is unstructured, which means that it knows nothing about the objects contained within its documents. It doesn't have to, since the objects are smart enough to know how to save and load themselves, draw themselves and activate themselves for editing.

We'll look at an example called `Depot`, an MFC AppWizard-generated application. I added a few lines of code to complete the OLE document container support necessary for a real container application. However, the application doesn't provide any native functionality of its own. Its sole purpose is to store all kinds of objects, as a depot stores all kinds of things. It uses the MFC support for compound documents to store its objects persistently.



Later in this chapter, we'll convert the file support to store the objects persistently to a database. (You read correctly, a *database*!) We're going to use the ODBC classes to read and write records to the Acme database.

The Benefits of Compound Documents

In the old days, we could copy text and pictures to the clipboard and then paste those objects into a cooperating application. The problem was that, because the objects didn't carry any information about where they were created, the user would have to remember which application created them, and would have to recreate the object in order to modify it. There was no way to reactivate the object's application so that the object could be changed.

Copying the data back to the application that created it was not always possible, since, in many cases, the native data was lost, and the only data that existed was a bitmap representation of the object. The application that created the object originally couldn't make heads or tails of the bitmap representation and, therefore, couldn't allow the original data to be edited.

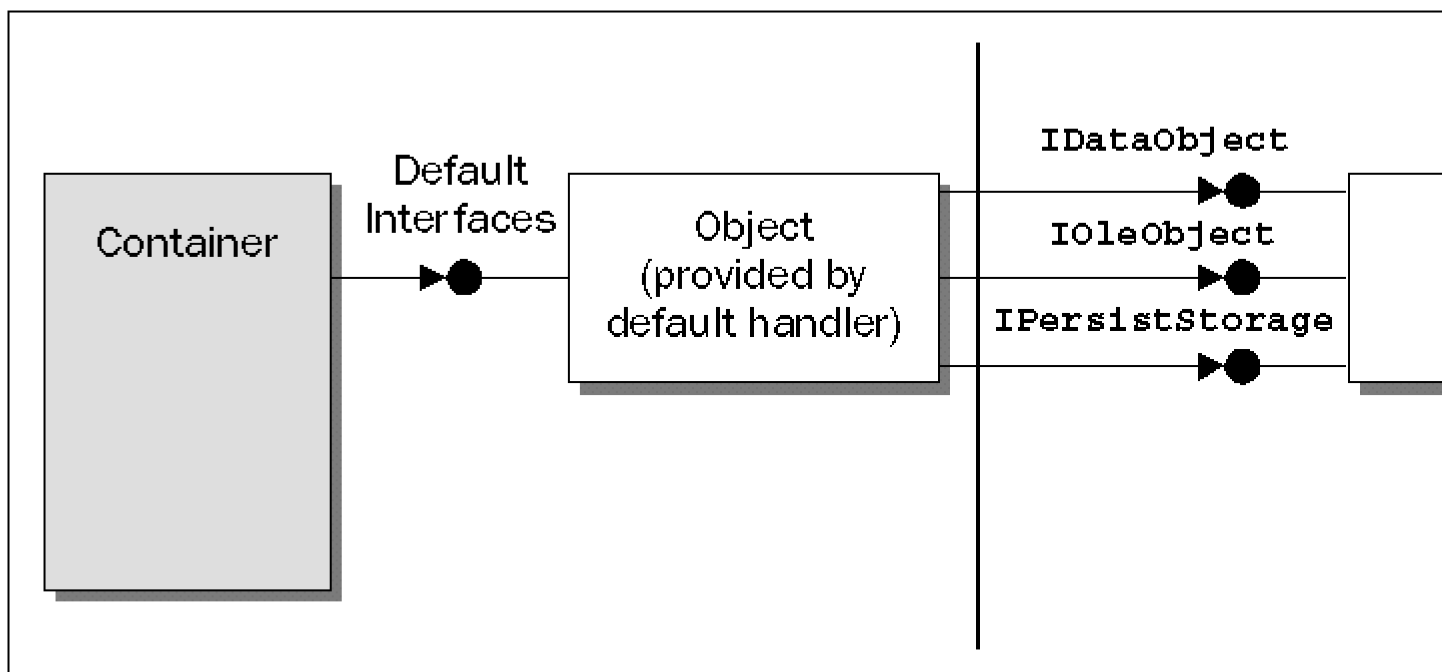
With OLE Documents, content objects store information about their server (in particularly, the CLSID) so that they can be reactivated for editing the original data at a later point. An embedded object must support several interfaces: **`IDataObject`**, **`IOleObject`**, **`IViewObject2`**, **`IOleCache2`**, **`IRunnableObject`**, **`IPersistStorage`**, and optionally, **`IOleCacheControl`**. There are also additional interfaces which must be supported for in-place activation and linking.

A container will make use of these interfaces for communicating with the embedded objects. Since these interfaces are part of a standard specification then once they have been implemented with the expected functionality, both the container and the objects can communicate in a very generic manner.

Default Handler

Any content object that you create must provide implementations for at least **`IDataObject`**, **`IOleObject`**, and **`IPersistStorage`**. It can rely on OLE's default handler for the other interfaces. OLE will create an object which aggregates the content object and provides the other interfaces for the object. The object is known as the default object and its implementation sits inside the default handler, which is registered with the server in the registry. You can also provide your own handler and aggregate on the default handler for some of the interfaces. Note that some interfaces, such as **`IViewObject2`**, must be in an in-process server (in this case, because device context handles can't be marshaled across process boundaries).

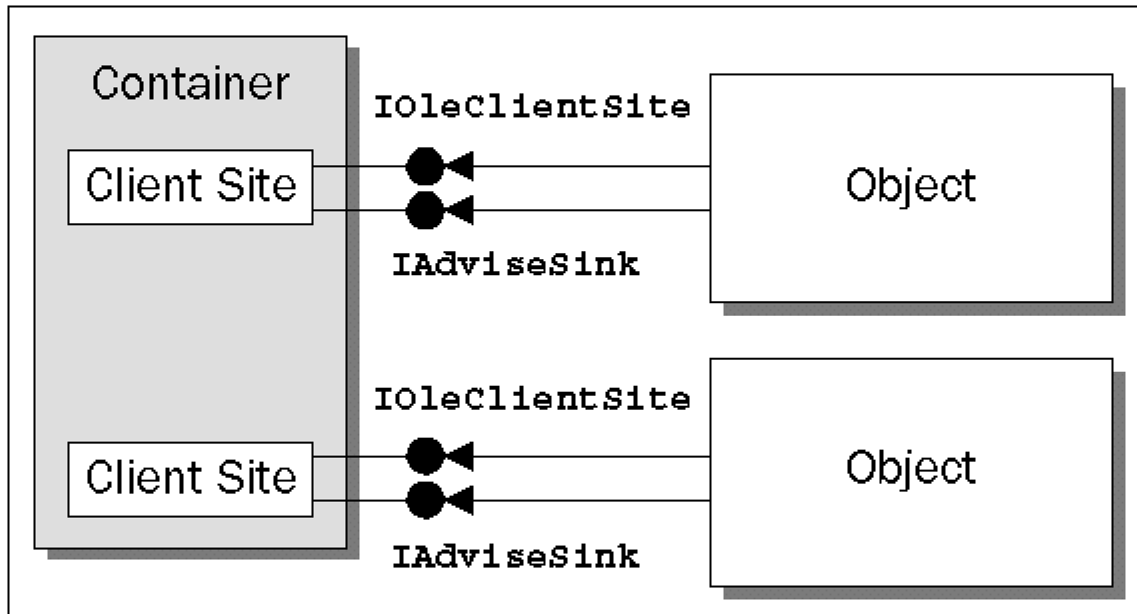
`IDataObject`s exist so that the container or other interfaces can ask the objects for data in different formats. As we saw in the last chapter, an object implementing **`IPersistStorage`** can be asked to persistently save or load itself from a given storage (**`IStorage`**) provided by the container. The figure below illustrates how the container, default handler and the content object server interact:



The default object's **`IOleCache2`** interface will fetch renderings from the content object via its **`IDataObject`** interface and store them in streams inside the object's storage in the compound file provided by the container. Next time the object has to be loaded and drawn, the container will most likely call **`OleLoad()`**, which will create a default handler object and assign its **`IPersistStorage`** interface the given **`IStorage`**. When the container wishes to draw the object, the default object will load one of the renderings and draw the rendering by calling **`IViewObject2`**'s **`Draw()`** function, which renders the object to a given device context provided by the container. This mostly goes on underneath the covers. MFC and just about every OLE application that supports OLE Documents out there performs the same tasks.

Container Interfaces

The container needs to provide an object for each content object provided by the server. These **client site** objects implement several interfaces, including **`IOleClientSite`** and **`IAdviseSink`**. The content object will then use these interfaces to send notifications and request information from the container.



MFC makes handling the client site interfaces a piece of cake. It actually provides a class, called **COleClientItem**, to perform most of the work for us. This class implements the **IOleClientSite** and the **IAdviseSink** interfaces. If you derive a class from **COleClientItem**, you can override several **virtual** functions which are called when the interface functions are called by the associated content object. For example, when the **IAdviseSink::OnDataChange()** function is called, the framework calls **COleClientItem::OnChange()** with an appropriate notification of the event that occurred.

When you generate an application with AppWizard and you select the option that allows your application to become a container, AppWizard will create a class derived from **COleClientItem**, automatically. Most of the default functionality of the class is great, but there are some areas where we can improve it.

Before we get to these improvements, let's look at the code generated by AppWizard. You can find all the code for this example on the CD in the **Depot** directory.

The Skeleton Code

Just as we saw with the OLE server, the application class must always call **AfxOleInit()** in **InitInstance()** in order to initialize the application with the OLE libraries. It also calls **CDocTemplate::SetContainerInfo()** to assign the resources for the menu and accelerator that should be used when the objects are being edited in place. The resources are called **IDR_CNTR_INPLACE**.

The view class is given a data member, called **m_pSelection**, to hold a pointer to the currently selected item. In our example, it's a pointer to a **CDepotItem** (which is the class derived from **COleClientItem**). When an item is selected, this member will point to the item; otherwise, it will contain **NULL**.

As for the view's **OnDraw()** function, AppWizard assumes that all you want to draw is the item pointed to by **m_pSelection**. This is one of the things that we will change.

The view's **IsSelected()** function is called when a part of the application wants to compare a given item

against the currently selected item to verify whether the item is, in fact, selected. The current implementation simply returns **TRUE** or **FALSE** and works well for containers that allow only one item to be selected at a time. Containers that wish to allow more than one object to be selected at a time will have to modify this code, but I didn't have to, since I only allow one object at a time to be selected.

AppWizard generates a message response function, **OnInsertObject()**, in the view for the Edit/Insert menu item that responds by prompting the user with the Insert Object dialog box. When the user selects the object they wish to insert, the default code creates and initializes a **COleClientItem** object, makes it the currently selected item and updates the view. The document class maintains a list of all the items inserted into the document. When the item is created, it is passed the document's **this** pointer, which it uses to call the document and add itself to the document's list. The Insert Object dialog allows the user to insert a brand new object or insert an object that already exists in another file. If the object is inserted as a new object, it's immediately activated by calling the item's **DoVerb()** function to allow the user to edit the object. Otherwise, the object is simply displayed (but not activated).

The AppWizard-generated code for the view's **OnSetFocus()** passes the focus to the in-place window handling the active object, if an object is, in fact, being edited. This code will work for most containers without any change.

Again, if an item is being edited in-place, we need to alert it of any changes to the window's size. That way, the server can be aware of any clipping problems it might have to deal with. AppWizard generates an **OnSize()** override for the view. The override determines if there's an object being edited in-place and calls its **SetItemRects()** function.

When an item is added to the document (either from **CDepotView::OnInsertObject()**, the clipboard, or via a drag-and-drop operation), a **COleClientItem** is created to handle communication with the OLE content object. When the document is destroyed, so are its client items (but not before they're saved, if the user saved the document).

AppWizard does override several functions of the **COleClientItem** class by providing a derived class; **CDepotItem** in my case. The first function, **OnChange()**, simply calls the base class's **OnChange()** function and then invalidates all the views of the document by calling **COleDocument::UpdateAllViews()**.

When the server is in-place, it will call the client site to find out the position and size of object as it appears in the compound document. The server will then use the position and size to offset its in-place window accordingly. The request for position and size result in a call to the client item's **OnGetItemPosition()** function.

When the server application is editing the object in-place, **COleClientItem::OnChangeItemPosition()** is called whenever the size of the object changes in the server's window. The default implementation calls the base class which in turn calls **COleClientItem::SetItemRects()**.

Adding Additional Support for Containers

The AppWizard generated code for containers is good, but it can still use improvements. First of all, although you can insert objects, they are all placed on top of each other and there's no way to move them around or even show the one currently selected. When it comes time to paint the objects, only one item is ever painted. To improve the container support, we need to modify and add some code to the application .

We'll begin by allowing the objects to maintain their own position. This will allow us to move the objects around and draw them at their correct position. First, we need to add a data member of type **`CRect`** to the **`CDepotItem`** class, which will maintain the current position and size of the item. Then we need to add two helper functions to the class. The first will help to optimize painting by causing the appropriate objects to be invalidated. This function is called **`InvalidateItem()`**:

```
void CDepotItem::InvalidateItem()
{
   GetDocument()->UpdateAllViews(NULL, HINT_UPDATE_ITEM, this);
}
```

`InvalidateItem()` calls the document's **`UpdateAllViews()`** in a very smart way. It passes a hint and its **`this`** pointer to ensure that the area to be invalidated is the area contained by its rectangle. This information is passed to the view's **`OnUpdate()`** function, which can then make use of the information and invalidate the necessary area:

```
void CDepotView::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint)
{
   switch (lHint)
   {
      case HINT_UPDATE_WINDOW:   // invalidate entire window
         Invalidate();
      break;

      case HINT_UPDATE_ITEM:      // invalidate single item
      {
         CRectTracker tracker;
         SetupTracker((CDepotItem*)pHint, &tracker);
         CRect rect;
         tracker.GetTrueRect(rect);
         InvalidateRect(rect);
      }
      break;
   }
}
```

The second helper function for the client item is **`UpdateFromServerExtent()`**, which retrieves the extent of the item in **`MM_HIMETRIC`** coordinates. It then stores the coordinates in device points in **`m_rect`** and invalidates the item. Here's the source code of the function:

```
void CDepotItem::UpdateFromServerExtent()
{
   CSize size;
   if (GetCachedExtent(&size))
   {
      // We need pixels
      CClientDC dc(NULL);
      dc.HIMETRICtoDP(&size);

      // only invalidate if it has actually changed and also only
      // if it is not in-place active.
      if (size != m_rect.Size() && !IsInPlaceActive())
      {
         // invalidate old, update, invalidate new
         InvalidateItem();
         m_rect.bottom = m_rect.top + size.cy;
         m_rect.right = m_rect.left + size.cx;
         InvalidateItem();

         // mark document as modified
         GetDocument()->SetModifiedFlag();
      }
   }
}
```

When the object is activated in-place, the size of the object might change in the server window. This change should be reflected and saved for further painting of the object. The framework will call **CDepotItem::OnChange()** and our current implementation calls the base class. We need to modify the code so that it calls our **UpdateFromServerExtent()** function. Here's what the code should look like:

```
void CDepotItem::OnChange(OLE_NOTIFICATION nCode, DWORD dwParam)
{
   ASSERT_VALID(this);

   COleClientItem::OnChange(nCode, dwParam);

   // When an item is being edited (either in-place or fully open)
   // it sends OnChange notifications for changes in the state of the
   // item or visual appearance of its content.

   switch (nCode)
   {
      case OLE_CHANGED:
         InvalidateItem();
         UpdateFromServerExtent();
      break;

      case OLE_CHANGED_STATE:
      case OLE_CHANGED_ASPECT:
         InvalidateItem();
      break;
   }
}
```

During in-place activation, the **CDepotItem::OnChangeItemPosition()** function will be called by the server to change the position of the in-place object. This might have occurred if the extent of the object in the server has been resized. We need to set our **m_rect** member to this new size and force the object to repaint itself:

```
BOOL CDepotItem::OnChangeItemPosition(const CRect& rectPos)
{
  if (!COleClientItem::OnChangeItemPosition(rectPos))
   return FALSE;

  InvalidateItem();
  m_rect = rectPos;
  InvalidateItem();

  GetDocument()->SetModifiedFlag();

  return TRUE;
}
```

In the code above, I invalidate the old position and size, set the new size and repaint the object. We also might be called to return the position and size. The code AppWizard generated for the **OnGetItemPosition()** returns an arbitrary position and size, so I modified the code to return whatever is in **m_rect**:

```
void CDepotItem::OnGetItemPosition(CRect& rPosition)
{
   rPosition = m_rect;
}
```

The last thing I did to the **CDepotItem** class was to serialize the **m_rect** field by modifying the **Serialize()** member function:

```
void CDepotItem::Serialize(CArchive& ar)
{
   COleClientItem::Serialize(ar);

   // now store/retrieve data specific to CDepotItem
   if (ar.IsStoring())
   {
      ar << m_rect;
   }
   else
   {
      ar >> m_rect;
   }
}
```

The next and last class I had to modify is the **CView**-derived class, **CDepotView**. My first stop was the drawing code which I modified so that it ran through the document's list of items and painted each one at its appropriate position and size. I also made use of the **CRectTracker** class and initialized a **CRectTracker** object by calling a helper function named **SetupTracker()**. Here's the code for **OnDraw()**:

```
void CDepotView::OnDraw(CDC* pDC)
{
   CDepotDoc* pDoc = GetDocument();
   POSITION pos = pDoc->GetStartPosition();
   while (pos != NULL)
   {
      // draw the item
      CDepotItem* pItem = (CDepotItem*)pDoc->GetNextItem(pos);
      pItem->Draw(pDC, pItem->m_rect);

      // draw the tracker over the item
      CRectTracker tracker;
      SetupTracker(pItem, &tracker);
      tracker.Draw(pDC);
   }
}
```

The **SetupTracker()** helper function examines the client item it receives and initializes the tracker with the styles necessary, so that the user will be able to determine the current state of the objects on the screen. If the object is an embedded object, it's drawn with a solid line. If it's a linked object, it has a dotted line. The currently selected object is drawn with resize handles inside the object and, if an object is activated (as an opened object), it's drawn with hatch lines inside it.

I created one other important helper function: **SetSelection()**. This selects the given item and deselects the currently selected item, causing both to be invalidated. It is called from several places in the view.

Finally, I added functionality to the view class for cutting, copying and pasting embedded objects from the clipboard. Copying and cutting to the clipboard was easy. I simply called the currently selected item's **CopyToClipboard()** function. Pasting from the clipboard was only a little more difficult. I had to create a new **CDepotItem**, but instead of asking the dialog box to create the object and associate it with the client item (as I did in **OnInsertObject()**), I had to call **CDepotItem::CreateFromClipboard()**. Next, I retrieved the item's size, selected the item as the currently selected object and invalidated it.

I think the rest of the details are pretty simple to understand from the source code. This will provide a basis for your own container applications. It uses compound files to serialize its data and objects, and provides a solid user interface for containing objects.

Database Support in an OLE Container

Using compound files for storage is fine when your application is a commercial application sold at retail, but what if it's used in the corporate environment? Furthermore, what if your application is using a relational database to store and retrieve its data? In most cases, developers can't find a use for OLE Documents in the corporate environment, because they can't see the point in using compound files. The biggest reason for this is because it doesn't make sense to use files that will sit on someone's hard-drive and can't be shared with other users on different machines. But what if I told you that you could share the data via your back-end database?

If we could make content objects write their streams to a storage in memory, rather than in a compound file, we could then take the bytes in memory and copy them into a database field. When we need to reload the data, it would be great if we could load the bytes into memory and tell the content objects to recreate themselves from the stream of bytes. Guess what? We can actually implement this very idea relatively easily, using MFC and a few overridden functions.

Each database record has a binary field (BLOB) to hold the objects. We store the objects into a database field by first creating an `ILockBytes` on an `HGLOBAL` (global memory) and creating or opening a compound file on the `ILockBytes`. From that point on, any objects, storages and streams created by the framework will actually store the information directly into internal memory. I can then take that memory and store it in the binary field in the database.

Saving Embedded Objects to a Database

I took the Depot application and converted it to save and load its data in an Access database. This database is the same as the one in the Acme server example, so you won't have to register another database to get the new example, `Dbdepot`, working. The database contains one field that wasn't displayed in the previous example—a **long binary field**, called `Embeddings`, that I added to the `Customers` table to store OLE documents related to each customer. I can move from record to record, loading the binary information stored in the records with a `CLongBinary` object.

I used ClassWizard to create a `CRecordSet`-derived class based on the `Customers` table and added a data member of this class to the document class. I then opened the database in the document's constructor as follows:

```
CDbDepotDoc::CDbDepotDoc()
{
   // Use OLE compound files
   EnableCompoundFile();

   // TODO: add one-time construction code here
   m_dbDepotSet.Open();
   m_lpRootStg = NULL;
}
```

Notice that I'm still calling `EnableCompoundFile()`. I do this to trick MFC into thinking that we're going to allow it to save the content objects into the compound file that MFC provides. The key is that, before MFC does any writing to its compound file, we're going to pull the old switcheroo and change its compound file to our own file in memory.

MFC usually creates its compound files in `OnNewDocument()` and opens them in `OnOpenDocument()`. The root storage pointer ends up in a member called `m_lpRootStg`. If we override those two functions and

place our own root storage (which is implemented in the internal memory) into **m_lpRootStg**, we can make MFC think that it's still writing into the compound files that it provides (although it never got a chance to create the file).

Since I provided my application's implementation in an SDI application, **OnNewDocument()** will always be called. I take advantage of this fact and open the first record's content objects (if there's data in the long binary field). Here's the code for **OnNewDocument()**:

```
BOOL CDbDepotDoc::OnNewDocument()
{
   DisplayRecordName();
   if (!m_dbDepotSet.IsEOF() && !m_dbDepotSet.IsEOF())
   {
      VERIFY(OpenStorageOnRecord());
      // Force a load of the Embeddings.
      return OnOpenDocument(m_strPathName);
   }

   return TRUE;
}
```

I call a helper function, **DisplayRecordName()**, which takes the first name and last name fields of the record and displays them in the caption bar of the window (along with the application name). I then call another helper function, **OpenStorageOnRecord()**, which creates the **ILockBytes** pointer and then creates a compound file in memory on top of the **ILockBytes** pointer. It does this only if the recordset didn't come back with data in the long binary field:

```
BOOL CDbDepotDoc::OpenStorageOnRecord()
{
   // Create the ILockBytes in memory.
   HRESULT hr = ::CreateILockBytesOnHGlobal(
      m_dbDepotSet.m_Embeddings.m_hData,
      FALSE,
      &m_lpLockBytes);
   if (FAILED(hr))
      return FALSE;

   // Did we already get a docfile? If not create one.
   if (m_dbDepotSet.m_Embeddings.m_hData == NULL)
   {
      hr = ::StgCreateDocfileOnILockBytes(
         m_lpLockBytes,
         STGM_SHARE_EXCLUSIVE|STGM_CREATE|STGM_READWRITE,
         0,     //Reserved; must be zero
         &m_lpRootStg);
      if (FAILED(hr))
      {
         m_lpLockBytes->Release();
         m_lpLockBytes = NULL;
         return FALSE;
      }
   }

   // Open storage if it wasn't created from scratch.
   if (m_lpRootStg == NULL)
   {
      HRESULT hr = ::StgOpenStorageOnILockBytes(m_lpLockBytes, NULL,
         STGM_SHARE_EXCLUSIVE|STGM_READWRITE, NULL, 0, &m_lpRootStg);
      if (FAILED(hr))
         return FALSE;
   }

   return TRUE;
}
```

When I call **CreateILockBytesOnHGlobal()**, I always pass it the long binary field's data handle. If this handle is not **NULL**, it will be used for recreating the **ILockBytes**. Otherwise, OLE will allocate a new shared memory block for us, on which it then bases a new **ILockBytes**. We'll see how we get to the **HGLOBAL** of this block a little later when we need it to save the data from memory back to the database.

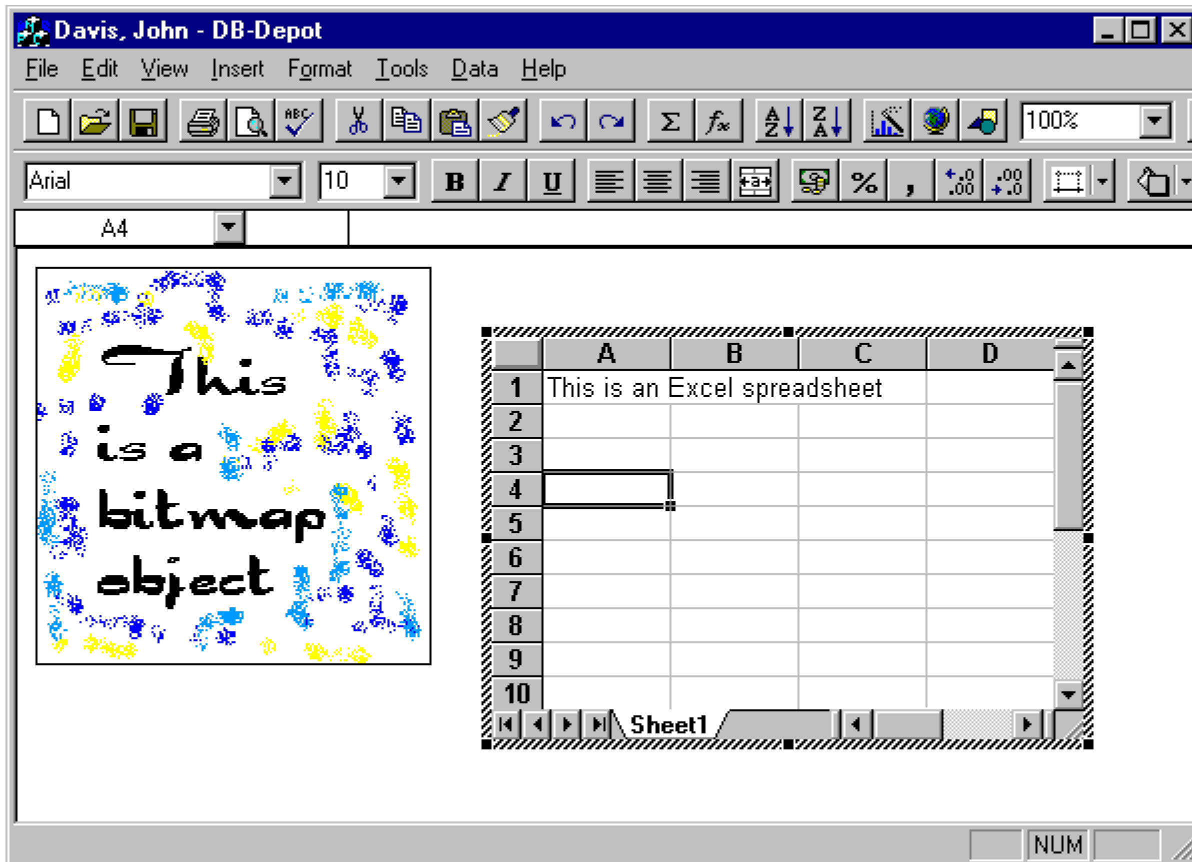
If the long binary field's handle isn't **NULL**, we know that it already points to a compound file in memory. If it *is* **NULL**, we have to create a new memory compound file, hence the call to **StgCreateDocfileOnILockBytes()**. If it isn't **NULL**, we know that we already have a memory-based compound document from the database, but we still need to open it, hence the call to **StgOpenStorageOnILockBytes()**. At the end of **OpenStorageOnRecord()**, we should have a valid pointer to **ILockBytes**, a memory-based compound document and a pointer to its root storage.

We call **OnOpenDocument()** from **OnNewDocument()** to force the framework to reload the objects. I call the base class's **OnOpenDocument()**, passing it **NULL** for its one and only parameter, which keeps the base class from calling **DeleteContents()** (which would destroy my memory compound file).

```
BOOL CDbDepotDoc::OnOpenDocument(LPCTSTR lpszPathName)
{
   // Let the existing code do its thing.
   // Don't call DeleteContents.
   BOOL bRet = COleDocument::OnOpenDocument(NULL);

   return bRet;
}
```

When the base class is called, MFC will do its normal processing of the objects stored in the compound file. It will assume that, since the **m_lpRootStg** member has a legitimate value, it is the compound file that MFC normally creates. Keep in mind that, since we pass **NULL** to **OnOpenDocument()**, **DeleteContents()** won't be called and MFC will continue to use the value in **m_lpRootStg**. Normally, MFC destroys the compound file in **m_lpRootStg**, so it can open up the file specified in the parameter, placing the new storage in **m_lpRootStg**.

I also provided a way to navigate through the other records via menu options or the tool bar.



When you choose to move to another record, one of the **OnRecord***xxx***()** functions will be called, where *xxx* is **First**, **Next**, **Prev**, or **Last**. These functions will attempt to save any changes that have been made to the current record by calling **COleDocument::SaveModified()** before moving to the next. **SaveModified()** calls a function **CDocument::DoFileSave()** which can be a little troublesome, since it looks for a file with the current filename on disk. If it doesn't find it, it calls **CDocument::DoSave()**. The default version of **DoSave()** will display the common Open file dialog to ask the user for a file name to save the data. Since this is not the behavior we want, we need to override **DoSave()** in our class. Here's my implementation:

```
BOOL CDbDepotDoc::DoSave(LPCTSTR lpszPathName, BOOL bReplace)
{
   // If we try to close without saving, dialog box is displayed.
   // We override DoSave and the dialog never appears.
   return OnSaveDocument(lpszPathName);
}
```

Pretty complicated function, huh? I simply force it to call **OnSaveDocument()**. We'll look at the actual saving code in a minute, but let's finish the record movement functions first. After **SaveModified()** has been called, I call **DeleteContent()** which forces the objects in the document's list to be destroyed. Next, I call a helper function which I provided, called **CloseStorageOnRecord()**. This handy function performs cleanup for the **m_lpRootStg** member and the **ILockBytes** member (which, by the way, is called **m_lpLockBytes**). It also frees the data held by the long binary field in the record of the recordset (since the **CLongBinary** field won't do this on its own within the context of the recordset).

Back in the record movement function, I move to the next record in the record set and then call **OnNewDocument()**. Here's one of the record movement functions (the rest are pretty much the same, except they move to a different record in a different direction):

```
void CDbDepotDoc::OnRecordNext()
{
   if (!SaveModified())
      return;
   DeleteContents();
   CloseStorageOnRecord();
   m_dbDepotSet.MoveNext();
   OnNewDocument();
   UpdateAllViews(NULL, HINT_RESET_ITEM);
}
```

If the user selects the File/Save option from the menu, or presses the Save button on the toolbar, this normally results in a call to a function called **CDocument::OnFileSave()**. This function also calls **CDocument::DoFileSave()**, which I don't want to happen, since I don't need the framework to ask the user whether they want to save the document, so I simply provided an override for the function and call **OnSaveDocument()** directly. So let's look at **OnSaveDocument()**.

OnSaveDocument() calls the base class to allow it to save the information for the embedded objects into the compound file that we provided, and then I take over. Note that **OnSaveDocument()** in the base class won't bother to look at the filename supplied, since first it looks at the **m_lpStorage** member and if it's valid (and compound files have been enabled) it will use that. I get the **HGLOBAL** from the **ILockBytes** and store it into the record of the recordset. The **HGLOBAL** will either be a brand new **HGLOBAL**, or it's the one that I provided from the record in the recordset. I then take the record and write it back to the database table, and I'm done. Here's the complete code for the **OnSaveDocument()** function I created:

```
BOOL CDbDepotDoc::OnSaveDocument(LPCTSTR lpszPathName)
{
   // Let the existing code do its thing.
   BOOL bRet = COleDocument::OnSaveDocument(lpszPathName);

   // Prepare for changes.
   m_dbDepotSet.Edit();

   // Get the HGLOBAL and store it.
   HRESULT hr = ::GetHGlobalFromILockBytes(m_lpLockBytes,
      &m_dbDepotSet.m_Embeddings.m_hData);
   m_dbDepotSet.m_Embeddings.m_dwDataLength =
      GlobalSize(m_dbDepotSet.m_Embeddings.m_hData);
   if (SUCCEEDED(hr))
   {
      // Commit changes to the database.
      m_dbDepotSet.SetFieldDirty(&m_dbDepotSet.m_Embeddings, TRUE);
      m_dbDepotSet.SetFieldNull(&m_dbDepotSet.m_Embeddings, FALSE);
      m_dbDepotSet.Update();
   }
   else
      bRet = FALSE;

   return bRet;
}
```

When you're working with the **CLongBinary** class, make sure that you always clear out the memory as you move from record to record, as the framework won't do it for you. I found this out the hard way, and I'm trying to save you a bunch of headaches. This is the code I use to clear my long binary field, which you can find in the **CloseStorageOnRecord()** function in my document class:

```
   if (m_dbDepotSet.m_Embeddings.m_hData != NULL &&
```

```
        m_dbDepotSet.m_Embeddings.m_dwDataLength > 0)
{
   ::GlobalFree(m_dbDepotSet.m_Embeddings.m_hData);
   m_dbDepotSet.m_Embeddings.m_hData = NULL;
   m_dbDepotSet.m_Embeddings.m_dwDataLength = 0;
}
```

So there you have it. A real life situation where you can actually use embedded content objects and the OLE Documents technology. As long as your database back-end can handle long binary fields, you can implement this in your own code and get it working in a couple of hours. I've tried embedding all kinds of objects, embedded or linked, and they all worked. Try it, you'll love it.

Monikers

Although I prefer to embed objects into a document, there are times when you might have an object which needs to be linked to the document because you might want to share it with several documents. If you do this, you would only need to update the object in one place and all references to the object would automatically be updated via OLE.

This is possible because the only thing that lives in the documents is an object called a **moniker** and the presentation data of the object. The native data of the object lives somewhere else, and the moniker points to it. The information that the moniker maintains can be as simple as a file name, such as **`C:\My Documents\Fourth Quarter 95.xls`**, or as complicated as including a range of cells within a spreadsheet file that defines the object such as **`C:\My Documents\Fourth Quarter 95.xls!Sheet1!A1:D5`**.

When the linked object is inserted into the container's document, a snap-shot of the object's rendering is saved in the container's compound file. This is used for drawing or printing the object when the container is told to print its document. If the original object changes in its native server application, the object's presentation can be updated in the container.

Understanding Monikers

OLE maintains two monikers for a container: an absolute and a relative moniker to the linked object. For example, if you have a container document with a path of **`C:\My Documents\Docs\ForSale.doc`** and this document has a picture linked to a file named **`C:\My Documents\Pictures\My House.pic`**, OLE will maintain a moniker for **`C:\My Documents\Pictures\My House.pic`** and one for **`..\Pictures\My House.pic`**.

This seeming redundancy helps linked objects find their data again. If you rename the directory structure to **`C:\My New Documents`**, for example, the absolute moniker will fail, but the relative one will still succeed. However, it's not foolproof. If you move **`My House.pic`** to a totally different directory (such as **`C:\My Documents\All Pictures`**), both monikers will fail. As you can see, even with two monikers, it can become difficult for the moniker to find its file for editing the actual data. In the future, Windows is supposed to maintain the connection and update the information as the files are moved around on the hard-drive, but for the meantime, we'll have to find our own solution. This solution comes in the form of a custom moniker.

When you create monikers, you commonly use the Microsoft-provided monikers by calling the OLE API functions which return a moniker of the requested type. You usually get an **`IMoniker`** interface pointer which contains methods to **bind** to, or activate, the object. The **`IMoniker`** interface is derived from the **`IPersistStream`** interface which contains methods for storing or loading data to and from a stream. This gives the moniker the ability to become persistent. In other words, given a stream, a moniker can save information to it, or load information from it. What kind of information you ask? Whatever it needs to recreate or bind to the object that it points to.

Monikers also provide support for returning display names (text that gives a more descriptive title for the linked data that can be shown to the user). Usually, the display names are nothing more then the file name contained within the moniker. Monikers can also be told to create themselves based on a text string (sometimes provided by the end user). The process of creating a moniker from a display name is called **parsing**. Where would you use this feature? Well, let's say that you can't bind a moniker, what then? Why

not ask the user if they know where the linked object's data is? You can simply display a dialog box with an edit control so the user can type the new location of the linked object's file. You then would need to recreate the moniker, of course.

One of the most important functions in the **`IMoniker`** interface is **`BindToObject()`**. The function is called when the container application needs to bind to the object (meaning *load* the object if it's not in memory and connect to it) that the moniker points to, and return an interface requested by the container application. The requested interface is normally **`IOleObject`**, which gives the container application the power to activate or open the object to allow the user to edit it.

Also important is the **`BindToStorage()`** function, which allows a container with intimate knowledge of the server to bind directly to an **`IStorage`** in the linked object's file, giving complete access to any of the streams within it to the container.

Collectively, this technology is called the **OLE Naming Technology**. It gives programmers the ability to name anything that can be categorized or labeled. For example, let's say that we have a database and within that database, we have some tables. Within one of those tables, we'd like to execute a query, returning a result set of the records that met the criteria. We can make our database/table/query into a moniker that can then be bound to and a presentation of the data updated. Using monikers for object location in a file system is probably the simplest form and yet many find the concepts difficult to understand. I'll see if I can shed some light on the subject.

When programmers don't understand something, they usually go into a denial stage. You know that you're there when you say things like, "I just don't understand why this even exists", or, "Why is this so difficult, couldn't they have made it easier?" I'll have to admit that even I have been guilty of this from time to time, but let me clear up one thing: monikers are very powerful. The reason they exist is to provide a level of abstraction to the container application.

As we mentioned in the last section, containers are not supposed to be aware of the objects they contain in their documents, but how do you achieve this level of abstraction if the object that is being displayed in the document's window doesn't really live in the document. You could simply make the container aware of that fact and have it maintain a path to the linked object's location and that would probably work for file type objects, but what happens when you want to link to other types of object, such as database/table/query objects or file/sheet/cells objects? Does that mean that you would reinvent the wheel each time? I would rather have a solution that can be flexible enough to withstand the powers of progress and future development. Monikers give us this flexibility.

By simply loading an object that always maintains the same interface, having the ability to call one function, and letting it do its thing, we achieve a high level of abstraction from the container application. No matter what kind of moniker the container is holding, it can always rely on the fact that all it has to do is call **`IMoniker::BindToObject()`**, and the job will be done. The end result is that the container will get back a pointer to an interface on the object, no matter where the object was created from.

The responsibilities of the container are pretty simple. It needs to obtain a pointer to one of the interfaces on the object when it has been loaded. At some point, it might need to display text to be read by the user, or a description of the moniker (for example, when the linked object is selected), or it might want to activate the object, giving the user the opportunity to edit the data contained in the linked object.

The container begins this process by initially calling **`IMoniker::BindToObject()`** and requesting an interface from the moniker. At this point, the moniker takes over and attempts to locate the file that it points to and create a COM object that can handle the data maintained by the file. How does it find the

COM class for the object? Easy, by looking in the file for a registered CLSID. This is usually done with a call to **`GetClassFile()`** which returns the CLSID associated with a file. This CLSID was most likely written into the root storage of the file by the server application with a simple call to the **`WriteClassStg()`** function.

Once the moniker knows the CLSID, it can call **`CoCreateInstance()`** and before you know it, we have ourselves a COM object that can load the file. The next thing we do is ask for the **`IPersistFile`** interface from the COM object and call the **`IPersistFile::Load()`** function to have the object load the file for the moniker. Once **`Load()`** returns, the moniker can ask the COM object for the interface that the container application requested. At this point, the moniker can simply drop out of the picture.

There are other situations that make the binding process much more difficult to achieve; for example, a moniker that points to an item within an object within a file, or the database query example that I spoke about before. We'll see how this is handled shortly.

Obtaining a Moniker

There are many ways that a container can obtain a moniker. It can create a moniker itself by calling one of the many OLE API functions, such as **`CreateFileMoniker()`**, or it might receive one from a clipboard or a drag-and-drop operation.

In any case, the role that the container plays is simple: it stores the moniker in its compound file or loads it from its compound file, and when it's ready to talk to the object that the moniker points to, it binds the moniker. Of course, there are other operations that the container might need to do, but I won't discuss them here, as they don't relate directly to monikers.

The Standard Monikers

Before we learn about custom monikers, let's meet the current set of standard monikers. There are five types of moniker available as part of the standard OLE implementation. The first is called a **file moniker**, and its purpose is to store a path to the link source. File monikers are persistent (they can store their internal data to a stream), they are bindable (you can call the **`BindToObject()`** and it will do something), and they are useful outside of a composite moniker. (You'll understand the last point when we discuss composites below.) You can create file monikers with a call to **`CreateFileMoniker()`**.

I've already discussed the steps that the **`BindToObject()`** function will take when you call it. The second moniker type is an **item moniker**, and its purpose is to provide links to pseudo objects within the link source's file, such as a group of cells in a spreadsheet file. This moniker is also persistent, causing it to store its internal data into a stream or load it from the stream, and it's bindable. However, unlike the file moniker, it serves no purpose if it isn't joined by another item or a file moniker to its left.

For example, to identify a page within a document, you might need to maintain the filename, the section of the document and, finally, the page itself. As you've just seen, we can use a file moniker to maintain the filename. One item moniker can then identify the section, while another can identify the page. This would be described with a notation like **`C:\My Documents\VCMC.doc!Section 1!Page 43`**, where the **`!`** character is the delimiter between the file moniker, the first item moniker and the second item moniker. You can create item monikers with the **`CreateItemMoniker()`** function.

The next moniker is called a **composite moniker**. Its purpose is to provide the glue between a file

moniker and one or more item monikers. Getting back to the example of a group of cells within a spreadsheet file, the composite moniker can be made up of an item moniker to represent the group of cells, a sheet moniker representing the sheet containing the group of cells and a file moniker to represent the spreadsheet file containing the sheet. The result would look like **`C:\My Documents\My Money.xls!Bank Sheet!A1:Z15`**. You can create composites with the **`CreateGenericComposite()`**.

The composite moniker's role is to tell the contained monikers to load themselves. This occurs when a container tells the composite to bind. In turn, it binds the contained monikers. This process starts with the right-most moniker, but since it cannot successfully bind if the monikers to its left haven't bound, each moniker gets the one to its left to bind first. The objects to the left of the right-most item may already exist so, in this way, only the objects that need to be created in order to get to the right-most item are bound. The monikers look in the **Running Object Table** (sometimes referred to as the ROT) to determine whether an object is already running.

The ROT is a system-wide table that allows us to make an entry which contains a moniker and an associated object (identified by its **`IUnknown`** *pointer). You're free to place any object in the table, as long as it supports* **`IUnknown`**. *This means that you can even place a C++ object there if the vtable starts with the three functions of* **`IUnknown`**. *There are functions and an interface for dealing with the ROT. The interface is called* **`IRunningObjectTable`** *and it can be obtained via the* **`GetRunningObjectTable()`** *function (for more information concerning the ROT, see the OLE reference manual).*

But how does Microsoft know that I will want to pull out sheet and cell data from the objects? What if my example involved a word-processing application with a file, pages, and text paragraphs? Or something else entirely?

The answer lies in the **`IOleItemContainer`** interface. You see, the same COM object that returns the **`IPersistFile`** interface from the server must also implement the **`IOleItemContainer`** interface. This interface will be passed up to the sheet item moniker. When the sheet item moniker calls **`IOleItemContainer::GetObject()`**, it will pass its persistent data (**`"Bank Sheet"`**). This **`IOleItemContainer`** will see this name and return an **`IOleItemContainer`** for the sheet, suitable for binding the cell information if necessary. The sheet moniker will then return this **`IOleItemContainer`** to the cell moniker. The cell moniker will then call **`IOleItemContainer::GetObject()`** passing its own persistent data (**`"A1:Z15"`**). This **`IOleItemContainer`** will be asked to return the interface requested by the container application, which it will happily do so, as long as it supports the interface.

The **`IOleItemContainer`** interfaces that we have mentioned live in the server application, and since it was the server who created the moniker returned to the container application, the server shouldn't have any problems dissecting the persistent data held by the monikers.

The fourth type of moniker is called an **anti moniker**. This breed is used to negate the last part of a composite moniker causing the last item moniker in the composite to be removed. These monikers are mostly used internally by OLE and I've yet to find a use for them myself. You can create them with a call to the **`CreateAntiMoniker()`** function.

The last type is called a **pointer moniker**. These are used to wrap up monikers in order to pass an **`IMoniker`** pointer to a function wishing to receive a moniker. This is the only way to pass a moniker around, since the interface is not marshaled. This is the only way that you can use pointer monikers and they can't be persistently saved. You can create them with the **`CreatePointerMoniker()`** function.

Although you can use any of these types of moniker in your applications, you're not restricted to them.

You can provide custom monikers which can be used to locate and load link sources from anywhere, including another machine or a back-end database. In the course of this section, we'll implement a custom moniker which allows the linked object to find its link source, no matter where you place the link source's file. But first, we need to introduce a couple more concepts.

There's an object called a **bind context** that provides storage for information needed to bind the moniker objects. This structure is passed around to all the monikers and is used by the them to pass more information to other monikers. You can think of the bind context as a channel for all of the monikers to tune into and find out what has already happened. The bind context is implemented by an interface (you knew I was going to say that, didn't you?) called `IBindCtx` and is returned when you call the `CreateBindCtx()` function. Most of the functions that you will call to either bind or find out information from a moniker will involve a bind context. Even a container will need to create a bind context before it calls `IMoniker::BindToObject()`.

The purpose of the bind context is to achieve better performance by placing the object needed to allow the monikers to bind in the bind context, using **parameters**. These objects will then continue to exist as long as the bind context is kept alive. Also, if a container application has simply caused an action that hasn't completely activated an object, but has instead loaded the bind context, the binding will occur immediately, since all of the objects needed might already exist in the bind context.

For example, let's say that the container has called `IMoniker::GetDisplayName()` from a composite moniker and it, in turn, has caused some of the monikers internally to bind in order to find out the information to return to the container. These objects might already exist next time the container calls `IMoniker::BindToObject()`, causing the function to make a speedy return.

A Custom Moniker

When we actually implement our custom moniker, we need to provide an implementation for the `IMoniker` interface. The most important function is `Reduce()`. Our function's implementation will create a standard file moniker and will return it for further use by whoever called the `Reduce()` function. The purpose of `Reduce()` is to cause a moniker to return another moniker that has been reduced to its bare minimum. For example, if we have a file moniker that contains the value of "`C:\Windows\..\My Documents\This File.doc`", reducing it would cause it to return a moniker with the value of "`C:\My Documents\This File.doc`".

I took severe advantage of the fact that this function is usually the main work horse for the other functions and placed my searching code here. If the moniker cannot locate the file that it thinks it should be able to locate, I go searching away through the hard drive and find the file. Then I create a moniker with the correct location of the file and return it from `Reduce()`. I also call the `Reduce()` function from the `BindToObject()` function.

It's the linked object (identified by `IOleLink`)stored in the container that will eventually need to tell the moniker object to link itself to its data source. The container must provide the object with the moniker by calling the object's `IOleLink::SetSourceMoniker()`. Later, it will call the object's `IOleLink::BindToSource()` to launch the data source's server with its data source loaded for editing.

Implementing a Custom Moniker

Custom monikers can be implemented for use in situations where the standard monikers' inner workings

aren't enough. As I've pointed out already, although the standard monikers are fine for some situations, they do have their drawbacks. For example, if you want the source of data to be a back-end database, or a source other than a file on your hard-drive, you'll need to implement a custom moniker.

There are several pieces of functionality that we must implement into our custom monikers. First of all, since the moniker is, after all, a COM object, we must implement a class factory for it. This is easy, since we can use the MFC macros **DECLARE_OLECREATE()** and **IMPLEMENT_OLECREATE()** which supply a class factory object (**COleObjectFactory**) for the COM class.

Since I'm providing the code in an in-process server, we need to expose a few functions, such as **DllGetClassObject()** and **DllCanUnloadNow()**. We've seen code for these functions before in some of the other sample applications that I've provided. We'll also have to provide a function similar to the **Create***xxx***Moniker()** functions that Microsoft provides. We'll call our function **CreateSearchableMoniker()**. This will exist in the same in-process server and a container or object can call it to create a searchable object. This object can find its data source, no matter where it is, as long as the data source is on the same drive as the container's compound file that contains the moniker object.

I wrapped up the functionality of the custom moniker in a class called **CSearchableMoniker**, which is derived from **CCmdTarget** and implements the interfaces we need. The class needs to implement **IMoniker**, **IPersistStream** (since **IMoniker** is derived from **IPersistStream**), **IPersist** (since **IPersistStream** is derived from **IPersist**), and **IUnknown** (which is implemented by **CCmdTarget** for us). The **CreateSearchableMoniker()** function will create an instance of this class and return a pointer to its **IMoniker** interface.

The bottom line is that the moniker needs to maintain a file name and find it if it's not in the location specified originally. Since the custom moniker can find a file name by searching through the user's directory structure, it's not necessary to provide the full path name of a file. However, having the full path makes finding the file a lot faster, because the moniker doesn't have to look for it (unless it has been moved).

Once the file has been located, the **Reduce()** function delegates to a standard file moniker and returns it. At this point, the custom moniker has done its work, so we can call upon the service of the standard monikers to complete the task.

Using the Searchable Moniker

Now it's up to the container to make use of this exciting new moniker. The appropriate time to use it is when a linked object is passed to the container or when the user chooses to link to a file using the Insert Object dialog.

When your container application is trying to make use of the searchable moniker, it will need to create one and pass it to the function which will create the link object:

```
hr = CreateSearchableMoniker(T2COLE(strPathName.GetBuffer(0)),
   &lpMoniker);

hr = OleCreateLink(lpMoniker, IID_IOleObject, render, NULL,
   lpClientSite, lpStg, (LPVOID FAR *)&lpOleObject);
```

When the container application is later reloaded with the link object and calls the moniker's **BindToObject()** or **BindToStorage()** function, the object locates the source and places the server in the

running state. The object can then be activated for editing. MFC handles most of these details for you, since the work is generic enough that it can decide what to do without any assistance from you.

The moniker's **`IPersistStream`** functions will be called for saving or loading the filename and path name. This is the extent of the functions. Therefore, the real work of finding the source is the responsibility of the **`IMoniker`** functions. When the **`Reduce()`** function is called, it uses the services of a global helper function I created, called **`FindFile()`**. This function searches recursively for a match until all directories have been searched, or the subdirectories have been exhausted.

You can find complete code for the searchable moniker in the **`SrchMonk`** directory on the CD.

IChapter->Release()

It's been a long ride in this chapter, but hopefully, you've seen some sights that have opened your eyes to new possibilities and tasks to which you can apply OLE. My goal was to provide you with some real-world examples of using compound documents and servers. Along the way, we just had to make a couple of pit stops and learn about several technologies to help us with the big picture.

In the next couple of chapters, I'll show you some ways that you can use OLE Automation, OLE Controls and ODBC to provide for a more open, component-based environment.

OLE Automation and Controls

In the last couple of chapters, we've discussed several OLE technologies and discovered some new uses for them. Continuing with the theme of OLE, this chapter examines a few more technologies and some handy ways of using them in different situations.

In this chapter, we'll discuss OLE Automation and OLE controls (or OCXs). These technologies, coupled with the techniques we've learned in previous chapters will play a major role in the next chapter, where we'll use them to implement metaclasses and business objects for a client-server environment.

While we're on the subject of OLE controls, you may have heard about **ActiveX controls**, and wondered how they fit into the picture. These are the next generation of OLE controls, designed to be smaller and faster, so that they can be used in the Internet environment (as well as anywhere else controls may be used). With the new specification, Microsoft have radically reduced the number of OLE interfaces that a control is required to implement, making it a vastly slimmed-down version of its former self. This means that all OLE controls are ActiveX controls, but not vice versa, since ActiveX controls are not required to support all the interfaces currently required by OCXs.

In this chapter, we'll also get a chance to play with multithreading and thread synchronization, and we'll have the opportunity to discuss ways of sharing global data between applications.

You'll find that this chapter is divided up into four main sections. The first is a high-level introduction to the how and why of OLE Automation. This is followed by sections on programming automation, firstly at the COM level, then with the help of MFC. The last section concentrates on OLE controls.

OLE Automation

Let's take a brief look at some of the concepts and terminology that are important to automation.

A good many programs over the years have been designed to be extensible and customizable, often through the use of macro languages. Microsoft Word has WordBasic, most of the other Office applications use VBA, and most editors have little languages so that users can write their own functions to extend and enhance the operation of the package.

Programs let users drive them automatically by writing macro functions, but how about having programs cooperate between themselves, without human intervention? For example, this would let a scientific data-gathering package send commands to a spreadsheet so that it could use the spreadsheet's graphing capability to display its data, instead of having to provide the functionality itself.

The problem is that every package has tended to implement its own macro language, or other means of being externally driven, so it was hard to produce any sort of generic functionality.

What was needed was some sort of standard mechanism through which programs could communicate, which all could implement and use. This was provided in earlier releases of Windows by the DDE (Dynamic Data Exchange) mechanism.

Using DDE, a server and client participate in a DDE 'conversation' by means of Windows messages, which allow them to exchange data and cause each other to execute command strings. DDE is very simple in concept, being composed simply of a number of Windows messages, plus a protocol for using

them. In general, this protocol involves 'call and response' sequences, with one side of the conversation sending a message (such as the client sending **`WM_DDE_REQUEST`** to request some data from the server, or **`WM_DDE_EXECUTE`** to send a command string) and the other replying with a response.

Many DDE-enabled applications have been written and DDE is quick because it operates at the level of Windows messages, but it's limited in what it can do and is also quite hard to program. This is because the onus for coding the protocol at the client end, setting up the right sequences of messages and providing code for error handling and recovery, is placed firmly on the shoulders of the programmer.

The successor to DDE is **OLE Automation**, which is a great improvement, especially when you're using MFC, because the communication mechanism is handled for you by OLE and COM.

First, a Recap...

Let's start off with a very brief recap of what we know about COM interfaces, just to put what we're going to discuss into perspective.

COM objects use interfaces to expose functionality to the outside world; nothing can be known about a COM object except what it makes known through its interfaces. An interface comprises a number of related functions, grouped together and given a name, and is analogous to a C++ class. In fact, the mapping of interfaces to C++ classes is very close and makes C++ a natural language in which to write COM code. All interfaces are uniquely identified by an interface identifier (IID) which is stored in the registry.

The most fundamental COM interface is **`IUnknown`**, which exposes three functions: **`AddRef()`**, **`Release()`** and **`QueryInterface()`**. The first two are used to implement a reference counting scheme which controls the life of the COM object; the third is fundamental to the operation of COM. Given an IID, **`QueryInterface()`** will return a pointer to an interface of the given type, if it supports it (and if it wants to give it to you!).

All COM interfaces have to provide the three **`IUnknown`** methods in addition to any others that they may implement. This means that if you know one interface on an object, you can use its **`QueryInterface()`** to get a pointer to any other interface that the object supports. It also means that, in a sense, all interfaces 'inherit' from **`IUnknown`**, since they all contain **`IUnknown`**'s methods.

OLE uses a whole bunch of predefined COM interfaces to implement its services, but we can also create our own custom interfaces, if we need to. This would allow us to provide programmability for any application we choose to write.

Problems with Custom Interfaces

However, although it would be perfectly possible to provide automation using ordinary COM interfaces, providing a custom interface for each group of methods we wanted to expose leads to a couple of problems.

Firstly, languages that don't support pointers, such as Visual Basic, have trouble using interfaces in the normal C++ way. Making these languages use raw interfaces isn't going to persuade people to adopt automation!

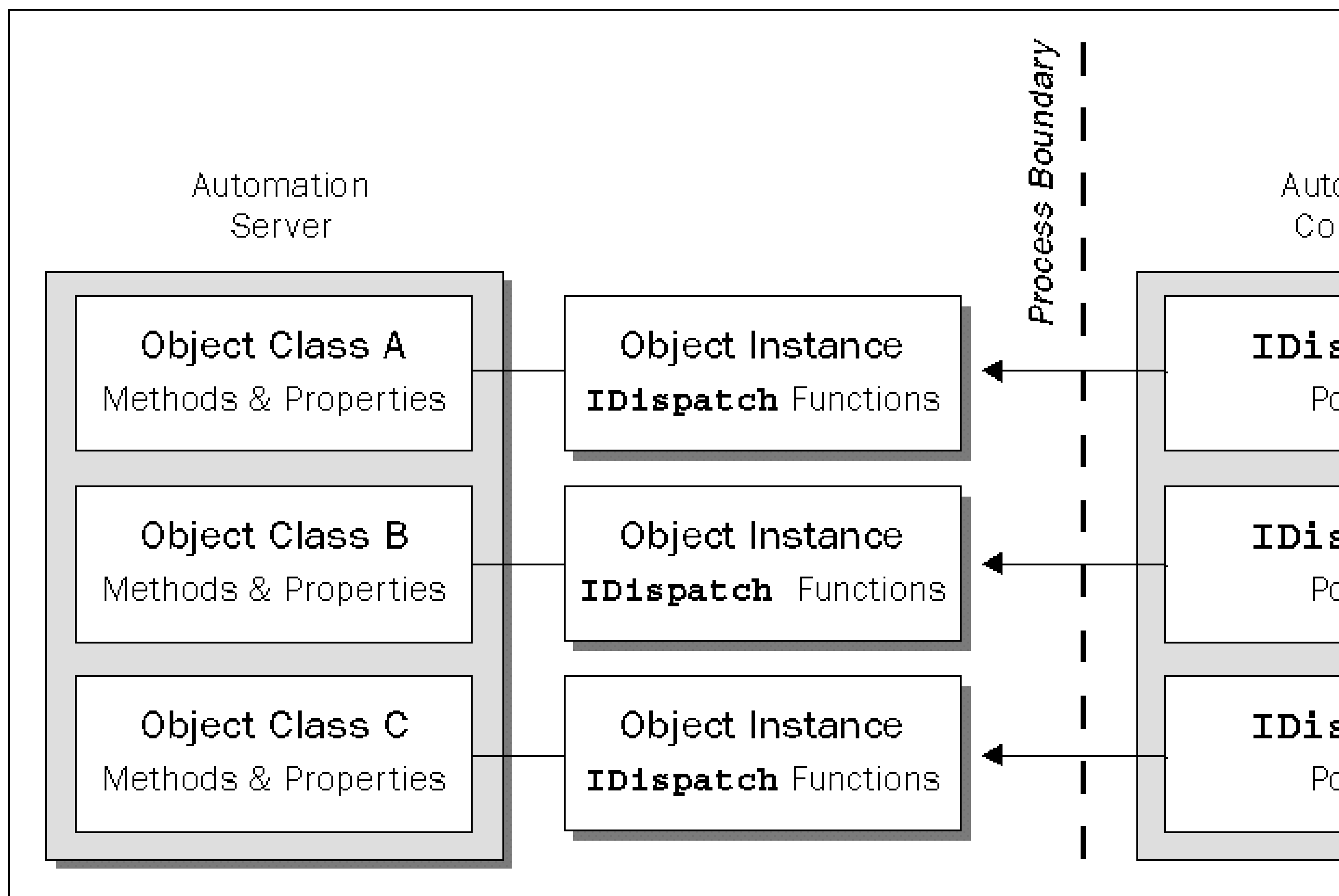
Another obvious problem is the proliferation of custom interfaces which would result, and the necessity for a client to know which interfaces to call on the servers it wished to use. If everyone handled automation through their own custom interfaces, there would be little opportunity for applications to work together in the way we'd like.

Invoke() to the Rescue!

What we need is some sort of standard interface for handling automation which all clients can use in the same way and which supports the ability for a client to find out at run time whether a server supports a particular automation method or property, and issue a dynamic call.

This interface is called **`IDispatch`**, and dynamic invocation of automation methods is done using the **`IDispatch::Invoke()`** method, which we'll meet in much greater detail as we progress through this chapter.

The **`IDispatch`** interface allows an object to communicate with others using **methods**, **properties** and **events**. Methods are akin to functions, in that they are requests to an object to do something; properties are akin to variables, holding values which describe the state of the object; events are things of which the object can notify a client. It's worth noting here that although properties look like variables, they're implemented using functions, because COM interfaces only expose functions.



Handling automation through one interface allows the implementers of languages like Visual Basic to hide the code for using this interface and provide a simple method in the host language for accessing it.

Automation capability has been added to many languages and development tools, such as Visual Basic, PowerBuilder and Delphi, but we're finding that the trend is leaning heavily towards Visual Basic as the language of choice for controlling and extending other applications that expose automation classes. This means that we, as C++ programmers, will write automation-enabled objects and that many users will want to drive them from Visual Basic, or one of its derivatives, such as VBA (Visual Basic for Applications) or VBScript. This is even more likely now that Microsoft are encouraging developers to provide scripting support for the VBScript DLL, so that developers can add Visual Basic as the macro language for their applications, and use that to perform automation.

A Simple Example

For example, let's say I have an application which contains functionality for establishing database connections, creating result sets based on queries to tables in a database and displaying data from result sets to a window. In the same way that I would divide this functionality into classes in C++, I could also provide automation objects which group the functions into logical categories.

Using Visual Basic, you could, for example, use these automation classes by creating a database object, a result set object and a display object. You could then establish a connection with a database using the database object and associate the database object with the result set object via one of the result set object's methods (such as **SetDatabase()** or something similar). Next, you could tell the result set object to look in a particular table and return some records that meet a specified criteria. Finally, you could associate the result set object with the display object by calling one of the display object's methods (such as **SetResultSet()**), causing the display object to display the data in the result set. Here's some Visual Basic code to simulate the situation:

```
Dim dbObj As Object
Dim rsObj As Object
Dim dpObj As Object

dbObj = CreateObject("BOM.Database.1")
rsObj = CreateObject("BOM.Recordset.1")
dpObj = CreateObject("BOM.Display.1")

dbObj.Connect("Orders")
rsObj.SetDatabase(dbObj.Connection)

rsObj.Query("Orders WHERE ID > 999")
dpObj.SetResultSet(rsObj.Data)
dpObj.Display
```

The code declares a few variables as being of type **Object**. It goes on to create and attach the variables to real automation objects. Visual Basic finds the program ID in the registry, converts the program ID to a CLSID and finds the associated server. It then goes on to load the server and create an object based on the CLSID. Finally, it retrieves the object's **IDispatch** pointer and attaches it to the Visual Basic variables.

When you call one of the methods or properties of the objects, Visual Basic calls the objects' **IDispatch::Invoke()** function, although this is invisible to the Visual Basic programmer.

The VARIANT Data Type

We've seen how **`Invoke()`** provides a way to make automation work, but we have a problem. **`Invoke()`** can be used to execute indirectly any function supported by the automation server, and these functions need to be able to take a wide range of number and types of arguments. How will we be able to use a single function call, **`Invoke()`**, to pass variable numbers and types of argument in a simple, portable and system- and language-independent way?

The answer is by using **`VARIANT`**s. A **`VARIANT`** is a structure which is able to store different types of data, using a type flag and a union. The flag denotes what sort of data is being stored, and the actual data is stored in the union as the appropriate type. The union in the **`VARIANT`** structure has entries to allow the storage of most of the data types you'll need, and we'll examine them more closely a little later in the chapter.

`Invoke()` uses **`VARIANT`**s for passing arguments and getting a return value. Arguments are passed to automation methods using an array of **`VARIANT`**s which is, in turn, packed into a **`DISPPARAMS`** structure. We'll see a little more of this structure later on. Likewise, the return value from the method is passed back as a **`VARIANT FAR*`**.

If you program automation at the COM level, you're responsible for converting data to and from **`VARIANT`**s, but if you use the MFC with its higher-level support, the conversion is automatic, and you'll seldom need to see a raw **`VARIANT`** structure.

Type Libraries

How can a controller find out what automation methods a server exposes? By using **type information**, usually stored in a **type library**. A type library is a repository of information that describes an object's methods, properties and events. Type libraries are indispensable to the operation of automation. Servers use the type library to obtain the interface descriptions they supply to clients, and tools can use type information to help in the automatic construction of automation client code.

Type library information may be held in a separate file (usually with a **`.tlb`** extension), or as a resource in a server's **`.exe`** or **`.dll`** or **`.ocx`**. You can browse through type information using the OLE object browser, called **`Ole2vw32.exe`**, which Microsoft provide in the **`\Msdev\Bin`** directory. You can start this application by choosing OLE Object View from Developer Studio's Tools menu. This program looks at the registry and lists all of the objects that have anything to do with OLE (including type libraries).

The browser shows different icons next to the entries to signify what type they are. The most interesting from our point of view are:

which denotes an OLE control

which shows a type library

When you double-click on the line containing a type library, you'll see a dialog box showing the methods and properties described in the library.

How do we create type library information? The hard way is to do it yourself using the **`CreateTypeLib()`** API call and the **`ICreateTypeLib`** and **`ICreateTypeInfo`** interfaces. The easier way is to write an ODL file, then compile it using the **`Mktyplib`** program. (The easiest way of all is to create your server as a Visual C++ project, where all this is done for you!)

ODL (Object Description Language) is an ASCII scripting language, similar in concept to the resource definition language used in **.rc** files, which is used to write descriptions of all the type information for an object. Here's a sample, showing part of an ODL file from a Visual C++ project:

```
// autoEx.odl : type library source for autoEx.exe

// This file will be processed by the Make Type Library (mktyplib) tool to
// produce the type library (autoEx.tlb).

[ uuid(D4911E21-8656-11CF-95F3-D169224F4B53), version(1.0) ]
library AutoEx
{
   importlib("stdole32.tlb");

   //  Primary dispatch interface for CAutoExDoc

   [ uuid(D4911E22-8656-11CF-95F3-D169224F4B53) ]
   dispinterface IAutoEx
   {
      properties:
         // NOTE - ClassWizard will maintain property information here.
         //    Use extreme caution when editing this section.
         //{{AFX_ODL_PROP(CAutoExDoc)
         [id(1)] long val1;
         [id(2)] long val2;
         [id(3)] short operation;
         [id(4)] long result;
         //}}AFX_ODL_PROP

      methods:
         // NOTE - ClassWizard will maintain method information here.
         //    Use extreme caution when editing this section.
         //{{AFX_ODL_METHOD(CAutoExDoc)
         [id(5)] void Calculate();
         //}}AFX_ODL_METHOD

   };

   //  Class information for CAutoExDoc

   [ uuid(D4911E20-8656-11CF-95F3-D169224F4B53) ]
   coclass CAutoExDoc
   {
      [default] dispinterface IAutoEx;
   };

   //{{AFX_APPEND_ODL}}
};
```

The dispatch interface, **IAutoEx**, exposes four properties, **val1**, **val2**, **operation** and **result**, with dispatch IDs 1 to 4, and a single method, **Calculate()**, with the dispatch ID of 5.

You can write ODL files yourself, but if you use Visual C++, an ODL file will automatically be created and maintained for the interfaces in your project and compiled into a type library when you build the project.

We can use Visual C++ to read the type library for a server and create a class based on what it finds. This allows us to create an object which represents our connection to the server, with the member functions of the object corresponding to the methods and functions exposed by the server. This process of fixing the calls to the server methods at compile time is called **early binding**.

The opposite occurs when we use the **IDispatch** interface methods (such as **GetIDsOfNames()** and **Invoke()**) to find out what methods are supported at run time. Leaving the nature of the call to be

decided at run time is called **late binding** and gives you more flexibility than early binding, although it may be considerably less efficient.

Dual Interfaces

A **dual interface** is an extension to `IDispatch`, the standard interface for OLE Automation, and it lets an automation controller bind to a vtable instead of having to use `IDispatch::Invoke()`.

The difference is mainly in speed. Using `Invoke()` requires the controller to package each argument as an element in the `VARIANTARG` array, while with a dual interface, the controller can call the method directly. One of the main disadvantages from our point of view is that, with version 4.1 at least, there is no direct support for creating dual interfaces with MFC. However, VC++ 4.1 does come with a sample project, called `ACDUAL`, that demonstrates the changes that you need to make to an existing OLE Automation server to convert it to use dual interfaces.

If we create a server with a dual interface, details of the interface will be entered into the server's type library, so we can use Visual C++ to read the type library and create a class to represent the dual interface, just as we would with any other interface.

You create an interface class which is derived from `IDispatch`, so that it has the same methods as `IDispatch`, together with one or more custom methods which are compatible with OLE Automation. A compatible method is one whose argument types will fit into a `VARIANT`. This restriction on the argument types is the main disadvantage of using a dual interface.

As its name implies, a dual interface serves a dual purpose. It can be used as an ordinary dispatch interface by controllers which need to use this access method, while more sophisticated controllers can call the vtable directly. C++ and Visual Basic 4.0 are smart enough to use the dual interface if one is present, so implementing automation via dual interfaces can give considerably better performance in these cases.

The dual interface handles the marshaling of vtable methods automatically, so you don't have to write custom marshaling code.

Implementing Automation

We're now going to take a look at how you can implement automation, but before getting to grips with MFC, we'll first look at automation at the bottom-most level: COM and the basic OLE services.

When you work with COM, you usually implement an interface by deriving a class from an existing one (which as we have already seen, is nothing more than an abstract class with all of its functions defined as `pure virtual`). Once we've declared our class, we're required to implement each and every function in the interface (or get MFC to do it for us).

When an application wishes to communicate with our object via its interfaces' implementation, the application must provide a pointer (at compile time) and must define the type of that pointer. The compiler will examine the pointer and take the opportunity to set up the virtual function table (vtable) used by the pointer. The interface, which is included in both applications (client and server) tells the whole story of what the vtable should look like. Examine the following code:

```
#include <ole2.h>
```

```
...
LPUNKNOWN lpUnknown;

CoCreateInstance(..., (LPVOID*)&lpUnknown);
lpUnknown->QueryInterface(...);
lpUnknown->Release();
```

Notice that the pointer in the sample code is of type `LPUNKNOWN`, which basically means that it's a pointer to an `IUnknown` interface (`IUnknown*)`. When the compiler sees this, it knows that it must use the vtable created for the `IUnknown` interface, and will use this to find the actual addresses of the functions to be executed, in much the same way as a polymorphic pointer in C++ is used to index into a virtual function table.

Since both the client and server use the same interface definition when they compile and link, they are guaranteed to be able to communicate at run time via the vtable.

Some languages (such as Visual Basic) don't have the ability to deal with vtables and pointers, so something had to be done to let these languages communicate using COM.

Automation provides the solution to the problem because, using the `IDispatch` interface, an application can specify at run time which functions it wishes to invoke, using a single function. Granted, languages like Visual Basic must still internally support some type of early binding using a vtable, but it's only one interface to deal with, and the interface will never change. This fact allows the compilers to provide a simple way of dealing with the `IDispatch` automation interface.

Your application choosing to either invoke a method or get or set a property in an automation object is called **late binding**. Because the functions were not called at compile time, (in other words the compiler never saw the function being called), the compiler had no way of binding it then. All that the compiler saw was your application making a call to `IDispatch::Invoke()`. That's the only time the compiler gets involved. It's then up to the server application to break down and dissect the parameters passed to the `Invoke()` function, which include the identifier of the actual function and any parameters that should be passed to the function in the server.

Let me give you an example. If a client application wants to call a method exposed by an automation object, it will pass an identifier that represents the function to the object's `IDispatch::Invoke()` function (this is known as a **dispatch identifier** or a **DISPID**). The server will then receive the `IDispatch::Invoke()` call and execute the function identified by the ID passed to the `Invoke()` function.

The `IDispatch` interface has four functions: `GetTypeInfoCount()`, `GetTypeInfo()`, `GetIDsOfNames()`, and `Invoke()`. Of these, the two most important are `GetIDsOfNames()` and `Invoke()`.

Invoking Automation Methods and Properties

The `Invoke()` function has the following prototype:

```
HRESULT IDispatch::Invoke(DISPID dispID,
                          REFIID riid,
                          LCID lcid,
                          WORD wFlags,
                          DISPPARAMS* pDispParams,
                          VARIANT* pVarResult,
                          EXCEPINFO* pExcepInfo,
                          UINT* puArgErr);
```

The **DISPID** identifies the function or property you wish to invoke. This value is determined by the server. A client receives this identifier when it calls **GetIDsOfNames()** (more on this function later). When a client wishes to invoke a method or property, all that the client usually knows is the name of the method (or property). It next calls **GetIDsOfNames()**, passing it the name of the method to invoke, and gets the identifier of the method. It can then finally call **Invoke()** with the appropriate identifier for the method that it wishes to invoke.

Since this can be a time-consuming process, clients should gather the identifiers early in the life of the application and cache the identifiers away for later use, or use the type library (if one is available) to get this information at compile time.

The second parameter, **riid**, is a reference to an interface. Presently, this parameter is not used and should be set to **IID_NULL**. The third parameter, **lcid**, is the locale identifier and is used in Win32 National Language Support. Applications that don't need to make use of anything that may depend on the locale can simply ignore the **lcid** parameter.

Since **IDispatch::Invoke()** can be used to invoke methods as well as set and/or get properties, the fourth parameter, **wFlags**, is a flag that is used by the server to determine the task that the client wants the server to perform. This parameter can be set to one of four values: **DISPATCH_METHOD** which means that the client wants to call a function; **DISPATCH_PROPERTYGET** or **DISPATCH_PROPERTYPUT** determines that the client wants to retrieve or set the value of a property respectively; **DISPATCH_PROPERTYPUTREF** determines that the property is being changed by a reference assignment, rather than a value assignment.

The next parameter, **pDispParams**, is a pointer to a structure of type **DISPPARAMS**:

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

The first member holds the actual parameters that are to be passed to the automation method. You need to create an array of **VARIANT**s (one element per parameter) and assign the number of parameters to the **cArgs** member. You would create the array of variants as follows:

```
VARIANT vars[3];
DISPPARAMS dp;          // parameter array
dp.rgvarg = vars;
dp.cArgs = 3;
```

What's this **VARIANT** data type that we're using here? Since **Invoke()** has no way of determining what kind of parameters you'll be sending to the automation methods, or what kind of properties you'll be getting or setting, it needs to pass data around in a generic manner. It does this with the assistance of a structure called a **VARIANT**, a structure into which many different types of data can be packed, and which allows OLE to pass around values without worrying about their type. Each parameter sent to the **Invoke()** function is sent as a **VARIANT**, but before I give you an example of how it can be done, let's first take a look at the **VARIANT** struct:

```
struct tagVARIANT{
    VARTYPE vt;
    WORD wReserved1;
    WORD wReserved2;
```

```
        WORD wReserved3;
        union
        {
           long            lVal;            /* VT_I4                */
           unsigned char   bVal;            /* VT_UI1               */
           short           iVal;            /* VT_I2                */
           float           fltVal;          /* VT_R4                */
           double          dblVal;          /* VT_R8                */
           VARIANT_BOOL    bool;            /* VT_BOOL              */
           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|*           */
           unsigned char*  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*   pbool;           /* VT_BYREF|VT_BOOL     */
           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  */
           void*           byref;           /* Generic ByRef        */
        };
    };
```

The **`vt`** member determines the type of data being sent to the **`Invoke()`** function. The comment next to each element in the **`union`** indicates what the value of **`vt`** should be when you use that element. For example, if the data type is a Boolean value, the **`vt`** member should contain **`VT_BOOL`**.

As you can see, there's a type in the **`union`** for just about any kind of value you would want to send to an automation method. Packing data in this way means that the server must unpack the data before calling the automation functions, which expect to see the data in their natural form.

When the automation function returns, it needs to return its value as a **`VARIANT`** to the client side. You'll notice that this is done via one of **`Invoke()`**'s arguments, rather than using the function return value itself. This is because the function return is used to determine whether the call to **`Invoke()`** worked, rather than the result of the automation function.

The result returned by the automation function is passed back in the sixth parameter of the **`Invoke()`** function, the **`pVarResult`** field. The client can then check the **`vt`** field of the **`VARIANT`** and pull out the appropriate value from one of the **`union`** members.

The last two parameters of **`Invoke()`** are used to handle errors and exceptions that can occur in the server object.

The most important thing to remember when writing your server, is that you must devise a way to map incoming identifiers to properties or methods. You must use the **`wFlag`** parameter to determine whether the call is being made for invoking a method or requesting to set or get a property.

When the server application receives parameters, it must convert the **`VARIANT`**s to the native data type before passing the value to the actual automation method being called. It can do this using one of the

macros supplied by Microsoft. For example, if I want to extract an integer from a **VARIANT**, I could code it like this:

```
int nFirstParam = V_I4(var);
```

If I want to return a value back to the caller, I could set the **pVarResult** parameter as follows:

```
V_VT(pVarResult) = VT_I4;
V_I4(pVarResult) = nSomeValue;
```

These macros are just shorthand, used to access the fields of the **VARIANT**, so I could have written:

```
pVarResult.vt = VT_I4;
pVarResult.lVal = nSomeValue;
```

There are a number of functions to deal with **VARIANT**s, including **VariantInit()**, **VariantClear()** and **VariantChangeType()**. **VariantInit()** initializes a **VARIANT** structure. **VariantClear()** empties a **VARIANT** variable. You should always empty out a **VARIANT** before destroying it, since the **VARIANT** might be carrying a string or a safe array, in which case the data might need to be deallocated by the system. **VariantChangeType()** is used to change one type to another; for example, an integer to a floating point value.

Note that there is also an MFC class that wraps a **VARIANT**, called **COleVariant**, which provides numerous constructors, as well as overloaded comparison and assignment operators.

To send the parameters to the actual automation method, you'll have to traverse through the parameters in the **DISPPARAMS** array. This can easily be performed by calling the **DispGetParam()** API function:

```
DispGetParam(DISPPARAMS FAR* pdispparams,
             UINT position,
             VARTYPE vtTarg,
             VARIANT FAR* pvarResult,
             UINT FAR* puArgErr);
```

This function requires that you pass it the **DISPPARAMS** array along with the index of the element you wish to retrieve. The function returns the value at the location in a **VARIANT** that you provide.

One final word: in order to implement the **IDispatch::GetIDsOfNames()** function, you simply examine the list of dispatch methods and/or property names requested and fill the provided array with the DISPIDs of the methods and properties.

Since this is an MFC book, I won't give you any sample code for doing this kind of stuff by hand. Furthermore, I see no point in reinventing the wheel when MFC already does such a great job of implementing OLE Automation. In the next section, we'll find out exactly how MFC does this and how we can make use of it. All of this explanation will help us to understand the big picture later in this chapter, as well as in the next chapter, where we'll actually make use of OLE Automation to create a business object model with metadata and dynamic data object creation from information stored in a database.

OLE Automation Using MFC

In the last section, I explained the internals of OLE Automation and discussed the steps you need to follow to implement OLE Automation in your server application. Basically, the server needs to expose objects that support **IUnknown** and **IDispatch**. The **IDispatch** interface will be called to invoke automation methods and properties exposed by the automation object. We also learned that a single server can expose several objects, each containing their own methods and properties.

Believe it or not, the steps necessary to implement automation using COM-level techniques are just too hard! MFC provides a better solution with the help of ClassWizard and a new kind of map, called a **dispatch map**. The COM support is built into the **CCmdTarget** class and, therefore, allows any class that is derived from **CCmdTarget** to act as an OLE Automation class.

Implementing a Server

In earlier chapters, we learned that the **CCmdTarget** supports **IUnknown**. **CCmdTarget** also contains support for **IDispatch**. If you remember from our earlier chapters, we used several MFC macros to provide nested classes within our **CCmdTarget** derived classes. We simply derived a class from **CCmdTarget** and used the **BEGIN_INTERFACE_PART()** and **END_INTERFACE_PART()** macros to define our interface. Therefore, I'd expect to look in MFC's header files for the **CCmdTarget** class and see a **BEGIN_INTERFACE_PART()** and **END_INTERFACE_PART()** for **IDispatch**. Surprisingly enough, there's no mention of these macros in the class. What gives?

The answer is hidden in a class called **COleDispatchImpl**. This class implements the **IUnknown** members as well as the **IDispatch** members, but MFC must still provide some glue between **CCmdTarget** and **COleDispatchImpl**, right? How does it do that? With a **CCmdTarget** function, named **EnableAutomation()**.

MFC's Automation Code

To provide automation for a server's automation class, you'll need to call **EnableAutomation()** from the class' constructor. Here's is what the **EnableAutomation()** function looks like:

```
void CCmdTarget::EnableAutomation()
{
   // construct an COleDispatchImpl instance just to
   // get to the vtable
   COleDispatchImpl dispatch;

   // copy the vtable (and other data) to make sure it
   // is initialized
   m_xDispatch.m_vtbl = *(DWORD*)&dispatch;
   *(COleDispatchImpl*)&m_xDispatch = dispatch;
}
```

You can find this in **\Msdev\Mfc\Src\Oledisp1.cpp**.

Pretty weird, wouldn't you say? Why would they declare a variable on the stack, grab its address, cast the address to a **DWORD** pointer and then dereference it as a **DWORD**? Maybe they want to get at **COleDispatchImp**'s virtual function table. The **m_xDispatch** member is of type **XDispatch**, which is a nested class within **CCmdTarget**. The structure looks like this:

```
struct XDispatch
{
   DWORD m_vtbl;
   size_t m_nOffset;
} m_xDispatch;
```

Initially, **CCmdTarget**'s constructor sets **m_xDispatch.m_vtbl** equal to zero. Therefore, by default, there's no automation support because **CCmdTarget** doesn't have an **IDispatch** to return from **IUnknown::QueryInterface()**. As soon as you call **EnableAutomation()**, **CCmdTarget** gains access to an **IDispatch** vtable and can, therefore, return a pointer to its implementation.

How is automation handled inside **CCmdTarget**? If you look inside **Cmdtarg.cpp**, you'll find the following code:

```
const AFX_INTERFACEMAP_ENTRY CCmdTarget::_interfaceEntries[] =
{
   INTERFACE_PART(CCmdTarget, _afx_IID_IDispatch, Dispatch)
   { NULL, (size_t)-1 }     // end of entries
};
```

What does all this mean? A class contains an **_interfaceEntries** array which defines the COM interfaces supported by the class in the form of an **AFX_INTERFACEMAP_ENTRY** structure. This is a structure which holds a pointer to an interface ID and its offset from the **IUnknown** entry; in the case of **CCmdTarget**, it consists of one entry for **IDispatch**. The **INTERFACE_PART()** macro will convert the word **Dispatch** to the class name **m_xDispatch,** which will either be **NULL** or contain a pointer to the vtable of the **COleDispatchImpl** class, depending on whether or not the class supports automation.

In addition to **EnableAutomation()**, **CCmdTarget** has a few other useful functions. **GetIDispatch()** allows you to retrieve the **IDispatch** pointer from a **CCmdTarget** (or a derived class) object. **FromIDispatch()** goes the other way around. Given an **IDispatch**, **FromIDispatch()** can return the associated **CCmdTarget** object, or **NULL** if the **IDispatch** isn't associated with a **CCmdTarget** object.

Maps, Maps, and More Maps

If you don't know anything about the MFC team, you should learn this: they love creating maps from macros. There are more macros in MFC than in any class library I've ever seen, but at least the team has been consistent. They use maps for messages to determine which functions should be called and to perform interface lookups. If they use maps in MFC for just about everything, why should automation be any different? As a matter of fact, it's not.

MFC uses **dispatch maps** to perform the lookup for **IDispatch::GetIDsOfNames()** and **IDispatch::Invoke()**. Just like message maps, dispatch maps are implemented in the form of a table with rows and columns. Each row represents a method or property of the automation class, and each column contains information concerning the method or property. For example, if a client calls **GetIDsOfNames()**, MFC can simply look up the name in the table and return the associated identifier. If a client calls **Invoke()**, MFC can look up the identifier in the table and call the associated method or property.

When you create a message map in your application, you usually have to use **DECLARE_MESSAGE_MAP** in the header file and the associated macros, **BEGIN_MESSAGE_MAP** and **END_MESSAGE_MAP**, in the implementation file, respectively. Dispatch maps are no different. You use **DECLARE_DISPATCH_MAP** in the header file and **BEGIN_DISPATCH_MAP** and **END_DISPATCH_MAP** in the implementation file.

Just as you fill in the message map with macros to bind the messages to functions, you must do something similar to bind the methods and properties to DISPIDs. The macros for automation are shown in the following table:

Macro Name	Description
`DISP_DEFVALUE`	Makes a property the default for the class.
`DISP_FUNCTION`	Defines an automation function.
`DISP_PROPERTY`	Defines an automation property using a member variable.
`DISP_PROPERTY_EX`	Defines a property, plus the `Get`/`Set` functions used to access it.
`DISP_PROPERTY_NOTIFY`	Defines a property, plus a notification function which gets called when it's changed.
`DISP_PROPERTY_PARAM`	Defines a property, using `Get`/`Set` functions and an index parameter.

Let's examine each of these in turn:

DISP_DEFVALUE

This macro makes one property the 'default property' for the class, the one which is retrieved or set when a reference to an object doesn't specify a method or property. The macro looks like this:

```
DISP_DEFVALUE(theClass, szPropName)
```

DISP_FUNCTION

You use `DISP_FUNCTION` to add an automation method to the appropriate automation class. The macro prototype looks like this:

```
DISP_FUNCTION(theClass, szExternalName, pfnMember, vtRetVal, vtsParam)
```

The macro takes several parameters. First is the name of the `CCmdTarget`-derived class that represents the automation class. The second parameter is the external name of the method as seen by automation controllers (such as Visual Basic). The third parameter is the name of the C++ member function which represents the automation method (this is the function that is actually called when a controller calls the `IDispatch::Invoke()` function). The fourth parameter determines what type of value is returned from the C++ member function; it's a variant tag, of type `VARTYPE`, so it can take values such as `VT_I2` (a `short`), `VT_I4` (a `long`) and `VT_VARIANT` (a `VARIANT`).

The last parameter to the `DISP_FUNCTION` macro is a list of one or more variant tag strings to represent the types of the parameters passed to the actual automation method. The reason they are strings and not integers (like the variant tags used for the return type) is because there's no way for MFC to know how many parameters a given function will have. A macro can only be defined with a fixed number of parameters, so to allow for any number of *function* parameters, the last *macro* parameter is a string that MFC splits apart to get at each of the function's parameter types.

It's easy to specify the parameter list because adjacent string literals are concatenated if nothing appears between them. Thus, a list of string literals separated by nothing more than white space is, in fact,

equivalent to a single string. This is ANSI standard behavior, although it's not used very often. For example, this line of code results in **strMyString** having a value of **"Johnny & Matt & Jerry"**:

```
CString strMyString = "Johnny &" "Matt &" "Jerry";
```

In passing the strings to the last parameter of the **DISP_FUNCTION**, you accomplish the same thing, except you use the string constants defined by MFC. You can find a list of these string constants (which all begin with **VTS_**) in the online documentation provided with VC++. For example, if I want to pass three parameters to my automation method, a Boolean, a string and an integer, I'd write the entry like this:

```
DISP_FUNCTION(CMyClass, "aFunc", Func, VT_I2, VTS_BOOL VTS_BSTR VTS_I4)
```

This creates a string that looks like **"\x0B\x0E\x03"**, since **VTS_BOOL** represents **"\x0B"**, **VTS_BSTR** represents **"\x0E"** and **VTS_I4** represents **"\x03"**. MFC can later parse this string and determine whether the parameter values passed to the **IDispatch::Invoke()** function are correct.

DISP_PROPERTY

The next macro, **DISP_PROPERTY**, is used to expose a property of an automation class. This is the macro definition:

```
DISP_PROPERTY(theClass, szExternalName, memberName, vtPropType)
```

The first parameter is the name of the **CCmdTarget**-derived class (which represents the automation class). The second parameter is the name that the automation controllers will use to identify this property. The third parameter is the name of the data member in the automation class that is set or retrieved when **IDispatch::Invoke()** is called. The last parameter is the type of the property. You must pass it one of the variant tags that we saw earlier.

So, as an example, we might specify,

```
DISP_PROPERTY(CMyClass, "result", m_Result, VT_I2)
```

which will provide **CMyClass** with a property called **result**, bound to a two-byte integer data member called **m_Result**.

DISP_PROPERTY_EX

The **DISP_PROPERTY_EX** macro is also similar to the **DISP_PROPERTY** macro. The difference is that this macro allows you to specify a function that is called to set the value of the property and another function to return its value. How these are actually implemented is entirely up to you. The property doesn't even have to be represented by a data member in the automation class; it could be in a database field across the LAN, for example.

The purpose of this macro is to give you more control over the property. That way, a controller never has direct control of the internal data representing the property, because it's always your function that is called to set or return the value. This, of course, is useful for data validation, as well as allowing you to use a different internal representation of the data than the one you hand out to automation controllers.

The prototype of the **DISPLAY_PROPERTY_EX** macro is:

```
DISP_PROPERTY_EX(theClass, szExternalName, pfnGet, pfnSet,
   vtPropType)
```

So we might use it like this:

```
DISP_PROPERTY_EX(CMyClass, "value", GetValue, SetValue,
   VT_I2)
```

DISP_PROPERTY_NOTIFY

DISP_PROPERTY_NOTIFY is very similar to **DISP_PROPERTY**, except that it allows you to pass it the name of a function which will be called when the property is changed by the automation controller. This function should take no parameters and return no value, and will be called immediately after the controller alters the value of the property, thus allowing the server to be notified automatically of changes.

The prototype of the macro is:

```
DISP_PROPERTY_NOTIFY(theClass, szExternalName, memberName, pfnAfterSet,
   vtPropType)
```

So we could use it like this:

```
DISP_PROPERTY_NOTIFY(CMyClass, "aProperty", m_Prop, PropNotify, VT_I2)
```

The function **void CMyClass::PropNotify()** will be called whenever a controller modifies **aProperty**.

DISP_PROPERTY_PARAM

The last of the entry macros is the **DISP_PROPERTY_PARAM** macro. Again, this macro is very similar to the **DISP_PROPERTY_EX** macro, but it allows you to specify parameters that should be passed to the **Get()** and **Set()** functions. This is the prototype for this macro:

```
DISP_PROPERTY_PARAM(theClass, szExternalName, pfnGet, pfnSet, vtPropType,
   vtsParams)
```

If we used the macro like this,

```
DISP_PROPERTY_PARAM(CMyClass, "aVal", GetVal, SetVal, VT_DISPATCH, VTS_I2
   VTS_I2)
```

we'd be defining a property **aVal**, which is of type **VT_DISPATCH**, and which is accessed by **Get**/**Set** functions, each of which take two parameters of type **VT_I2**. This would correspond to the function prototypes:

```
LPDISPATCH CMyClass::GetVal(short m, short n);
void CMyClass::SetItem(short m, short n, LPDISPATCH newVal);
```

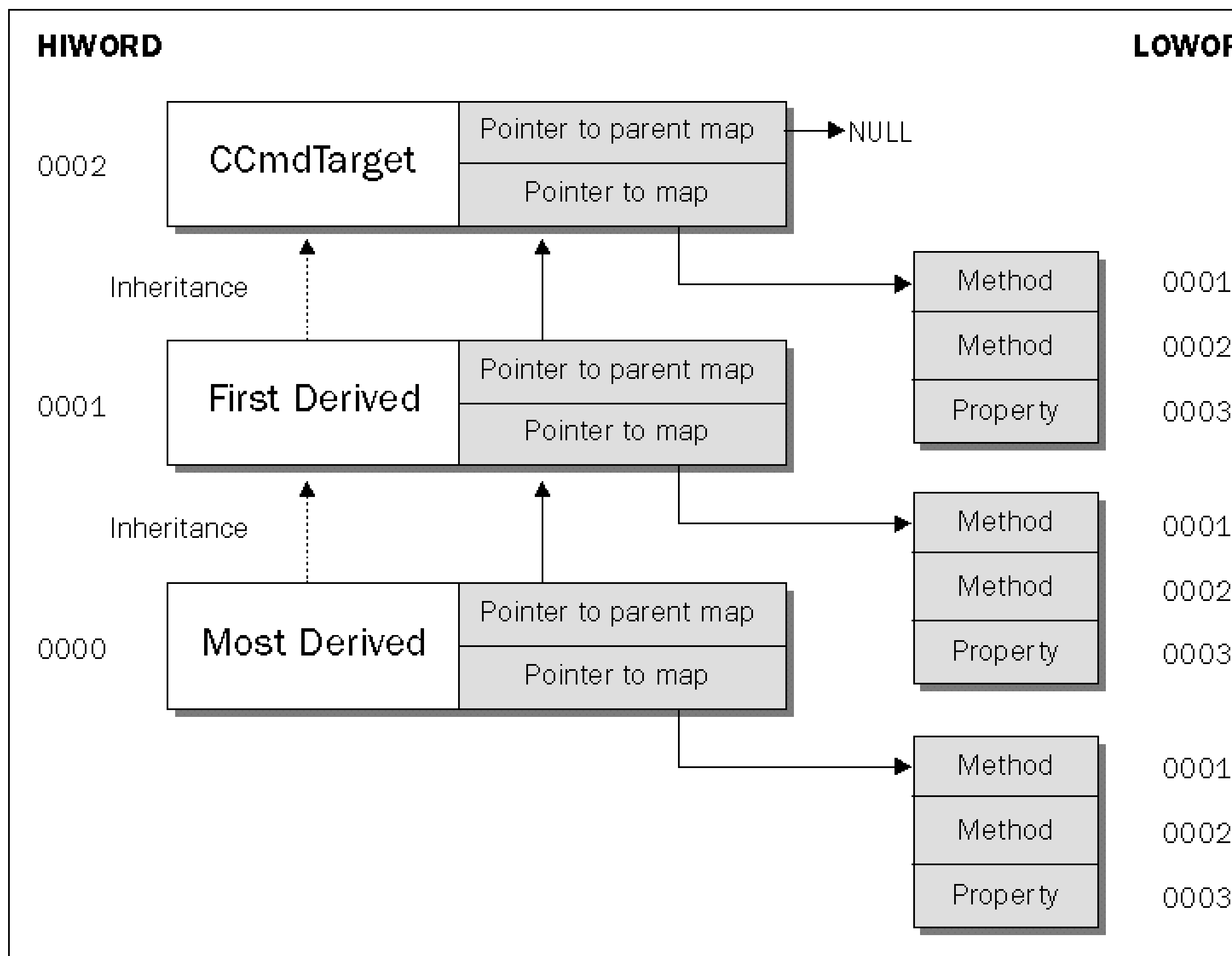
Now, the bad news is that you must use these macros when you're adding methods and properties to your dispatch maps. The good news is that if you use ClassWizard, you can get it to do all of the work for you. It adds the entries to your dispatch maps, creates the function declarations in the header files, adds the function bodies to your implementation files and keeps the **.odl** file up to date.

DISPID Assignment

If you've been following along, you should have noticed that these macros don't allow you to specify the DISPID for the methods or properties. Why is that? It turns out that MFC assigns the DISPIDs automatically for you, when the `IDispatch::GetIDsOfNames()` function is called.

When you use any of the above mentioned macros, MFC stores the information as an entry in the dispatch map. Part of this information includes the DISPID, but MFC sets this member to `DISPID_UNKNOWN`, which is `-1`. Bear in mind that dispatch maps are just like message maps in the sense that, if a method or property is not found in the most derived class, its parent classes are searched until the hierarchy has been exhausted, or the method or property is found.

To optimize the performance of `IDispatch::Invoke()`, MFC uses a pretty slick searching scheme. The DISPID for each method or property is divided into two words. The upper word determines the dispatch map that contains the method (the most derived class being zero, and each parent class incremented by one), and the lower word contains the index in the map of the method or property.



When **IDispatch::GetIDsOfNames()** is called, MFC returns a **DISPID** which contains both the index of the appropriate dispatch map and the index within the dispatch map containing the method or property. Since each method or property is identified only by its name, **GetIDsOfNames()** must resolve the name into its DISPID by performing a search through the dispatch maps. This function might take a few seconds if the class hierarchy is large and if there are many methods and properties to check. However, by the time that **Invoke()** is called, the controller should already have the appropriate DISPID, so the execution of the method or property will be speedy.

Invoking Automation Methods and Properties Using MFC

When **IDispatch::Invoke()** is called, MFC's **COleDispatchImpl::Invoke()** handles the call. The function uses the high word to go directly to the correct dispatch map. It then uses the low word to jump directly to the appropriate method or property within the dispatch map. If the entry is a method or a property with **Get()** and **Set()** functions, the functions are executed. If the entry is a plain property (with no functions for setting or getting it), **Invoke()** uses another information field (stored in the entry) to determine the offset from the beginning of the class to set or get the property. This offset field is initialized when the entry is created using the **DISP_PROPERTY** macro.

As for the other **IDispatch** members, Visual C++ 4.x now implements both **GetTypeInfo()** and **GetTypeInfoCount()**. These functions return information based on the type libraries cached or registered with the server.

The Automation Server

In my own work, I had a situation where I needed to create a server that could maintain information and share it with several clients at a time. I also wanted to maintain the information within an array inside the server. The data maintained by the server is never accessed directly by the clients, only by the server, but the data still needs to be maintained correctly so that all clients can display the data by calling a method in the server.

I knew that the server wouldn't need to contain a user interface (windows, controls, and so on), so I thought that putting the code in an in-process server would make the most sense. The problem is that, since an in-process server is run within the process space of its client, I wouldn't be able to share the data.

If only I could find a way to share the strings with the other clients... then I remembered reading somewhere that I could share data using memory-mapped files, so I set out immediately to create my in-process server. You can find the completed server on the CD as **TestServ**.

Inside **InitInstance()**, I created and initialized the memory map object using the following code:

```
BOOL CTestservApp::InitInstance()
{
   AfxMessageBox(_T("InitInstance was just called."));
   // Register all OLE server (factories) as running.  This enables the
   //  OLE libraries to create objects from other applications.
   COleObjectFactory::RegisterAll();

   m_hFileMap = OpenFileMapping(FILE_MAP_ALL_ACCESS, TRUE, strFileMap);
   if (m_hFileMap == NULL)
   {
      m_hFileMap = CreateFileMapping((HANDLE)0xFFFFFFFF, NULL,
                                PAGE_READWRITE, 0, MAP_SIZE, strFileMap);
```

```
        if (m_hFileMap == NULL)
        {
           AfxMessageBox(_T("Can't create file mapping object."));
           return FALSE;
        }

        BYTE* lpView = (BYTE*)MapViewOfFile(m_hFileMap,
           FILE_MAP_READ | FILE_MAP_WRITE, 0, 0, 0);

        *((UINT*)lpView) = 0;

        UnmapViewOfFile((LPVOID)lpView);
     }

     return TRUE;
  }
```

If the file mapping can't be opened, it must be because it hasn't already been created, so the code here will make sure that the mapping is both created and opened. Notice that I set the first **`UINT`** of data to zero. This **`UINT`** will be used to determine the number of strings within the memory mapped file. Then I simply close the handle inside of **`ExitInstance()`**. Since **`InitInstance()`** and **`ExitInstance()`** will always be called for each process that creates an automation object from this server, the server will always be able to get at the **`m_hFileMap`** member and know that it holds the correct value.

`AddText()` checks the memory-mapped file for the given key. If it finds it, it simply replaces the string value with the passed in string value. If it doesn't find it, the function adds a new string to the end of the memory mapped file. This is the code listing for the function:

```
void CMyObj::AddText(LPCTSTR strKey, LPCTSTR strValue)
{
   CMutex MapMutex(FALSE, strTextMutex);
   MapMutex.Lock();

   CTestservApp* pApp = (CTestservApp*)AfxGetApp();
   BYTE* lpView = (BYTE*)MapViewOfFile(pApp->m_hFileMap,
                  FILE_MAP_READ | FILE_MAP_WRITE, 0, 0, 0);
   BYTE* lpTemp = lpView;

   UINT nCount = *((UINT*)lpView);
   // Skip over the count.
   lpView += sizeof(UINT);

   if (lpView != NULL)
   {
      // Search for the string first.
      int nIndex = 0;
      int nBytes = 0;
      BOOL bFound = FALSE;
      while (nBytes < MAP_SIZE && nIndex < nCount)
      {
         if (strcmp((LPTSTR)lpView, strKey) == 0)
         {
            bFound = TRUE;
            // Skip over the key and point right to the value.
            nBytes += strlen((LPTSTR)lpView) + 1;
            lpView += nBytes * sizeof(TCHAR);
            break;
         }
         // Skip this key.
         nBytes += strlen((LPTSTR)lpView) + 1;
         lpView += nBytes * sizeof(TCHAR);
         // Skip this value.
         nBytes += strlen((LPTSTR)lpView) + 1;
         lpView += nBytes * sizeof(TCHAR);
         nIndex++;
```

```
      }

      // if we found it,
      if (bFound)
      {
         if (MAP_SIZE <= nBytes + strlen(strValue) + 1 /* for the NULL */)
            strcpy((LPTSTR)lpView, strValue);
      }
      // else add it to the end.
      else
      {
         // Since we didn't find it, we should be pointing right to the
         // next empty spot.
         if (MAP_SIZE >= nBytes + strlen(strKey) + strlen(strValue)
            + 2 /* for the NULLs */)
         {
            strcpy((LPTSTR)lpView, strKey);
            nBytes += strlen((LPTSTR)lpView) + 1;
            lpView += nBytes * sizeof(TCHAR);
            strcpy((LPTSTR)lpView, strValue);
            nCount++;
            *((UINT*)lpTemp) = nCount;
         }
      }
   }
   else
      AfxMessageBox(_T("Could not map to view"));

   UnmapViewOfFile((LPVOID)lpView);
   MapMutex.Unlock();
}
```

This technique works well with values of a fixed size, rather than variable length items like strings. Since we're dealing with strings, the possibility of data corruption definitely exists, but since the whole purpose of this exercise is to introduce a few concepts and introduce the techniques, I'll let the potential bugs slip by for now.

The **`DisplayText()`** function opens a view to the memory-mapped file, looks for the given key and displays the associated value using a message box. Here's the code:

```
void CMyObj::DisplayText(LPCTSTR strKey)
{
   CMutex MapMutex(FALSE, strTextMutex);
   MapMutex.Lock();

   CTestservApp* pApp = (CTestservApp*)AfxGetApp();
   BYTE* lpView = (BYTE*)MapViewOfFile(pApp->m_hFileMap,
                        FILE_MAP_READ | FILE_MAP_WRITE, 0, 0, 0);

   UINT nCount = *((UINT*)lpView);
   // Skip over the count.
   lpView += sizeof(UINT);

   if (lpView != NULL)
   {
      // Search for the string first.
      int nIndex = 0;
      int nBytes = 0;
      BOOL bFound = FALSE;
      while (nBytes < MAP_SIZE && nIndex < nCount)
      {
         if (strcmp((LPTSTR)lpView, strKey) == 0)
         {
            bFound = TRUE;
            // Skip over the key and point right to the value.
            nBytes += strlen((LPTSTR)lpView) + 1;
            lpView += nBytes * sizeof(TCHAR);
```

```
            break;
         }
         // Skip this key.
         nBytes += strlen((LPTSTR)lpView) + 1;
         lpView += nBytes * sizeof(TCHAR);
         // Skip this value.
         nBytes += strlen((LPTSTR)lpView) + 1;
         lpView += nBytes * sizeof(TCHAR);
         nIndex++;
      }

      CString str;
      // if we found it,
      if (bFound)
         str = (LPTSTR)lpView;
      else
         str = _T("Key was not found!");

      AfxMessageBox(str);
   }
   else
      AfxMessageBox(_T("Could not map to view"));

   UnmapViewOfFile((LPVOID)lpView);
   MapMutex.Unlock();
}
```

The in-process server is called **`Testserv.dll`** and is registered as **`TestServ.MyObj`**, so the client application (which we'll meet in the next section) can create a server object as follows:

```
m_Obj.CreateDispatch(_T("TestServ.MyObj"));
```

The techniques introduced in this section will become very useful in the next chapter, where we'll actually use automation and in-process servers to implement a complete business object model with metadata and data objects.

Implementing a Controller

The key to writing a successful automation controller is to know as much as possible about the server automation classes and their methods and properties. Remember that a single server can expose several automation classes. These classes are sometimes documented by placing information about them in the **type library** files we looked at earlier in the chapter.

Once you determine the automation classes and the methods and properties you want to use, it's time to write some code. The easiest way involves reading the type library using ClassWizard. ClassWizard will then generate a class derived from an MFC class named **`COleDispatchDriver`**.

This class encapsulates the **`IDispatch`** pointer needed to communicate with an automation object. It also has several functions for locating and creating the object, calling methods of the object and setting or retrieving properties of the object. These functions include **`CreateDispatch()`**, **`AttachDispatch()`**, **`DetachDispatch()`**, **`ReleaseDispatch()`**, **`InvokeHelper()`**, **`GetProperty()`** and **`SetProperty()`**. We'll now look briefly at each of these.

There are two versions of **`CreateDispatch()`**. One takes a CLSID and (optionally) a pointer to a **`COleException`** object and the second uses the ProgID instead of the CLSID.

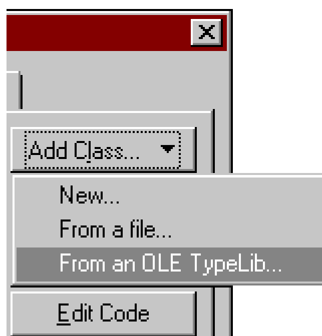
```
BOOL CreateDispatch(REFCLSID clsid, COleException* pError = NULL);
BOOL CreateDispatch(LPCTSTR lpszProgID, COleException* pError = NULL);
```

The version of **CreateDispatch()** that receives the ProgID instead of the CLSID simply searches through the registry for the corresponding CLSID, then calls the other version of the function with that value. The first version of the function then goes on to call **CoCreateInstance()** with the CLSID, retrieving the **IDispatch** interface from the object and attaching it to the **COleDispatchDriver** object, ready to call any methods or properties.

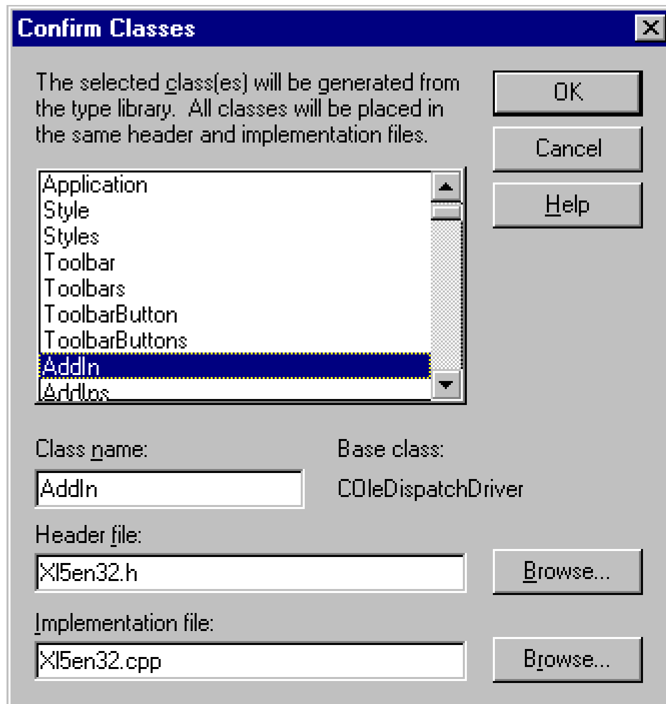
The next function, **AttachDispatch()**, allows you to attach a **COleDispatchDriver** (or **COleDispatchDriver**-derived) object to an **IDispatch** obtained from some other source (such as from another object). The **DetachDispatch()** function has the opposite effect. Once the interface has been detached, the connection between the **COleDispatchDriver** object and the dispatch interface is broken, so deleting the C++ object will have no effect on the interface or its associated object, for example.

ReleaseDispatch() directs the **COleDispatchDriver** class to call **Release()** on the **IDispatch** interface attached to the object and stop storing the pointer to **IDispatch**. (Note the difference between **ReleaseDispatch()** and **DetachDispatch()**. The latter doesn't call **Release()** on the pointer before setting it to **NULL**.) The C++ class is still available and you can call **CreateDispatch()** or **AttachDispatch()** again. Note that **ReleaseDispatch()** is also called from the object's destructor.

The last three functions are used to invoke functions and properties on the interface pointer attached to the C++ object. It manages the values passed in and generates **VARIANT** objects for use by the **IDispatch::Invoke()** function in the automation server.
When you use ClassWizard to generate a class based on **COleDispatchDriver**, you select the Add Class... From an OLE TypeLib option.



ClassWizard will then allow you to browse for the type library of your choice. Once you select the appropriate type library, ClassWizard will read through it and display a list of the classes it finds. It then gives you the option of selecting the ones for which you want to create C++ wrapper classes. These classes will be generated and derived from **COleDispatchDriver** and they will also have the appropriate member functions for calling methods and properties of the automation object. You can then make calls to the methods and properties of this class, as if you're calling the methods in the automation object directly. Of course, you'll need to instantiate one with the **COleDispatchDriver::CreateDispatch()** function first.



The member functions for each method or property you can call that ClassWizard generates in your class call **COleDispatchDriver::InvokeHelper()**, **COleDispatchDriver::GetProperty()**, or **COleDispatchDriver::SetProperty()** with the appropriate parameters. Eventually, **COleDispatchDriver** calls **IDispatch::Invoke()**. Here you can see the type of function that ClassWizard can generate:

```
void CMyObj::AddText(LPCTSTR strKey, LPCTSTR strValue)
{
   static BYTE parms[] = VTS_BSTR VTS_BSTR;
   InvokeHelper(0x1, DISPATCH_METHOD, VT_EMPTY, NULL, parms, strKey,
                strValue);
}
```

This function was generated for the **MyObj** automation class of the **Testserv.dll** server. Note that the function call already has the appropriate parameters and DISPID, and it knows whether it's calling a method or setting or retrieving a property. The function prototype also has the appropriate parameter names and types for receiving the values which will eventually get passed on to the automation method. The values are received in a C++ fashion.

Of course, anything ClassWizard can do, you can do too. You could create a class derived from **COleDispatchDriver** yourself and implement the functions for calling the methods and properties inside them. Maybe you could even call **IDispatch::GetIDsOfNames()** from within the constructor of the **COleDispatchDriver** derived class. You'll have to pick this method of creating your controller if you don't have a type library to work with. If this is the case, the manufacturer has to at least provide you with some written documentation for the server. The information should contain the names of the classes exposed by the server, along with their methods, properties, parameter types, and so on.

The Automation Client

Now that we have enough information about automation controllers, we can concentrate on the client side of the example that I started in the server section. You can find the completed project on the CD as **TestClnt**.

I started out by letting ClassWizard do its thing with the server's type library to generate a class derived from **COleDispatchDriver** with the appropriate member functions to represent the methods and properties of the automation class.

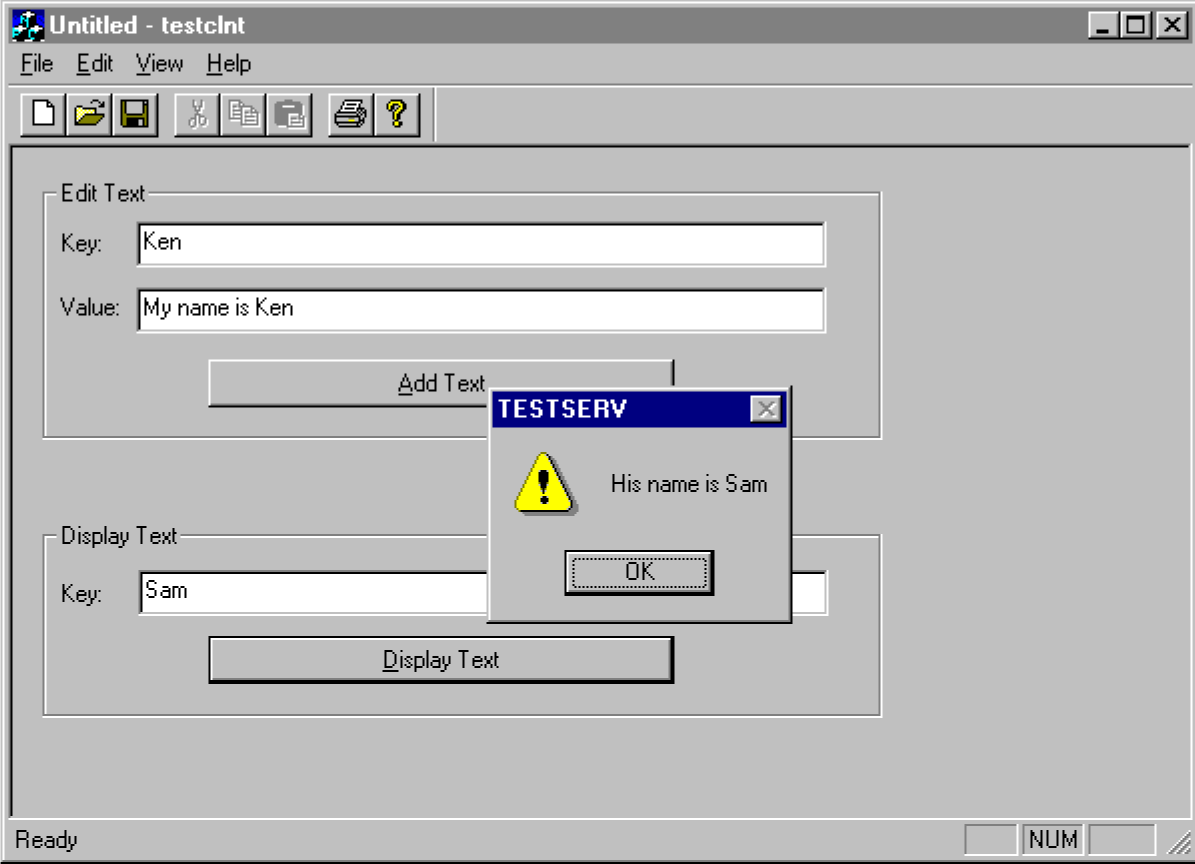
I then instantiated an object, **m_Obj**, of the derived class in my view class (where I'll be making heavy use of it):

```
class CTestClntView : public CFormView
{
protected: // create from serialization only
   CTestClntView();
   DECLARE_DYNCREATE(CTestClntView)

// Attributes
public:
   CTestClntDoc* GetDocument();
   CMyObj m_Obj;

// Implementation
public:
   virtual ~CTestClntView();
   DECLARE_MESSAGE_MAP()
};
```

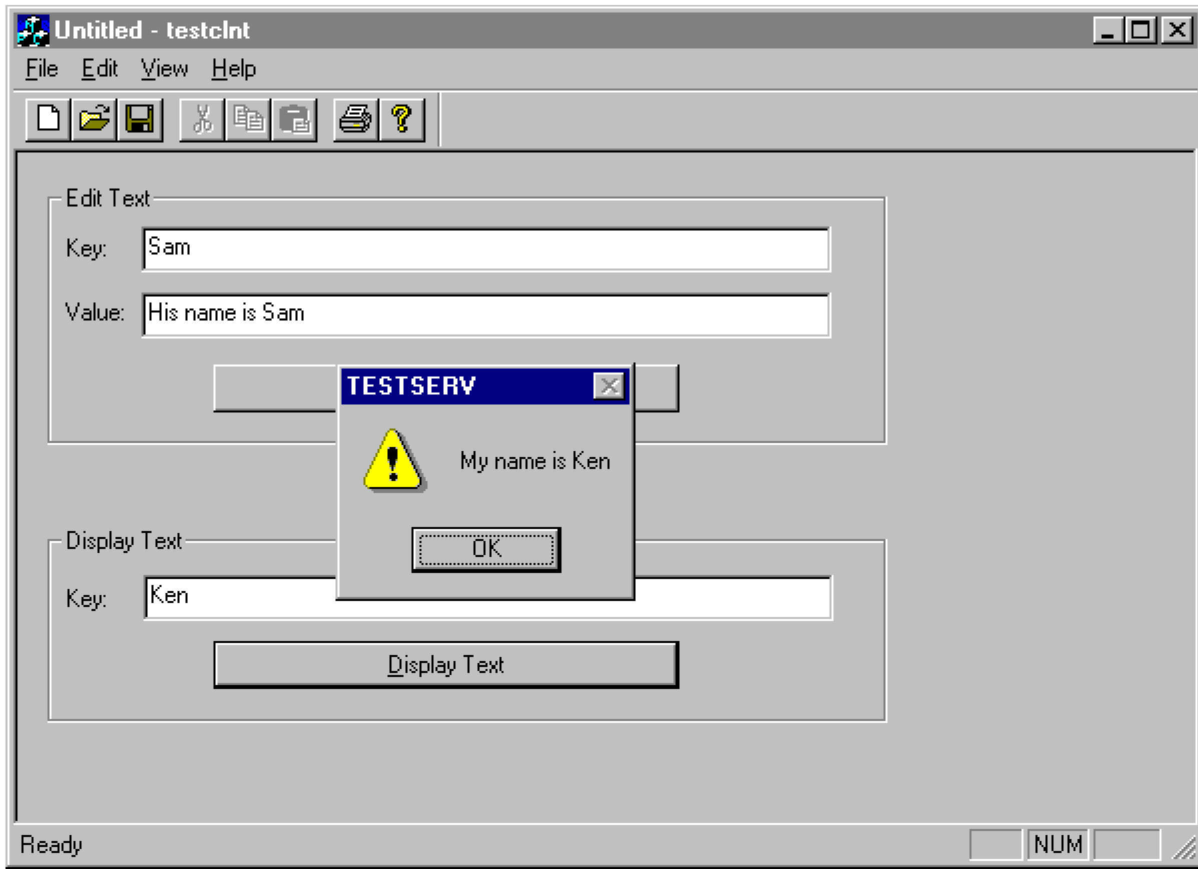
I call the **CreateDispatch()** function from the view's constructor like this:

```
CTestClntView::CTestClntView()
   : CFormView(CTestClntView::IDD)
{
   ...

   m_Obj.CreateDispatch(_T("TestServ.MyObj"));
}
```

My view class is derived from **CFormView** and I used some controls to enter the key and its value, and to add the text to the server and display the text. If I run several instances of my client, you'll notice that I can add text values from one client and display the values from another.



Untitled - testclnt
File
Edit
View
Help
Edit Text
Key:
Ken
Value:
My name is Ken
Add Text
TESTSERV
His name is Sam
OK
Display Text
Key:
Sam
Display Text
Ready
NUM



The instance on the top added the key and value for **`"Ken"`**, and the one below was used to enter the key and value for **`"Sam"`**, yet they could display the values created by the other as if they had entered it themselves.

When it's time to invoke methods on the server, I simply call the member functions that ClassWizard generated for me. Since the object is a member of my view class, when my view is destroyed, the **`COleDispatchDriver`**'s destructor is called and the **`IDispatch`** interface pointer is released.

OLE Controls

If automation doesn't provide you with enough features for writing software components, you'll have to move up to the next level: OLE controls (or OCXs, as they have been called for the longest time).

OLE controls are a much more powerful replacement for Windows custom controls and Visual Basic extensions. This new generation of control is built on top of several OLE interfaces to provide solutions in very small and fast packages. Each control is usually packaged as a DLL but with an OCX extension. The control is capable of exposing class factories, its type information, methods, properties, and events it can fire.

The controls can live with or without a user interface, allowing you to provide a solution, for example, to read data from an I/O port and generate an event whenever data arrives. You could then drop this control onto any window, giving the window the ability to read from the particular I/O port. This is just one example of the power that OLE controls can provide.

Generating one is pretty simple too. You just create a new project using the OLE Control Wizard, select the appropriate choices and bang, you're done. Then you use ClassWizard to add properties, methods and events. Add some drawing code and functionality, and you have your control. You can even add property pages, to be displayed by a client window when the user adding the control to their application wants to change its properties.

There are now a large number of applications on the market that can host OLE controls. This includes everything from Access to Visual Basic, and Visual C++ even comes with a test container that allows you to test your control in the development stage. This can be accessed from the OLE Control Test Container in Developer Studio's Tools menu.

With Visual C++ 4.x, you can also add OLE controls to your applications, and you get support from MFC (along with AppWizard and ClassWizard) to provide the necessary code to contain the control within a **CWnd**-derived class. Before Visual C++ 4.x, you could have written your own container support, but it wasn't easy. Now MFC and its related tools make the whole process easy. You just use the Component Gallery to add a control to your application. Once that's done, you'll even have access to the control from Developer Studio's control palette, ready to include it in any dialog box. Adding the control to the project also causes ClassWizard to display the control's receivable events in the message map tab.

Microsoft provides an easy way of adding OLE controls to dialog boxes, but gives very minimal support for adding them to any other type of window. We'll discuss how we can add this support ourselves. Along the way, we'll create an OLE timer control and pick up a few more techniques.

OLE Controls and Events

To understand events, let's consider for a second what happens with regular controls. When you have a control, such as a button, sitting on your window, you usually want to be informed when something you're interested in happens. For example, if the user clicks the button, you want to be notified so that you can perform a specific task.

As soon as the user clicks the button, Windows generates a message called **WM_COMMAND** (this is usually the message sent whenever a control needs to notify its parent of an event). Window messages usually carry two pieces of information: a **WPARAM** and an **LPARAM**. In the case of the button being clicked, the high

word of the **WPARAM** contains **BN_CLICKED**, which identifies what's happened. The low word of the **WPARAM** contains the identifier of the button, and the **LPARAM** contains the window handle of the button.

In your application, you would catch this event by adding an event handler for the Windows message. Now, when we have an OLE control, we still want to receive events, but since the control and its container communicate using OLE, window messages don't apply. Somehow, we have to make the OLE events appear as if they are window messages.

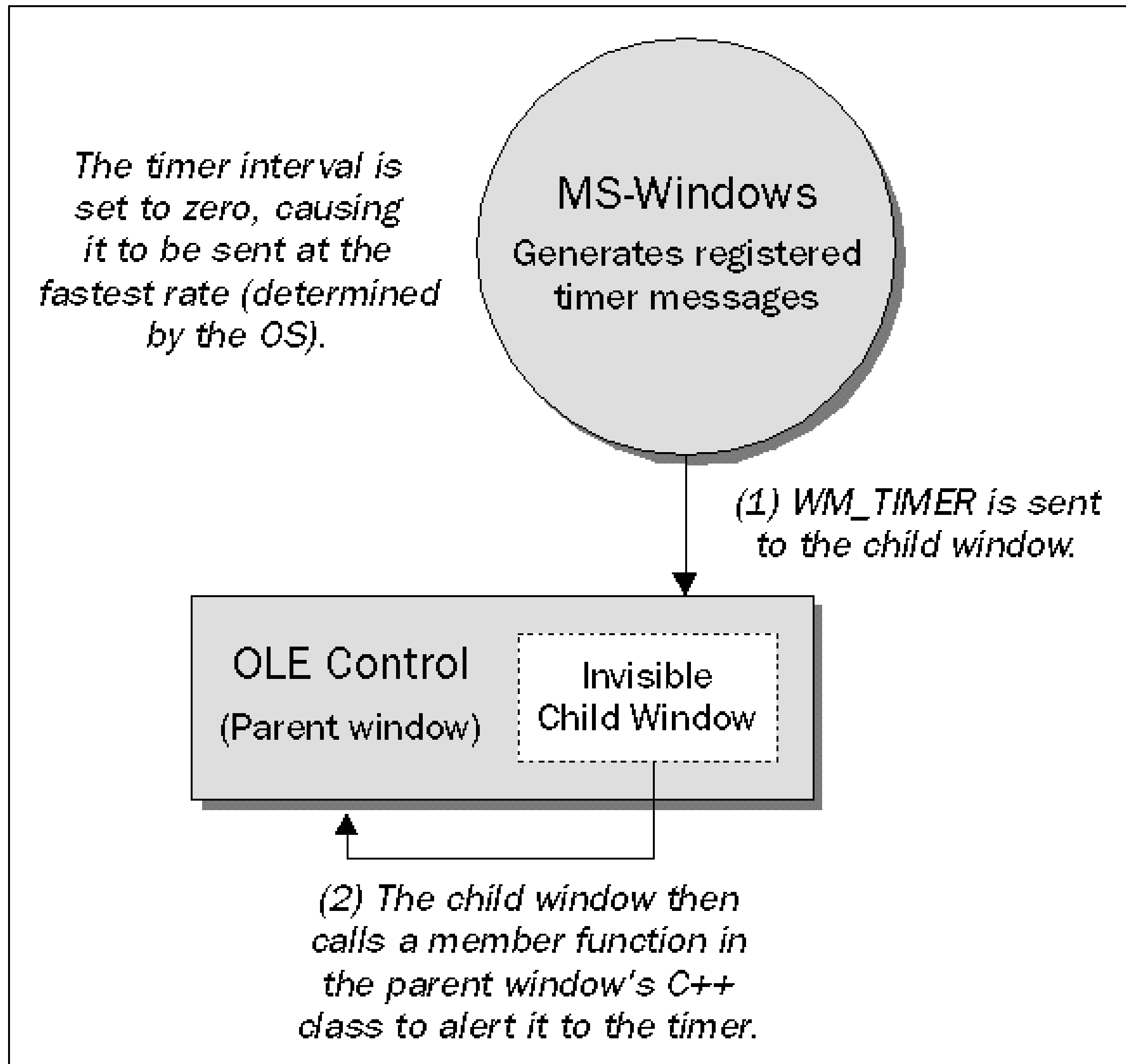
Underneath the covers, the container has to provide an automation object and pass its **IDispatch** interface pointer to the OLE control. The automation object's methods should fit the control's requirements (we'll see how this is achieved a little later). When the control is ready to send an event, it will call the appropriate method within the container's automation object.

This all starts when you create your control. You use ClassWizard to create the events that the control will fire. ClassWizard then adds an identifier for the event (similar to the window message ID), and generates a member function that the control can call whenever it's ready to fire the event.

Once you add an event to the OLE control, ClassWizard generates a function for you. You have the option of having parameters passed with the event and the event can return a value. Keep in mind that when you fire an event, you're really calling an automation method in the container application. This function can receive parameters and return a value, just like any other function.

About the Example Code

The example I chose to demonstrate OLE controls and their containers implements a timer. In my example, I only wanted one timer resource created, regardless of how many instances of the control were created. At first, I tried to create an invisible child window of the control's window and receive the timer messages in the child window, as shown below:

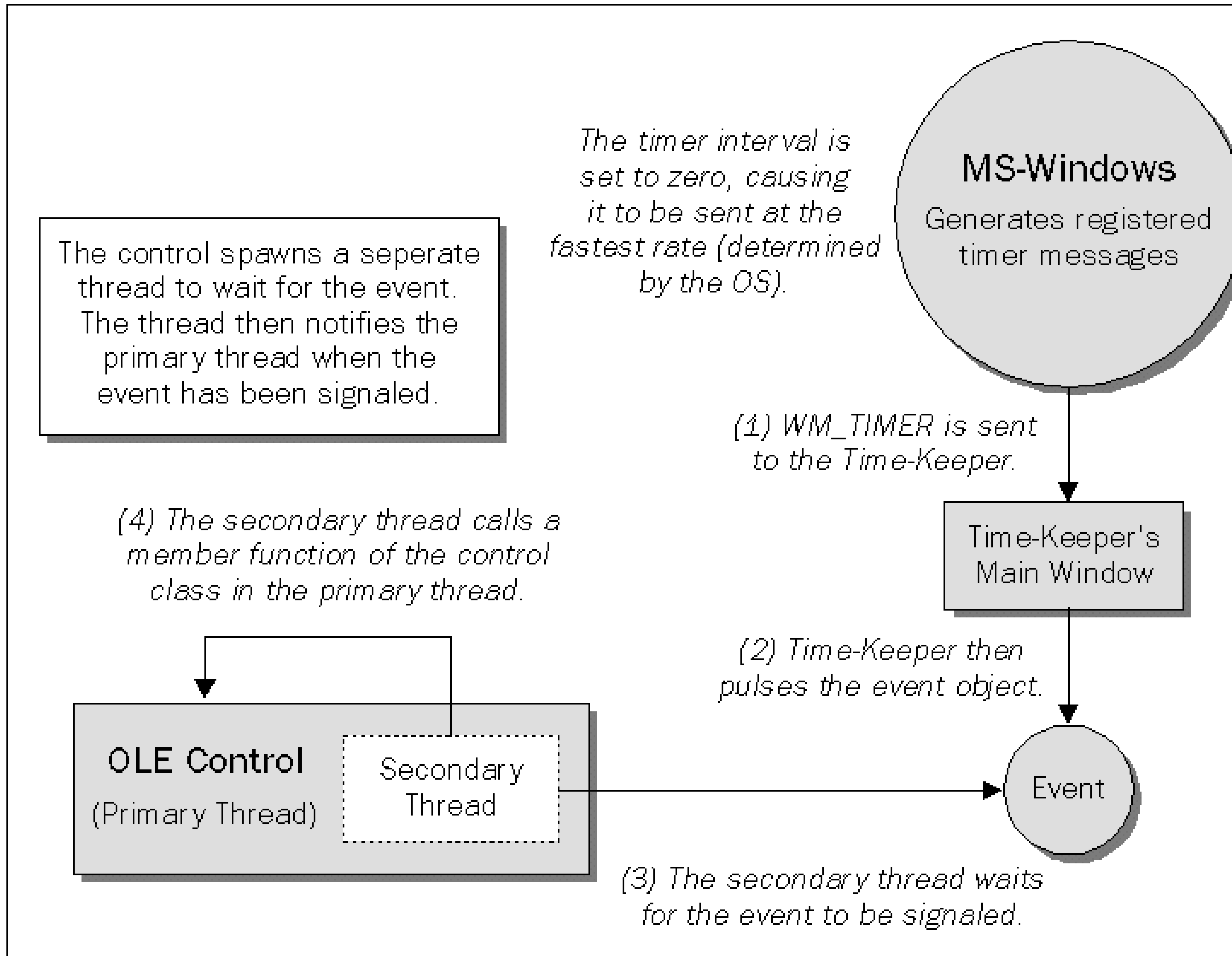


I also created a Win32 **event** object to signal to other instances of the control when the timer messages were received. I had the idea that I could initialize a single child window and timer for the first instance of the control, and every subsequent instance after the first would be signaled via a Win32 event object whenever the child window received a message.

This worked fine until the container application of the first instance went down. Since the message loop was owned by the primary thread of the first application to create an instance of the control, when that application closed down, it took its thread, message pump and timer messages with it.

My next thought was to have a separate 'hidden' application to control the timer messages from its main window, and pulse an event object whenever the timer messages were received. Then, the first instance of the control could load the hidden application (which I call the Time-Keeper), and the other instances would automatically sync themselves to the event. The last instance of the control to be closed would have to close down the Time-Keeper, so I'd have to keep track of the number of instances with a global variable within the OLE control. This variable would also need to be synchronized for completeness,

since several threads could potentially access this variable at the same time. The figure below shows the new layout:



If you look closely at the diagram, you'll notice that I also used multithreading to accomplish the task. I always say that you should never use multithreading in your application unless it's absolutely necessary. This is a perfect situation where it *is* necessary, since the thread must go to sleep and be woken up only when the event has been signaled. I definitely couldn't put my primary thread to sleep, since I needed that one for the user interface. So, if anyone was going to sleep, it had to be the secondary thread. (This reminds me of the problems I sometimes have getting my kids to sleep; I know they wish that they could get the primary thread, me, to sleep first!)

Don't confuse the event object, which is a Win32 Kernel object, with the events that are fired from a control to its container, which we'll discuss shortly. The name is the same, but the concepts involved are slightly different.

The OLE Control

Let's look at some of the issues involved in implementing an OLE control. You can find the code for this example in **\Timer\Timer.mdp** on the CD. You'll also find the code for the Time-Keeper in the **\Timer\TimeKeeper** subdirectory and a test application that uses the timer control in **\Timer\TestBed**.

Sending an Event to the Container

I added an event for firing a timer event to the control's container using ClassWizard, without specifying any parameters or return value. This is the function that ClassWizard generated for me:

```
void FireTimer()
   {FireEvent(eventidTimer,EVENT_PARAM(VTS_NONE));}
```

The function calls **COleControl::FireEvent()** to fire the actual event. **FireEvent()** receives the DISPID of the event to be fired. It then uses this DISPID to determine the actual method that should be invoked in the container's automation object.

In my example code, I generate the event upon receiving a call from the secondary thread. When I originally spawn the thread in **CTimerCtrl::OnCreate()**, I pass it the **this** pointer of the control's **COleControl** class, which the secondary thread later uses to call the class's **OnTimer()** message. Here's the code for my secondary thread function:

```
UINT TimerThread(LPVOID lpTimerCtrl)
{
   CTimerCtrl* pCtrl = (CTimerCtrl*)lpTimerCtrl;
   pCtrl->m_ShutDownEvent.ResetEvent();
   while (TRUE)
   {
      pCtrl->m_TimerEvent.Lock();
      pCtrl->OnTimer(ID_TIMER);
      if (pCtrl->m_bShutDown)
      {
         pCtrl->m_ShutDownEvent.SetEvent();
         break;
      }
   }

   return 0;
}
```

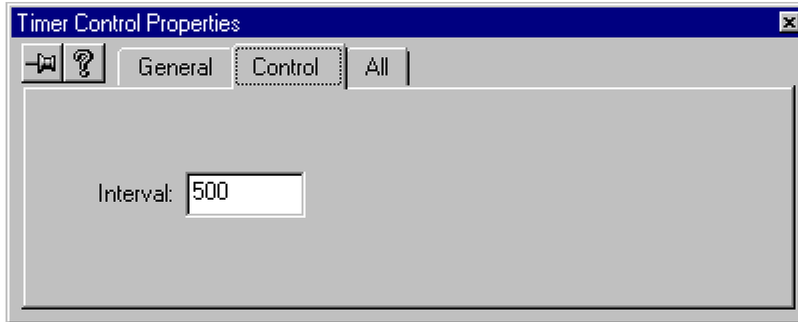
When the control is shut down, it needs to alert the secondary thread so that the thread shuts down too. I do this by setting a flag (represented by the **m_bShutDown** member) which the thread is constantly checking. Once the primary thread alerts the secondary thread, it goes to sleep until the secondary thread has received the notification and signals the **m_ShutDownEvent** object. The primary thread then awakens and continues with its shutdown process.

I gave the individual controls the power of setting their own intervals. As soon as they receive notice of the timer event, they check their internal interval value. If this has been reached, an event is fired to the container. Take a look at the following code:

```
void CTimerCtrl::OnTimer(UINT nIDEvent)
{
   DWORD dwNew = GetTickCount();

   if ((dwNew - m_dwOld) >= (DWORD)m_nInterval)
   {
      FireTimer();
```

```
            m_dwOld = dwNew;
            InvalidateControl();
        }

        COleControl::OnTimer(nIDEvent);
    }
```

`m_nInterval` is a property of the control, which can be set via the control's exposed properties, either through code or via the control's property sheet.



There's still a piece to this puzzle that is missing. How does the container know what methods it should implement, and how does it know the number of parameters and their types? To answer this question, we need to look at `IConnectionPoint` interface and its sidekick, the `IConnectionPointContainer` interface. We won't be looking at it in very great detail, because, luckily, a lot of the work is done for us.

The Connection Point Interfaces

When OLE was first created, an interface called an **advise sink** was put together. This interface allowed an object to notify its client of any changes to its data. The problem with the advise sink technology is that it's not flexible, since interfaces usually are static by nature (meaning that their methods never change). To extend the advise sink interface, you'd have to derive a class from the advise sink interface (`IAdviseSink`) and implement any new functions.

Imagine if every control that wanted to send events or notifications to the containers required the container to derive from the advise sink interface for each and every control. We'd have to implement thousands of advise sink derived interfaces within our containers (it would be havoc).

Its successor, the **connection point**, is much more flexible, since it just defines what possible events the container is interested in receiving. This is done using automation in reverse, with the container implementing methods and properties which are called by the control. So, if the container is interested in receiving events or notifications, it needs to implement some or all of the automation methods, implement an `IDispatch` to call the appropriate function when its `IDispatch::Invoke()` function is called and basically provide the glue between the automation interface and functions.

The control is said to be the **source** of the connection point since it originates the **outgoing** interface call to the container's automation method. The container is said to be the **sink**, since it creates the automation object containing the `IDispatch` interface. Each set of functions are said to be one **connection point**. You can have as many connection points as you wish. Incidentally, connection points are a general extension to COM, so you're not restricted to using them just in OLE controls.

In MFC, all controls implement two connection points. One is used for the events that the control can fire, and the other is used for notifications that can be sent to a container when a property of the control has changed.

Connection points are implemented by two interfaces: **`IConnectionPoint`** and **`IConnectionPointContainer`**. The connection point container interface manages a list of points to which the container would like certain interfaces to connect (pass it an IID of an interface) and the **`IConnectionPointContainer`** will pass back the address of a connection point object that wants to talk to that particular interface. The container also implements a method to allow callers to enumerate all the connection points it supports.

The connection point interface defines several methods, the most important of which are **`Advise()`** and **`Unadvise()`**. **`Advise()`** is called by the interface implementor (the sink) to connect the interface implementation to the connection point, and returns a handle which identifies the connection (because there might be more than one active at a time). If the **`Advise()`** fails, the handle is returned as zero. **`Unadvise()`** is used to break the connection.

The Container

Just as the control needs to support several interfaces, the container has a few interfaces of its own that must be supported to make the marriage work. In MFC, these interfaces are handled by the **`COleControlSite`** class which implements the following interfaces:

`IOleClientSite`
`IOleInPlaceSite`
`IOleControlSite`
`IDispatch` (for events)
`IDispatch` (for properties)
`IPropertyNotifySink`

Generally speaking, this class is hidden from you, as MFC wants to make OLE controls as simple to use as possible, so ClassWizard will provide you with classes and functions to use which hide the complexity. If you look under the surface, though, you'll find that **`COleControlSite`** is the interface through which a container communicates with an OLE control.

The container's dispatch interface for receiving events is implemented within **`COleControlSite`** as a nested class with a class name of **`XEventSink`** and represented by a member called **`m_xEventSink`**. This is the member that will eventually be passed to the control so that the control can call the container when an event occurs.

This event sink is nothing more than an **`IDispatch`** implementation and, as we've already seen, **`IDispatch`** interfaces are handled by dispatch maps in MFC. In fact, events are handled by a variation on this theme called, you guessed it, an **event map** (yes, another map and more macros, which we'll meet shortly). Event maps are built into **`CCmdTarget`**, so that any class derived from **`CCmdTarget`** can act as an event sink, and you'll find that when a control issues an event, the event ends up being processed in **`CCmdTarget::OnEvent()`**.

This function performs a lookup in the event map and executes the appropriate method if an entry exists for it. Just as you don't have to respond to every possible Windows message, you don't have to respond to every event either. When the lookup is performed, if there are no entries for the event identifier,

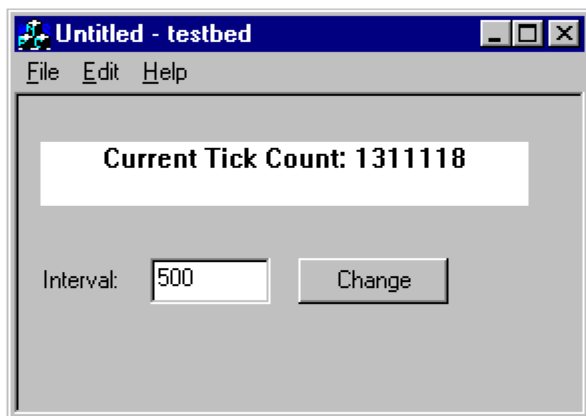
`OnEvent()` simply returns.

Using an OLE Control

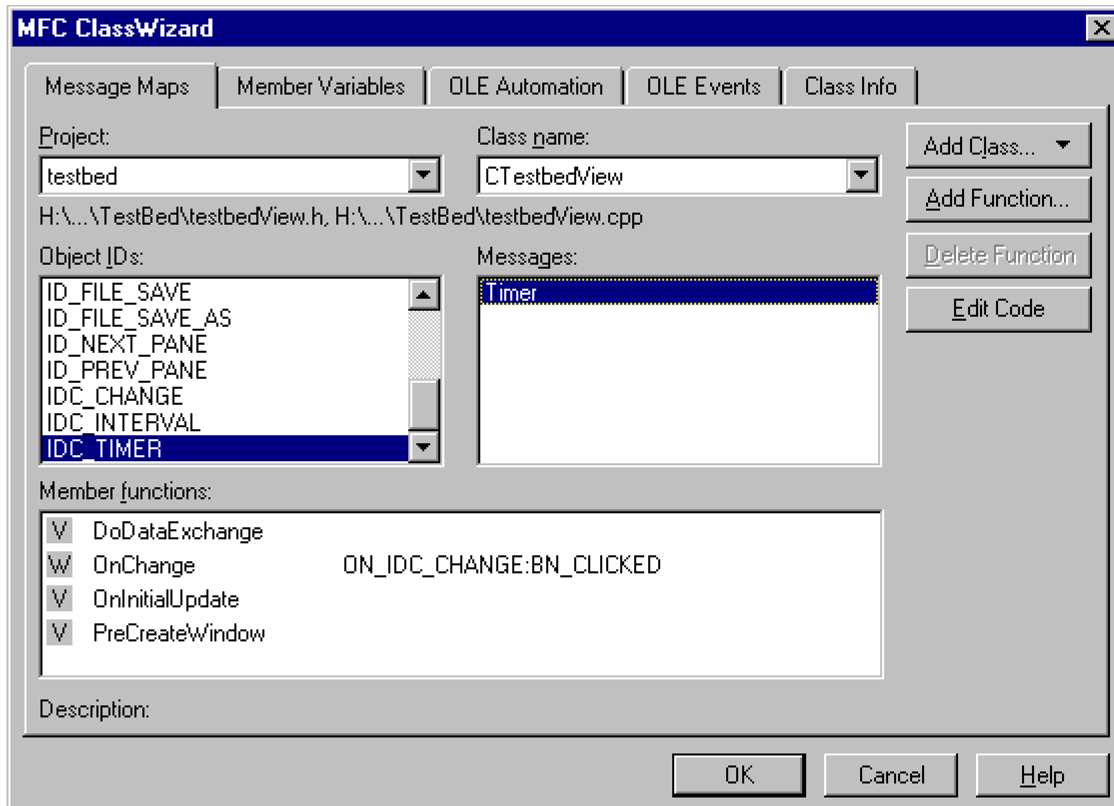
Adding an OLE control to a dialog box or a form view is a piece of cake, but if you try adding one to any other type of window, you'll have to implement a lot of code yourself, since ClassWizard won't give you much help.

First of all, you'll need to make sure that your control is registered in the system's registry. This will help the Component Gallery identify any OLE controls. Then you can use Component Gallery to insert a control into your project. ClassWizard will prompt you for the classes it wants to add to your project (this is very similar to adding wrapper classes for automation classes). The control's main class will be derived from `CWnd`, since this class now has support for implementing OLE controls in your projects.

With this done, it's simple to add the control to a dialog by drawing it on to the dialog with Developer Studio's dialog editor. Since I derived my view's class from `CFormView`, I found it easy to add the proper support for the OLE control. All I did was bring up the associated dialog for the form view and add the control to the screen.



Once I'd added the control to the dialog box, I used ClassWizard to add any needed variables and view the events for the control. Here you can see what ClassWizard showed me when I selected the control's identifier in the Message Maps tab:



I can then select and add member functions to support any of the events in the selected class. ClassWizard takes care of providing the map entry in the event map, and so on. This is too easy. But what if I want to place my control in a window that isn't based on a dialog template?

Using an OLE Control in Other Types of Window

Once you've gone through the steps of creating the necessary classes using Component Gallery, you can add an instance of the control to your windows by creating an instance of the control's main class (in my case, **`CTimer`**). If I wanted to do this, I would have a data member in my **`CWnd`**-derived class like this:

```
CTimer m_ctlTimer;
```

As with all **`CWnd`**-derived classes, creating a window (which is what the control is), requires that you create the C++ object and then the Windows object. You do this by calling the **`CWnd::Create()`** function like this:

```
m_ctlTimer.Create(NULL, WS_VISIBLE, CRect(10, 10, 100, 100), this,
   IDC_TIMER);
```

The first parameter is a title for the control (if any), the second is the window styles, the third is the position and size of the control, the fourth is the parent window of the control, and the last (as with any control) is the identifier.

This creates and displays the control, but there's still something that we must do: respond to events. How do we do that if we can't use ClassWizard? The answer is that we must do it by hand, or create a fake

dialog box to include the control and then copy and paste the entries from the dialog's class to the real window's class.

If you choose to roll it by hand, you'll have to add an event sink to the class and the member functions that will respond to the events. Adding the event sink map is really not that difficult. You simply add the **DECLARE_EVENTSINK_MAP** macro to the class' header file and use the **BEGIN_EVENTSINK_MAP** and **END_EVENTSINK_MAP** macros in the implementation file. In between, you fill in the event sink entries.

You perform this last step with the **ON_EVENT** macro:

```
ON_EVENT(theClass, id, dispid, pfnHandler, vtsParams)
```

The macro takes the following as parameters:

The class that contains the event sink
The control's identifier
The dispatch identifier of the event
The member function that should be called when the event occurs
The signature of the event handler specified using variant tag strings (as in the automation section)

Here's an example of how the macro is used:

```
ON_EVENT(CMyWnd, IDC_MYOLECONTROL, 1, OnMyEvent, VTS_BOOL);
```

As with automation, the best place to look for the information you need is in the associated type libraries, which should be supplied with the control. Again, you can use **Ole2vw32.exe** to locate and display the type information for the control, only this time, you'll only be searching for the objects with the OLE control icon next to the
items.

Summary

In this chapter, we've seen OLE Automation and OLE controls in action and seen how these technologies provide an alternative to the proliferation of custom interfaces. Hopefully, you've been convinced of the importance and versatility of OLE by now, but just in case you haven't, we'll drive the point home in the next chapter when we examine how OLE Automation can be coupled with ODBC to provide a powerful data warehouse.

OLE and ODBC

In this chapter we're going to examine how we can improve on MFC's ODBC classes to provide dynamic querying, as well as how we can use them to create a data warehouse. We'll be covering some pretty advanced topics, such as building component objects at run time, object-oriented architectures and metaclasses, so you'll need a thorough understanding of SQL, the ODBC classes and OLE Automation in order to get the most from this chapter.

If you need a bit of revision in the database area, you should be able to find enough basic information in the documentation supplied with Visual C++ to get you by, or you might like to refer to *Beginning Visual C++ 4* or *The Revolutionary Guide to MFC 4 Programming with Visual C++* (both from Wrox Press) for information on using MFC's database classes. For a complete run-through of SQL, check out *Instant SQL Programming* (also from Wrox Press). You can find information about COM and OLE in previous chapters of this book.

Dynamic Recordsets

Although MFC database classes have a lot of functionality, there are one or two areas that could use some improvement. My biggest quarrel has been **CRecordset**'s lack of dynamic information. What do I mean by *dynamic information*? Well, as you probably know, when you create a **CRecordset**-derived class using ClassWizard, you need to select the data source on which it will act. This adds member variables to the class that represent fields (or columns) in the database and sets up the record field exchange (RFX) macros for these members.

This is fine for a data source with a well-defined and unchanging structure, but let's say I have a database table that frequently has its schema changed, or I want to create an application that can browse any data source. Using the standard **CRecordset**-derived class, I'd have to provide static information about the table and its fields, along with variables to hold the data as I move from field to field. I'd even have to provide the name of the table. This just wouldn't be possible for an arbitrary or changing data source.

So, how do we make old, static MFC perform some new tricks for us? The answer is to extend the **CRecordset** class. Fortunately, since MFC exposes the ODBC handles and provides a number of **virtual** functions, we can make use of these to extend the class to do whatever bidding we see fit.

With a little work and the help of our newly extended recordset class, we could set up an application that allows the user to select both a data source and a SQL statement to execute against it at run time. We could then display the information requested by the user in the SQL statement, as well as asking the recordset how many columns came back and the name and type of each column. For example, our application's users could simply type:

```
SELECT * FROM Customers
```

They would get back information about all the rows and all the columns in the **Customers** table.

Providing a Dynamic Recordset

As it turns out, we had a strong need for such a dynamic recordset class in my office. One of my colleagues (thank you, Philip Jacobs) needed to call various stored procedures from his front-end Windows application. The stored procedures resided in a LAN-based Sybase SQL Server and returned

various pieces of information. Although the `CRecordset` class can be told to fire off stored procedures and return the procedures' results, the code for the stored procedures could change. In my office, this meant that the fields returned by the stored procedures did, in fact, change frequently. Using only the standard `CRecordset` would have meant that my colleague would often have to change his code to support the presence of new columns (or the removal of old ones) in the results of the stored procedures.

Clearly, we needed a new class, so mapping the requirements was pretty easy; we needed to build a class that is derived from `CRecordset` and that provides storage for cursors in the usual way (as `CRecordset` already does). In addition, the recordset should not have to be told what fields will be coming back. It should react dynamically and provide storage for the data, regardless of the number of columns and rows returned from the query. Furthermore, the query should be able to handle direct SQL statements, as well as calling stored procedures.

Keep in mind that MFC has strict syntax requirements for stored procedures. The SQL statement must be enclosed in curly braces and the `call` keyword must be used. For example:

```
"{call MyStoredProc 1010, '3/25/95', '3/25/96'}"
```

Implementation

My colleague has allowed me to reuse the C++ class he developed to fulfill these requirements in this chapter to show you exactly how to extend the ODBC classes. I'll explain what it took to build a class, called `CVarRecordset`, which is derived from `CRecordset`.

There are basically three functions that you would normally override for any MFC `CRecordset`-derived class: `GetDefaultConnect()`, `GetDefaultSQL()` and `DoFieldExchange()`.

```
virtual CString GetDefaultConnect();
virtual CString GetDefaultSQL();
virtual void DoFieldExchange(CFieldExchange* pFX);
```

The framework calls the `GetDefaultConnect()` function to get the default connect string for the data source on which the recordset is based. `GetDefaultSQL()` is called to get the default SQL statement on which the recordset is based. This might be a table name or a SQL `SELECT` statement.

`DoFieldExchange()` is called by the framework to automatically exchange data between the field data members of your Recordset Object and the corresponding columns of the current record on the data source. It also binds your parameter data members (if there are any), to parameter placeholders in the SQL statement string for the recordset's selection. The exchange of field data, called **record field exchange** (RFX), works in both directions; from the Recordset Object's field data members to the fields of the record on the data source, and from the record on the data source to the Recordset Object.

The only action you must normally take to implement `DoFieldExchange()` for your derived recordset class is to create the class with ClassWizard and specify the names and data types of the field data members. When you declare your derived recordset class with ClassWizard, the wizard writes an override of `DoFieldExchange()` for you, which resembles the following:

```
void CCustomerSet::DoFieldExchange(CFieldExchange* pFX)
{
   //{{AFX_FIELD_MAP(CCustomerSet)
   pFX->SetFieldType(CFieldExchange::outputColumn);
   RFX_Text(pFX, "Name", m_strName);
   RFX_Int(pFX, "Zipcode", m_nZipcode);
```

```
    //}}AFX_FIELD_MAP
}
```

DoFieldExchange() is very similar to **DoDataExchange()** for dialogs. So similar, in fact, that it also has a set of exchange functions to deal with moving data from the cursor to data members within the recordset class. However, the recordset class will normally only hold one record at a time. This means that you must call one of the movement functions to interact with data of other records.

Whenever any of the movement functions are called, they usually result in a call to a **virtual** function named **Move()**.

```
virtual void Move(long lRows);
```

The **lRows** parameter is used internally by the function to determine how many records to move forward or backward. A negative number tells the function to move backward. When you call **MoveNext()** for example, it, in turn, calls **Move()**, passing it **1** for the **lRows** parameter. As you'll see, we'll also need to override this function to provide the features that we need.

The **CVarRecordset** class is implemented so that you can either derive your own classes from it (for example, if you want to bind parameters) or create **CVarRecordset** objects directly (without having to derive a class) from it.

Since **CVarRecordset** is going to need to store and supply all the necessary information about the recordset, it features a number of data members and access functions. The data members are shown below, along with a description (each has a corresponding access function that we won't show here). Note the use of the array classes. These are particularly useful as they're easily resizable, which is an important feature, since we don't know in advance how much data they'll need to hold.

```
private:
   SWORD m_nCols;                     // Number of columns in the result set
   CWordArray m_wTypeArray;           // Type of each column
   CWordArray m_wScaleArray;          // Scale of each column
   CWordArray m_wNullableArray;       // Nullable flag of each column
   CDWordArray m_dwPrecArray;         // Precision of each column
   CStringArray m_strColNameArray;    // Name of each column
   CStringArray m_strTypeNameArray;   // Type name of each column
   CStringArray m_strResultArray;     // Current row's results
```

The class automatically determines the number of result columns and their attributes (names, types, and so on) in an override of another **virtual** function, called **PreBindFields()** (this one has no implementation in the **CRecordset** class).

PreBindFields() is called only once when you invoke your SQL command. If you derive your own class from **CVarRecordset** and you need to override **PreBindFields()**, you must call the base class function (**CVarRecordSet::PreBindFields()**) from within your override to retain **CVarRecordset**'s functionality.

To make this all work, you must tell MFC that you'll only move in one direction as you traverse through the records. You do this by calling the **CVarRecordset::Open()** function with the first parameter (**nOpenType**) equal to **CRecordset::forwardOnly**.

CVarRecordset supports all the SQL statements that the **CRecordset** class does, except for table names. For example, if you had a **CRecordset** with a SQL statement of **TableName** you would need to change it to **SELECT * FROM TableName** to use a **CVarRecordset**. This is because the MFC code requires that at least one result field be bound if plain table names are used. Table names are normally returned from the

CRecordset::GetDefaultSQL() function. MFC will attempt to call **DoFieldExchange()** and expect to find some fields that it can bind to the SQL statement, which is not what you want for a dynamic recordset.

A Close Look at the Code

Let's start by examining the code for the three functions **GetDefaultConnect()**, **GetDefaultSQL()** and **DoFieldExchange()**:

```
CString CVarRecordset::GetDefaultConnect()
{
   return "";
}

CString CVarRecordset::GetDefaultSQL()
{
   return "";
}

void CVarRecordset::DoFieldExchange(CFieldExchange* pFX)
{
   //{{AFX_FIELD_MAP(CVarRecordset)
   //}}AFX_FIELD_MAP
   pFX->m_bFieldFound = TRUE;   // kludge
}
```

Quite simple, isn't it? The code didn't have to be very extravagant for these three functions. It just had to make MFC think that everything is as usual. That is the reason why we set the **m_bFieldFound** member of the **CFieldExchange()** class to **TRUE**. It makes MFC think that all of the fields have been bound as usual, when in reality there are no bound fields.

The next function we'll examine is **CVarRecordset::PreBindFields()**:

```
void CVarRecordset::PreBindFields()
{
   RETCODE nRetCode;                   // SQL function return code
   SWORD i;                            // loop counter
   char szColName[MAX_COLNAME+1];      // col name
   char szTypeName[MAX_COLNAME+1];     // col type name
   SWORD cbColName;                    // num of bytes in szColName
   SWORD fSQLType;                     // SQL data type
   UDWORD cbPrec;                      // column precision
   SWORD cbTypeName;                   // num of bytes in szTypeName
   SWORD cbScale;                      // column scale
   SWORD fNullable;                    // column nullable flag

   // Determine the number of columns
   AFX_SQL_ASYNC(this, ::SQLNumResultCols(m_hstmt, &m_nCols));
   if (!Check(nRetCode))
      ThrowDBException(nRetCode);

   // Size the column descriptor arrays
   m_wTypeArray.SetSize(m_nCols);
   m_wScaleArray.SetSize(m_nCols);
   m_wNullableArray.SetSize(m_nCols);
   m_dwPrecArray.SetSize(m_nCols);
   m_strColNameArray.SetSize(m_nCols);
   m_strTypeNameArray.SetSize(m_nCols);
   m_strResultArray.SetSize(m_nCols);

   // Get the column descriptor information for each column.
   // Note that in CVarRecordset references to columns are
   // zero-based; however, in ODBC they are one-based (column
   // zero has a special meaning in ODBC).
```

```
      for (i = 0; i < m_nCols; i++)
      {
         AFX_SQL_ASYNC(this, ::SQLDescribeCol(m_hstmt, i+1,
            (UCHAR far *)szColName, MAX_COLNAME, &cbColName, &fSQLType,
             &cbPrec, &cbScale, &fNullable));
         if (!Check(nRetCode))
            ThrowDBException(nRetCode);

         AFX_SQL_ASYNC(this, ::SQLColAttributes(m_hstmt, i+1,
            SQL_COLUMN_TYPE_NAME, (UCHAR far *)szTypeName,
            sizeof(szTypeName), &cbTypeName, 0));
         if (!Check(nRetCode))
            ThrowDBException(nRetCode);

         m_wTypeArray[i] = fSQLType;
         m_wScaleArray[i] = cbScale;
         m_wNullableArray[i] = fNullable;
         m_dwPrecArray[i] = cbPrec;
         m_strColNameArray[i] = szColName;
         m_strTypeNameArray[i] = szTypeName;
      }
   }
```

This function is called by MFC's implementation of the **`CRecordset::Open()`** function immediately after executing the query, but before fetching the data. This makes it the perfect place to determine the number of result columns and their attributes.

The code here is very simple. The function starts by calling **`SQLNumResultCols()`** to determine the number of columns returned. Next, it initializes the arrays which will hold the information returned from the query.

Once the initialization has been performed, we're ready to begin reading in the data returned from the query. The first thing we must do is to find out more about the columns returned. We want to know the name of the column, the precision and scale of the column, and whether or not the column is nullable (as used by databases, not C++). This information is easy to get at with the **`SQLDescribeCol()`** function, which returns the information just described.

The **`SQLColAttributes()`** function allows us to ask the column for its type so that we know how to treat the data if we need to do anything with it. The value will come back as one of several possible ODBC data type values. All of this information is then simply stored away for later use.

Notice that I used the **`AFX_SQL_ASYNC`** macro which makes sure that no more than one call goes out at a time to the ODBC API. In other words, since ODBC can work asynchronously, you could potentially make a call to ODBC which will return immediately before the requested data has come back from the server. If the application made another asynchronous call while still waiting for the data, this could lead to problems. The **`AFX_SQL_ASYNC`** macro determines whether another call is still out there, and if there is, throws an appropriate exception.

You can find more information on the **`AFX_SQL_ASYNC`** macro and the **`SQL`***`xxx`***`()`** functions in the documentation supplied with VC++.

The last function we need to look at is the implementation of the **`Move()`** function. As I said before, this function is called either directly or via one of the other movement functions (such as **`MoveNext()`** or **`MovePrev()`**) to fetch the data for a different record and place it in the member variables provided by the recordset. However, since we don't know ahead of time what type of data or the number of columns we're getting, we need to provide our own implementation for the **`Move()`** function. Here is the code implemented for the **`CVarRecordset::Move()`** function:

```
void CVarRecordset::Move(long lRows)
{
   RETCODE nRetCode;            // SQL function return code
   SWORD i, j;                  // loop counters
   char szData[MAX_COLNAME+1];  // Data buffer
   SDWORD cbData;               // Num of bytes returned in data buffer

   // Row positioning.
   // Only call the CRecordset::Move function if result
   // columns are defined. Will GPF if m_nCols == 0.
   if (m_nCols > 0)
      CRecordset::Move(lRows);
   else   // clean up - code is from MFC (DBCORE.CPP)
   {
      ReleaseCopyBuffer();
      m_bEOFSeen = m_bBOF = m_bEOF = TRUE;
      m_bDeleted = FALSE;
   }

   // Get the ASCII result value for each column.
   for (i = 0; i < m_nCols && !IsEOF(); i++)
   {
      AFX_SQL_ASYNC(this, ::SQLGetData(m_hstmt, i+1, SQL_C_CHAR,
         (UCHAR far *)szData, MAX_COLNAME, &cbData));
      if (!Check(nRetCode))
         ThrowDBException(nRetCode);

      // remove trailing blanks
      for(j = lstrlen(szData) - 1; j >= 0 && szData[j] == ' '; j--);

      szData[j+1] = '\0';

      m_strResultArray[i] = szData;
   }
}
```

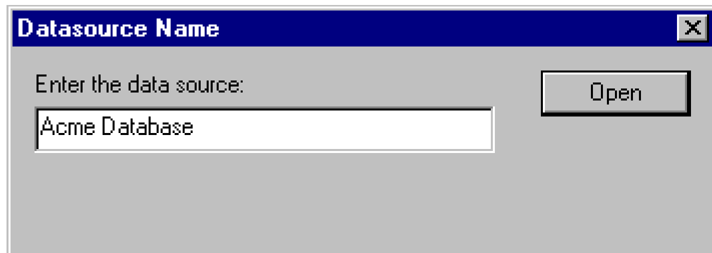
The important call we're making here is the call to **SQLGetData()**, which is where the data for the row comes from. We call this function once for each column so that we can fetch an entire row of data. Again, since we use the MFC macro **AFX_SQL_ASYNC**, it's possible that an exception will be thrown if this function is called at an inappropriate time (like when another asynchronous **SQL***xxx***()** function is in progress).

Using the Dynamic Recordset Class

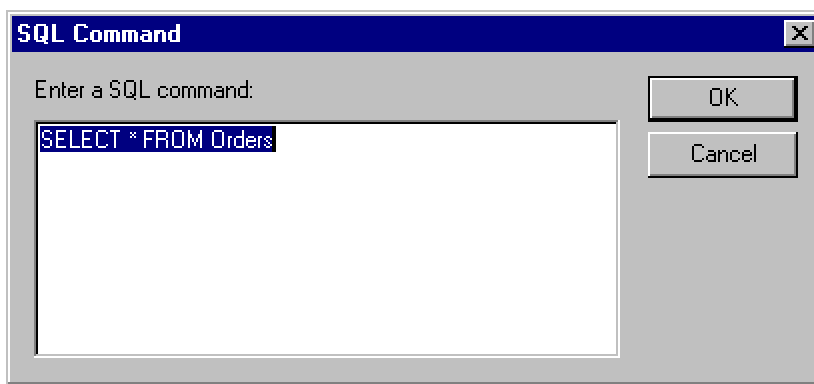
To try this stuff out,I wrote an MFC-based application which you can find on the CD as `Dynrcset` (for dynamic recordset application). I used MFC AppWizard (exe) to generate an SDI application with the only database option being Header files only. I also told AppWizard to derive the view class from **CScrollView** .

To keep things as simple as possible, I kept all of the code changes I needed to make in the view class. I normally wouldn't code like this, but I'll allow it for the sake of simplicity (if it means that you'll walk away from this chapter with some new found knowledge).

You use the application by first entering the name of the data source you want to connect with.



Once you've entered a valid data source name, the application calls the `CDatabase::Open()` function on the `m_Database` data member defined in the view class. If the connection is successful, you can enter a SQL command by selecting the Edit/Enter SQL Command... menu option or the corresponding toolbar button. This option displays the dialog box shown:



Once you've entered a new SQL command, you click the OK button and off it goes. The associated recordset is opened with the SQL command as its parameter, and the result set is queried dynamically, as shown in the code below:

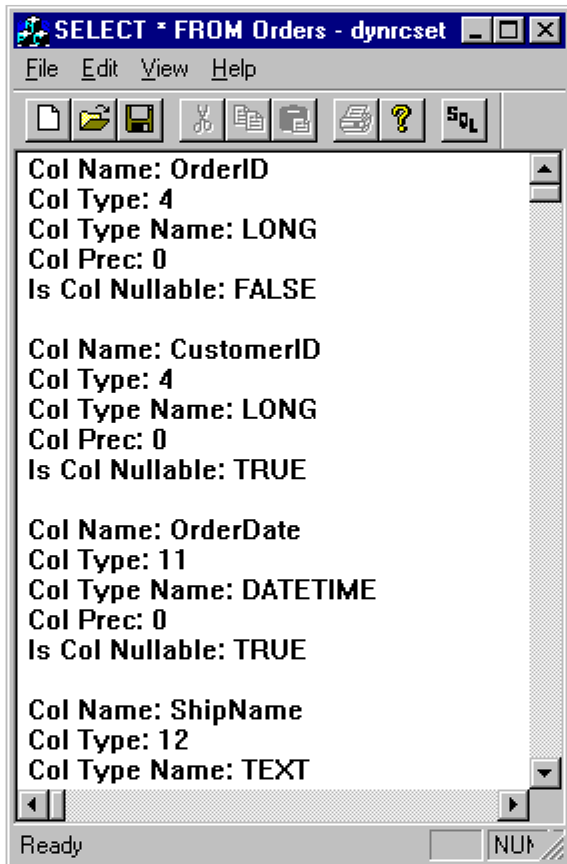
```
void CDynRecordsetView::OnEditEnterSQLCommand()
{
   CSqlCmdDlg dlg;
   dlg.m_strSqlCmd = GetDocument()->GetTitle();

   if (dlg.DoModal() == IDOK)
   {
      CVarRecordset rs(&m_Database);

      rs.Open(rs.forwardOnly, dlg.m_strSqlCmd, rs.readOnly);

      int nCols = rs.GetNumCols();
      m_Lines.RemoveAll();
      CString strLine;

      // Fetch the column information.
      for (int i = 0; i < nCols; i++)
      {
         strLine.Format(_T("Col Name: %s"), rs.GetColName(i));
         m_Lines.Add(strLine);
         strLine.Format(_T("Col Type: %d"), rs.GetColType(i));
         m_Lines.Add(strLine);
         strLine.Format(_T("Col Type Name: %s"), rs.GetColTypeName(i));
         m_Lines.Add(strLine);
         strLine.Format(_T("Col Prec: %d"), rs.GetColScale(i));
         m_Lines.Add(strLine);
         strLine.Format(_T("Is Col Nullable: %s"),
```

```
            rs.IsColNullable(i) ? _T("TRUE") : _T("FALSE"));
         m_Lines.Add(strLine);
         m_Lines.Add(_T(""));
      }

      // Fetch the result set information.
      for (; !rs.IsEOF(); rs.MoveNext())
      {
         strLine = _T("");
         for (i = 0; i < nCols; i++)
         {
            strLine += rs.GetColResult(i);
            strLine += _T("\t");
         }
         m_Lines.Add(strLine);
      }
      rs.Close();

      GetDocument()->SetTitle(dlg.m_strSqlCmd);
      GetDocument()->UpdateAllViews(NULL);
   }
}
```

As you can see, this function prepares an array of strings based on the information that is returned from the result set. The views are then invalidated and the strings are painted in the client area via the `OnDraw()` function:

```
void CDynRecordsetView::OnDraw(CDC* pDC)
{
   CDynRecordsetDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);

   int nTab = 125;
   int nCount = m_Lines.GetSize();
   int nHeight = 15;
   for (int i = 0; i < nCount; i++)
   {
      pDC->TabbedTextOut(5, nHeight * i, m_Lines.GetAt(i), 1, &nTab, 0);
   }
}
```

You can see the result below. The text on the screen contains descriptions of the columns returned, followed by the actual data for all of the rows.



That's basically all there is to it; nothing more and nothing less. Just a couple of lines of code and you can dynamically fetch data using the MFC ODBC classes. Pretty cool! However, you might feel that this just isn't enough power for you. If that's the case then maybe the next section will please you, because you're going to meet my all time favorite code for creating a totally awesome data warehouse.

A Data Warehouse

I'm sure you appreciate how much of an improvement ODBC is over the previous situation when each database vendor offered their own API and dialect of SQL. As a standard supported by most vendors, ODBC succeeded in reducing the training and support costs associated with database application development. Any problems associated with using the C-based ODBC API have been largely overcome by MFC's classes, which offer an object-oriented view of ODBC that is much easier to use than raw functions and handles.

However, as I mentioned earlier in this chapter, these classes still have several limitations. They don't allow dynamic information to be returned easily, SQL joins are not updatable, and there's no way of binding fields from the recordsets to controls on your application's screens (without using an intermediary class, such as `CRecordView`, and even this is limited to the types of control it can exchange data with).

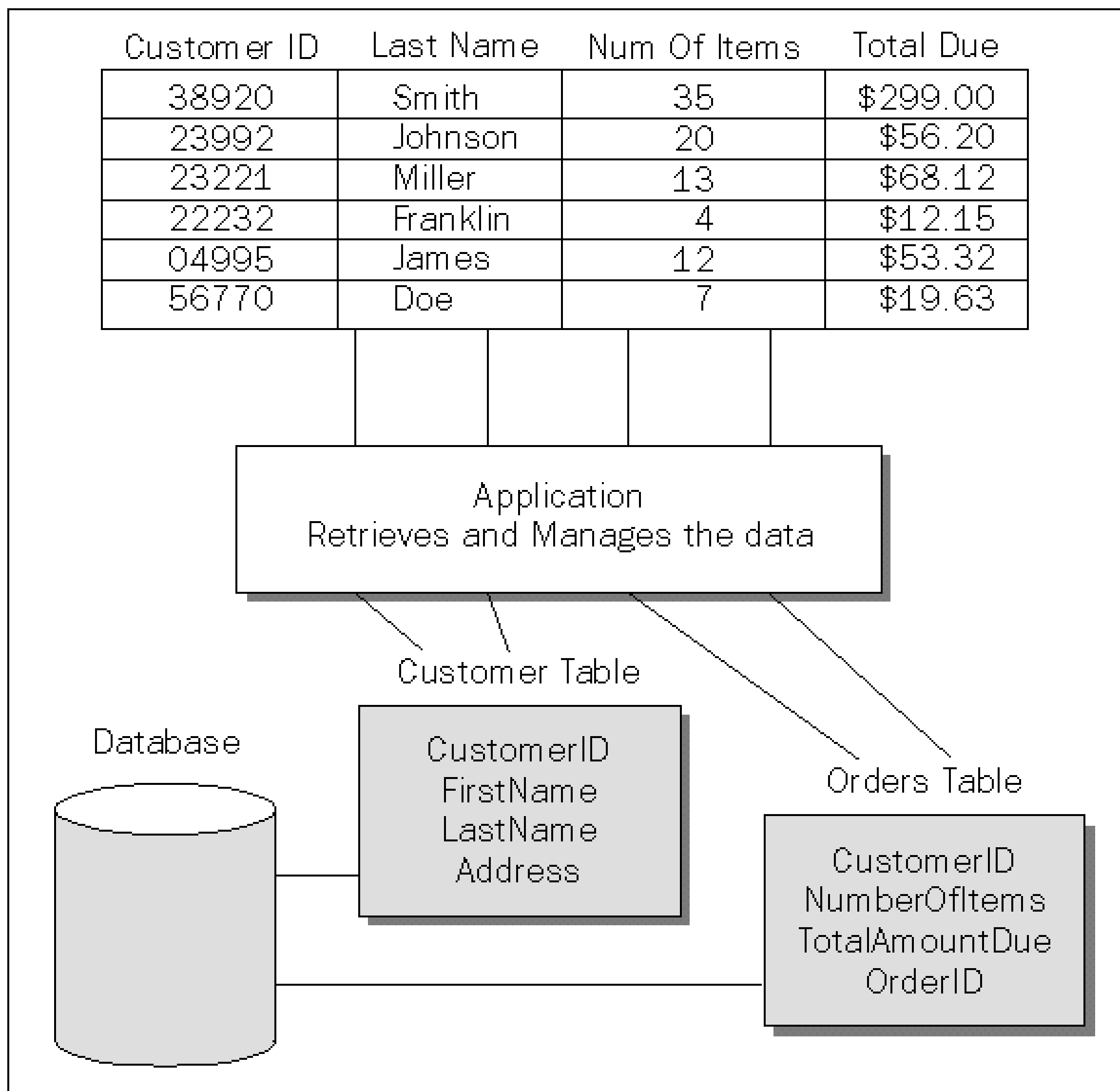
The section on dynamic recordsets solves the first of these problems, but it doesn't solve any of the others. If your intention is to simply read dynamic information from a database as read-only data and provide your users with a generic solution for viewing the data then dynamic recordsets are the way to go. However, if you want a much more powerful database engine that can provide an object-oriented database layer above any database that supports ODBC and can adapt itself (without any further coding from you) to reflect new database structural changes, read on.

The Problem of Loose Change

For years, developers have had to struggle with the fact that whenever a database changes on the back-end, they have to change the front-end application. Let me explain this with an example. Let's say that you've been hired to create an application for the Acme Home Shopping Network. Your manager says that the company wants to move into the 90s, so they want to track their customers, the orders they place and the inventory, using a client-server architecture.

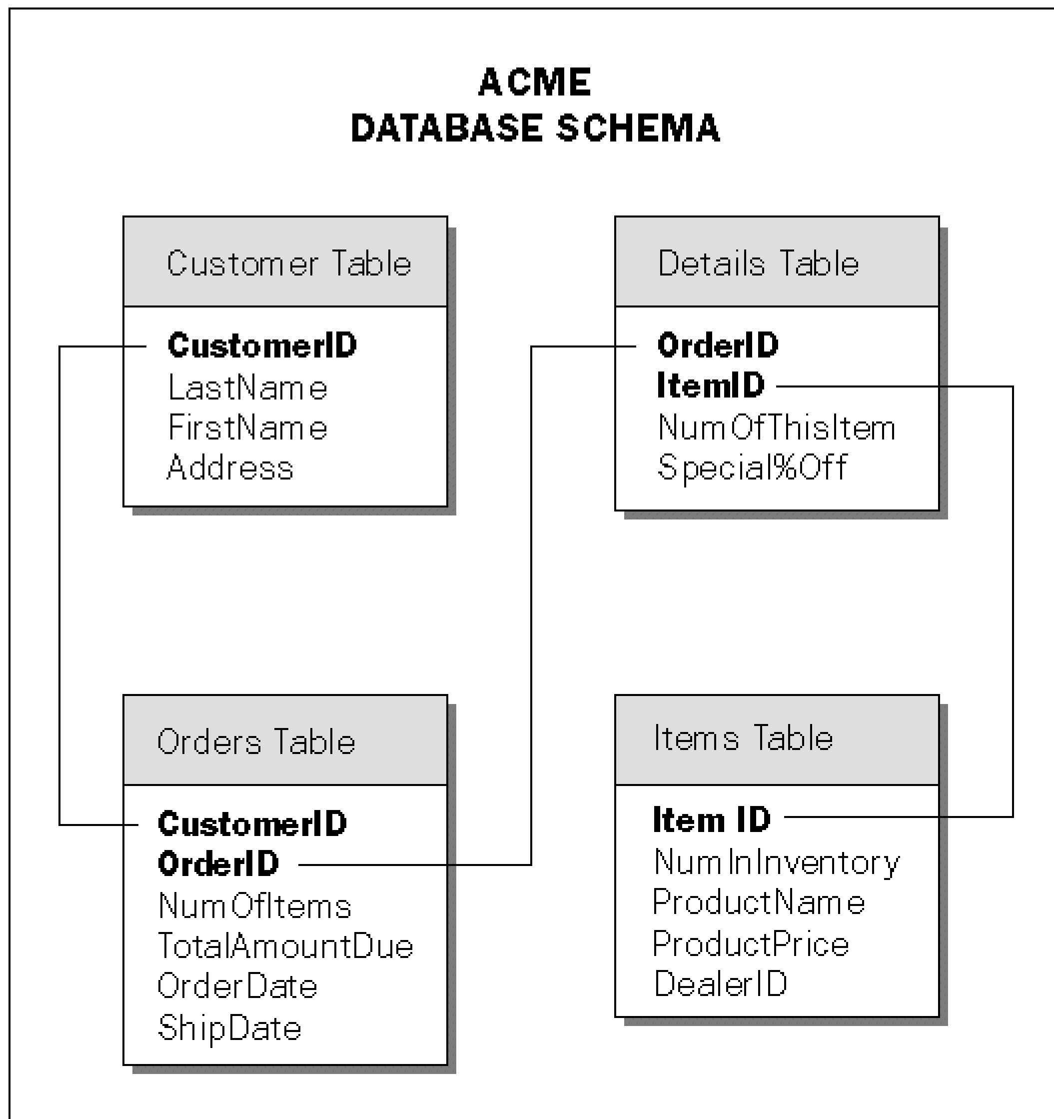
The architecture should be based on a back-end database sitting on a LAN-based server with a Win32 front-end application providing the user interface, which the Customer Service and Inventory departments will use to enter orders and ship products. While you're performing your design and analysis, you decide that you'll need a database with several tables. The first table will be a Customer table which will contain information about the customer, such as their home address, their shipping address, phone number, credit card information and several other pieces of data.

You also realize that you'll need to keep track of one very important field: the customer ID. This field can then be used to look up other pieces of data in other tables.



Once you've studied the situation further, you come to the conclusion that the database will need somewhere to place the orders, so you create an Orders table, which will be joined to the Customer table in order to list the orders for a particular customer. You could place all of the details of the Orders in this table, but it would probably make more sense to provide an additional table to hold the details of the orders. That way, you avoid having to repeat some of the general order information. This Details table can then be joined to the Orders table based on an OrderID field.

The final table that you'll need is one that contains the inventory data. This table will be updated as new inventory arrives or as orders are logged. The Details table will contain an ItemID field that is also present in the Inventory table. The figure shows how everything fits together:



Now that you have the structure of the database created, you need to start writing the front-end application. The application needs to provide the Customer Service department with a way of creating new customer records and entering orders for the customers. It should also provide a few screens for the Inventory department to enter new inventory when it arrives.

Once the application has been developed and tested, you hand it over to the users. Time goes on and the users start requesting other features or changes. In the meantime, you become a slave to this one application because as more features or changes are made to the back-end database, you reflect those changes in your front-end application with code hard-wired to the database.

You soon realize that you're working doubly hard to provide fixes or changes to the application and the back-end. Then you start to wonder if there's a better solution. Currently, the application has all the code for requesting or updating data to the database tables explicitly bound to the back-end. Furthermore, the recordsets created by your code that span across tables with SQL joins are overly complicated. Given that it's not easy to deal with these concepts from any type of application, it's pretty obvious that you're

stepping in to a potential maintenance nightmare.

Furthermore, let's say that some of the fields need to be calculated based on the values entered in other fields. For example, you might want to calculate an order's total amount based on the state tax and the subtotal of the order. This means that you would have to wait until the state is entered into the State field of the Customer record. You need to keep track of the details of the order and each time a new detail is added, you simply update the total. This logic is normally built into the front-end application, but if someone decides that other calculations must be added or changed, you have to update the application, compile and test the code and, finally, deploy the application once again.

My First Attempt

I've been placed in this very situation time after time, until one day I got smart and decided to create a generic solution, one that could virtually maintain itself without much intervention from me. That way, things could continue to run smoothly and I could go off and teach seminars, courses, or write books.

In my first attempt to provide a solution to the maintenance problem, I created a custom resource in my application's resource file, using the **RCDATA** resource type. **RCDATA** allows you to create any structure imaginable and read the data contained in the **RCDATA** section however you choose to from your application. The data can be delimited by commas. My **RCDATA** type contained a header section and a fields section. The structure looked something like this:

```
CUSTOMER DATADEF
BEGIN
4, "Customer\0", "CustomerID = ?\0",
   "LastName DESC\0"
"Customer ID\0", "\0", "\0", 0, 0, 0, "CustomerID\0",
   SQL_INTEGER, DDF_PRIKEY
"Last Name\0", "\0", "\0", 0, 0, 0, "LastName\0",
   SQL_CHAR, DDF_DEFAULT
"First Name\0", "\0", "\0", 0, 0, 0, "FirstName\0",
   SQL_CHAR, DDF_DEFAULT
"Ending Bal\0", "0.00\0", "10000.00\0", 2, 1, 1,
   "EndingBal\0", SQL_FLOAT, DDF_DEFAULT
END
```

The first line **CUSTOMER DATADEF** identifies the structure. **CUSTOMER** is the name of the resource and **DATADEF** is the resource type. I called it **DATADEF** because I was trying to create a data definition object that could be used to identify the location of data coming from the data source.

I used the **FindResource()** Win32 API to locate and return a handle to the specified resource. The first parameter to this function is the instance handle of the application. The second identifies the resource, in this case, **"Customer"**. The last parameter identifies the type of resource. Normally, this value is set to one of the predefined constants, such as **RT_BITMAP**, or **RT_ICON**, but you're allowed to create your own types and simply pass the string as this identifier, in my case, **"DATADEF"**. Once the resource has been located, I then call **LoadResource()**, passing it the handle returned from **FindResource()**.

```
HRSRC hrsrc = ::FindResource(hInst, _T("Customer"), _T("DATADEF"));
HGLOBAL hRes = ::LoadResource(hInst, hrsrc);
if (hRes != NULL)
   UINT FAR* lpnRes = (UINT FAR*)::LockResource(hRes);
```

Next, I begin to traverse the lines between the **BEGIN** and **END** block. The first line is always the header, which contains the following items in this order:

```
FIELD COUNT, TABLE NAME, FILTER STRING, SORT STRING
```

The field count determines how many field detail lines will follow the header. The table name is the actual name of the table that the fields will come from. The filter string contains the value that will be used in the **WHERE** clause of the final **SELECT** SQL statement. In reality, I used this string in the MFC's **CRecordset::m_strFilter** string when constructing the query and I used the sort string to fill the value of **CRecordset::m_strSort**. I read in this information using a function that implemented code much like the following:

```
   m_nNumOfCols = *lpnRes++;
   ExtractString(lpnRes, m_strTblName);
   ExtractString(lpnRes, m_strFilter);
   ExtractString(lpnRes, m_strSort);
```

The **ExtractString()** function simply traverses through the characters that follow, until it finds a null terminator:

```
void ExtractString(UINT FAR*& lpnRes, CString& strText)
{
   char szBuffer[1000];
   char* pszBuffer = szBuffer;
   // Retrieve the string.
   while(*((char*&)lpnRes) != NULL)
   {
      *pszBuffer = *((char*&)lpnRes)++;
      pszBuffer++;
   }
   *pszBuffer = *((char*&)lpnRes)++;
   strText = szBuffer;
}
```

The detail lines include information about the actual columns that I want to retrieve from the specified table, as well as user interface details needed for displaying the data:

```
UI-TEXT, MIN VALUE, MAX VALUE, DECIMAL PLACES, COMMAS, DOLLAR SIGN,
   ODBC-TYPE, KEY FLAG
```

The user interface text is used to describe the data on the screen. For example, if the column that came back from the data source contains a value such as **"RAMIREZ"**, the user interface text might contain a value like **"LAST NAME:"**, which can be used on a dialog box or window. The window might contain a static control for the user interface text and an edit control to hold the value.

The minimum and maximum values are used when numeric data is being returned from the database. The dialog box or window code will most likely need to know how to limit and validate the user's input. These values determine what range the user can enter for the value that pertains to the current field.

The decimal places, commas and dollar sign values are used by the user interface to determine how to format the actual value. For example, the value might need to be displayed as $6628.161 or 6628.2. The ODBC type is used to identify the type of the data coming from the database. The possible values are constants from the `Sql.h` file (shipped with Visual C++) and include **SQL_CHAR**, **SQL_FLOAT**, and **SQL_INTEGER** (among several others).

The last value in the detail line is the key flag. This is used to optimize database updates by marking the field with one of a set of values that provide information about whether the filed is used as a key:

```
#define DDF_DEFAULT   0x0000
#define DDF_PRIKEY    0x0001
```

```
#define DDF_SECKEY   0x0002
```

DDF_PRIKEY is used for the primary key in the table and **DDF_SECKEY** is used for any secondary keys. You should always include all types of key field in the data definition because an **UPDATE** statement won't be able to update the correct row in the database without them. **DDF_DEFAULT** is used for any fields that aren't used as keys.

Once all of the information has been read from the resource, I use it to construct an MFC recordset. The recordset class uses the information contained in the resource to populate itself with the correct data. The constructor simply fills the **m_nFields**, **m_nParams**, **m_strFilter**, and the **m_strSort** data members based on the information from the header line. The **GetDefaultSQL()** function normally returns the name of the table from which the data should be extracted and this can also be constructed from the header line. The **DoFieldExchange()** function calls the appropriate record exchange function (one of the **RFX_***xxx***()** functions) with the appropriate information, which it can gather from the detail lines.

As you can see, this solution offers a very flexible system, since the data definition objects drive the application. All that the application has to do is construct a data definition object from the appropriate resource object stored in the resource file and have it load its data using the generated recordset.

However, there are still a few disadvantages with this method. The major one is that you still have to recompile and link the resource to the executable and, finally, redeploy the application each time a change is made to the resource file. This quickly became an ongoing headache for me.

It would be great if I could find a way to remove the data definition from the application completely. Since the data definition is going to be closely tied to the actual data, why not take the definitions and add them to the database? That way, each time that an object's definition changed, I wouldn't have to change the application at all. I could simply change the definition information on the back-end database and the application would automatically pick up the new definition the next time it created an object of the changed type. This became the essence of my Data Warehousing Model.

The Component Layer

After going back to the drawing board, I came up with a new design which I originally referred to as a *business object model*. As the term *Data Warehousing* became more popular and more widely understood, it became obvious that my design was based on the same theories, so I now call my code a *data warehouse*.

The code for the warehouse is split into two layers that I call the **component layer** and the **transient layer**. Each layer is represented by an OLE in-process server. The transient layer consists of objects that offer methods for accessing and manipulating data in a database and the component layer provides a higher level model that insulates applications from changes made to the database. In this section, we'll examine some of the theory behind the component layer.

Data Objects

With this model, I was trying to achieve an object-oriented layer on top of a relational database, giving me the ability to create persistent objects that know how to load and save themselves from a data source. This supports the object-oriented notion of encapsulation of data. Since these objects would represent data, I called them **Data Objects**.

Class Objects

I also wanted to encapsulate the definition of these objects, as well as their data. Where C++ uses classes to represent the definition of an object, I defined a **Class Object** to achieve the same thing. The Class Object's values (which represent the definition of a set of Data Objects) are stored in the back-end database (just like the data definition values in the previous example were stored in the resource file).

The Class Objects contain all the information needed to create a Data Object, including how the Data Object's data can be extracted from the database and any information that applications may need to display that data correctly. An application will never tell the Data Object where to get its data from, since this information is retrieved at run time and encapsulated by the Class Object. This removes the dependency that an application has on the location of data in the database.

Many applications will want to be able to get the appropriate Class Object from a Data Object, so I provided a function in the Data Object to make this possible.

Inheritance

To expand on these object-oriented concepts and to parallel C++ more closely, I wanted to implement something similar to class inheritance. The idea of defining a class with fields and then deriving a class from the first class, having it inherit all of the fields from that class, really intrigued me.

Basically, all that has to happen is that a class must first determine whether it is inherited from some other class and, if it is, it needs to load the fields of that class as well. If the parent class also has a parent, those fields should also be loaded. I didn't care about supporting multiple inheritance, since I never use the feature in C++ anyway, but you could add this feature yourself if you wish.

Container Objects

I introduced a new sort of object into my model, called a **Container Object**, to be responsible for creating Data Objects and Class Objects, maintaining lists of the active objects and providing an easy way of destroying all the active objects at once. This makes a client application much simpler, since it doesn't have to concern itself too much with the details of object creation and destruction.

A warehouse client application will use a Container Object to create a Data Object just like an OLE client uses a class factory to create an instance of an OLE object. Both the Container Object and the class factory offer a function that accepts a class identifier and returns the type of object requested by the client. Internally, the Container Object will create an appropriate Class Object (if one doesn't already exist) and use this to create a Data Object to return to the client. The client can then use either of these objects as required.

To ensure that the Container Object can return an existing object if the client application decides to create the same Data Object from two different parts of an application, I also require the client to pass the Container Object a key when it creates one. This key is combined with the class identifier to uniquely identify a Data Object. If a client tries to create a Data Object with both the class and the key the same as an existing object, the container will return that object, otherwise it will create a new one.

This concept can be further expanded to go across process boundaries and provide objects that can be shared by different applications, but it requires a bit of magic because of the limitations on sharing data between processes in Win32.

The Transient Layer

Now that we've met the component layer and we understand its concepts, there's only one thing missing. How do we actually retrieve the values of the Class Object and the data of the Data Object from the data source?

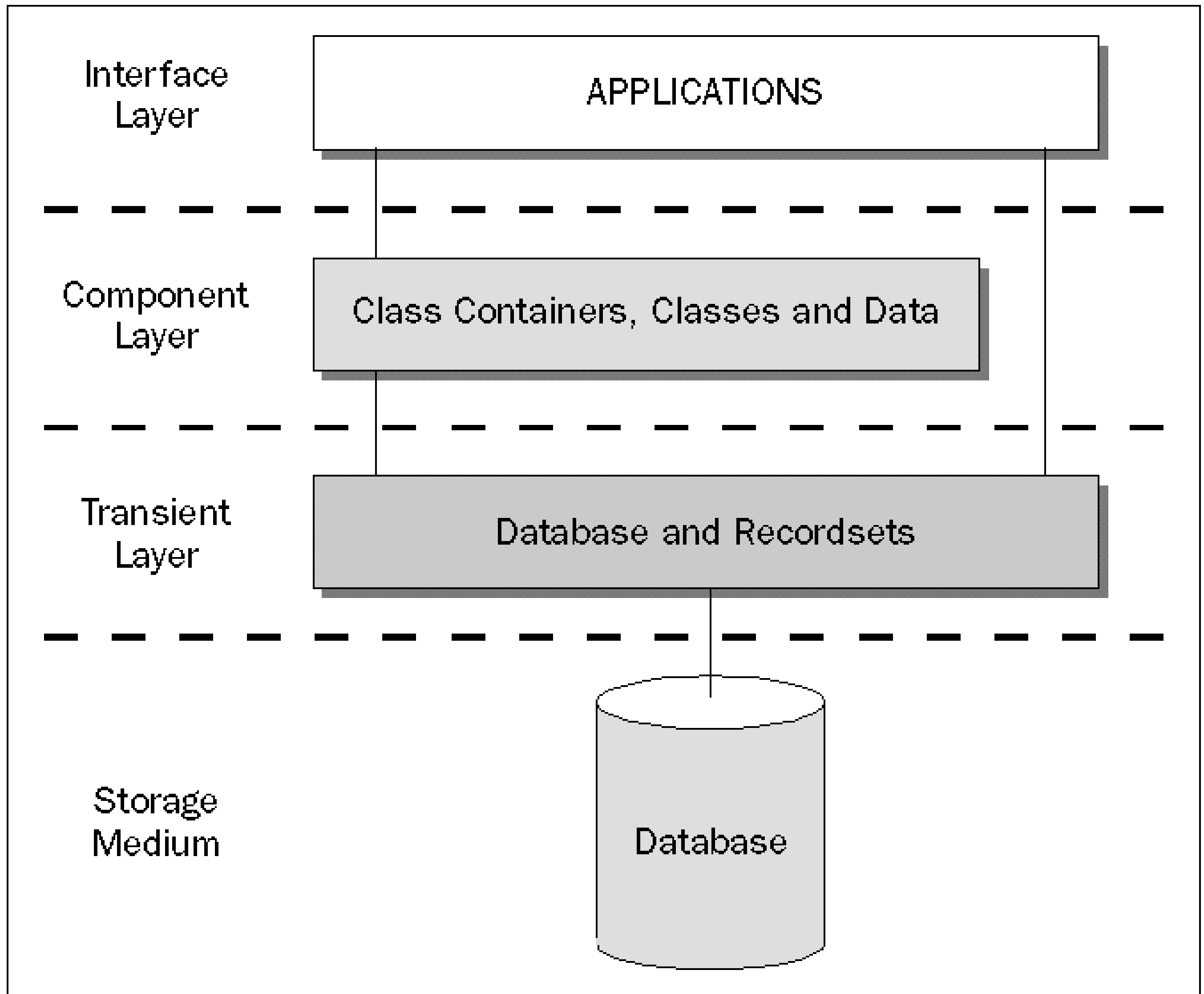
In my design, I wanted to abstract the method of accessing the data source and the medium being used as much as possible. I didn't want the Container, Class or Data Objects to know whether they were using a database, a memory mapped file or even a disk file, so I needed a lower-level layer that would deal with the implementation of the data source directly. This layer would be the only one to know how to communicate with the underlying data source. The component layer relies on the transient layer to provide the data (regardless of where it resides). As long as the interface of the transient layer remains constant, its implementation can change without affecting the component layer or any client code.

I reused the concept of a Database and Recordset Object for my transient layer to provide an abstraction of the data source. The Database Object is responsible for opening a connection to the data source. This could mean connecting to a database, opening a handle to a file, or allocating memory using pointers. It's also responsible for closing the connection. The Database Object can create Recordset Objects on demand and return them to any callers. Although the Database and Recordset Objects don't *have* to use databases in theory, my implementation does, so the rest of this chapter will assume that.

The Recordset Object's responsibilities include loading data from the associated database, updating changes back to the database, deleting records from the database and adding new records to the database.

Before the Recordset Object can perform a simple task, such as loading data from a table, it needs to be told where to go (the table and column that contain the data of interest). Therefore, the Recordset Object needs to be a dynamic object that can be changed on demand. It should have the ability to be told what columns from what tables to go after, and it should return the data based on parameters we specify. We should also have the ability to specify the sort order and any joins to other tables if necessary. The recordset has to internally generate any SQL it might need to complete its job.

Here you can see the different layers of the data warehouse:



Note that applications can communicate with the transient layer directly, so we could build a complete application which simply uses the Database Object and a bunch of Recordset Objects to retrieve and save data as necessary. But doing this places the responsibility of gathering data, knowing where the data lives and dealing directly with that data, in the application's hands. The data warehousing scheme provides a much richer and complete solution for developing client-server applications that take advantage of a three tier strategy and provides a much more flexible system. Besides, who wants to continuously be a slave to the front-end application, anyway? Not me, that's for sure.

Implementation

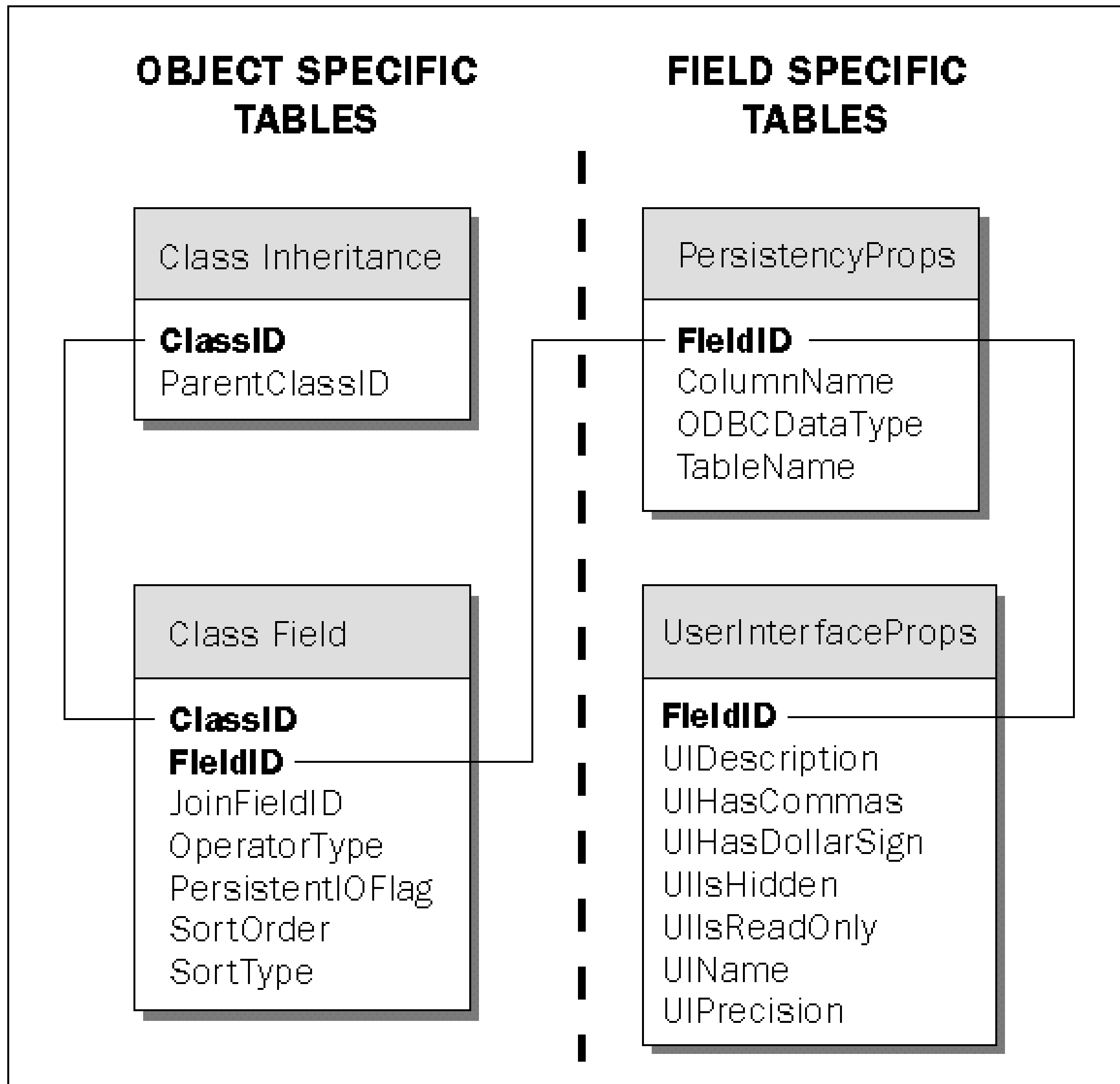
In the sections to follow, we'll examine how I implemented the objects that I've just described. We'll start by looking at the design of the back-end database, then we'll move up to the transient layer, the component layer and, finally, a client application that makes use of the lower layers to provide the user with an interface to the data. Once you have an implementation, you have more than just a model. You have working code that forces you to say, "It's Alive!".

The Back-end Tables

As you've seen, we introduced Class Objects into the component layer to provide the means of instantiating Data Objects. The Class Objects know everything about how and where to go for a Data Object's data, they know what their user interface attributes should be, and they know how to save their data. The Data Objects rely on the Class Objects to provide them with the necessary information about their own data, but where does the Class Object get its information from?

The answer is that the knowledge of where to find the information is hard coded into the Class Object itself. We've already seen one example of the way we could do this when we looked at using custom resources, but we've also seen some of the limitations of that solution. In order to avoid these problems, I decided to store the description of the Class Objects in the database. This means that I had to provide a few tables in the database to hold the information for the Class Objects.

Here's the schema I came up with:



You can see that this is quite similar to the information we stored in the custom resource example earlier in the chapter. The data stored here covers all the information needed to construct a Data Object and display it to the user.

Before I begin explaining how to use these tables, I first want to point out that each Class Object is identified by an integer which you choose when you define your classes. This will appear in the `ClassID` field of the tables shown. For example, if you have a Customer class, an Orders class, and a Details class, you might set aside the value `1000` to identify the Customer class, the value `1001` to identify the Orders class, and `1002` to identify the Details class.

> **I could have used strings such as `"Customer Class"`, `"Orders Class"`, and `"Details Class"` to identify the classes, but that would have taken up more space in the database, so I went with integers. If you're organization is large and there's a possibility of conflicting class identifiers, you should probably change the `ClassID` field so that it can accept GUIDs and use the** `Guidgen.exe` **application to generate new identifiers**.

Of course, you'd probably distribute a header file with constants defined for each of the class IDs that can be used in client applications:

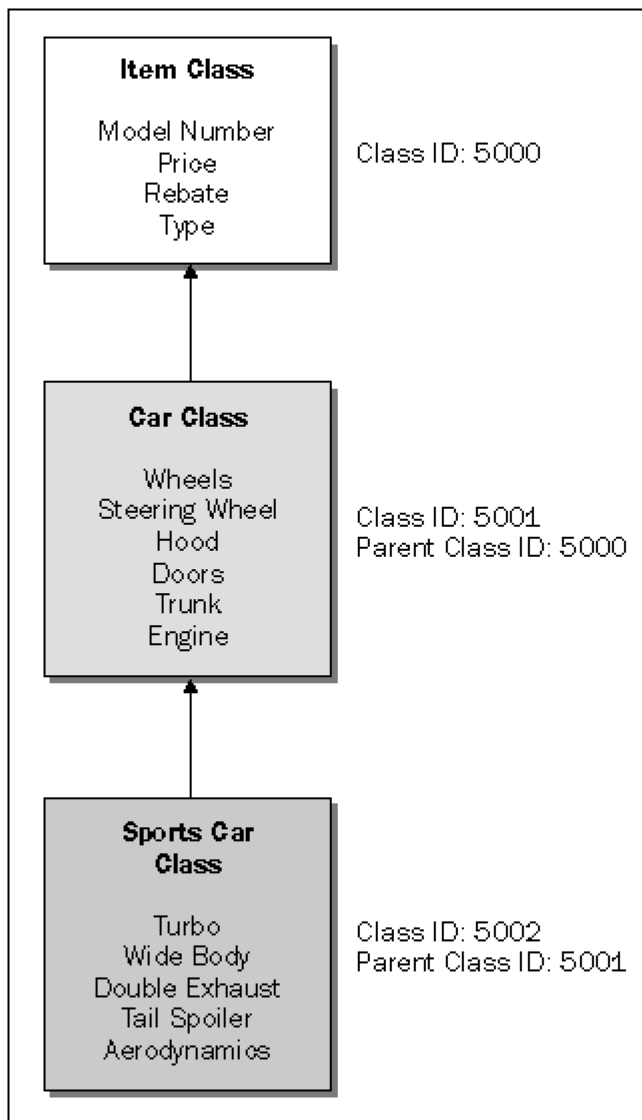
```
#define CUSTOMER_CLASS   1000
#define ORDERS_CLASS     1001
#define DETAILS_CLASS    1002
```

Now we'll look at each of the tables in the database as they're used to construct a Class Object.

ClassInheritance

The first stop is the `ClassInheritance` table, which is used to provide inheritance for the Class Objects. To elaborate on this, let's revisit the Acme Home Shopping Network.

AHSN sells many types of item, including several makes of car. Sports cars are a particular favorite with Acme's viewers, so we need to define a sports car class which inherits from the car and item classes as shown:



In this example, the Class Object for the sports car will be given the ID of **5002** when we create it. It will need to load all of the fields for its own class, plus all of the fields of its parents' classes so it begins by searching in the **ClassInheritance** table for all rows with a class ID of **5002**, returning back the parent IDs (if any records are returned). The SQL would look something like this:

```
SELECT ParentClassID FROM ClassInheritance
WHERE ClassID = 5002
```

This query will return one row (in our example). That row will contain a **ParentClassID** with the value of **5001**. The Class Object will need to hold on to its identifier as well as any parent identifiers. This can easily be done with an array of some sort. So, currently we have two values in this array:

```
INHERITANCE ARRAY
Element 1 = 5002
Element 2 = 5001
```

Next, the Class Object needs to query the table once again for any parent class of the class identified by **5001**. This can again be expressed using SQL as follows:

```
SELECT ParentClassID FROM ClassInheritance
WHERE ClassID = 5001
```

This time, the query comes back with a **ParentClassID** containing the value of **5000**. Again, this value is added to the array. The inheritance array now looks like this:

```
INHERITANCE ARRAY
Element 1 = 5002
Element 2 = 5001
Element 3 = 5000
```

The Class Object must continue to query the table until no rows are returned. So, once again, the SQL statement is executed:

```
SELECT ParentClassID FROM ClassInheritance
WHERE ClassID = 5000
```

This time, no rows are returned. We now know that we've reached the end. I'll show you the actual code I wrote to handle this situation when we cover the implementation of the component layer.

ClassField

With the inheritance array in hand, we can now proceed to load the fields of the class. Our next stop on the field loading highway is the **ClassFields** table. When I first designed this table, all it had was a mapping from the **ClassID** column to the **FieldID** column. In other words, the only purpose it served was for a SQL statement like this:

```
SELECT FieldID FROM ClassFields
WHERE ClassID = ?
```

The symbol **?** would, of course, be replaced with the appropriate value, such as **5000**, **5001** or **5002**. This SQL code would return all of the necessary field IDs needed to query some of the other tables for the attributes of the fields. Then after thinking about it some more, I realized that I wanted more flexibility so

that the classes could share the fields.

Some of the fields' attributes would depend on which class they are being used with. For example, we might have a field which needs to be sorted in descending order for one class, but in ascending order for another. Some field attributes (such as the table and column name) shouldn't change at all (regardless of the class that they are being used from). For example, if the field needs to go to the **Customer** table in order to pull out the **LastName**, that won't ever depend on the class that's doing it.

The attributes that depend on the class are stored in the **ClassFields** table, which uses both the **ClassID** and the **FieldID**, whereas those that don't depend on the class are stored in the tables with only a **FieldID** field (**PersistencyProps** and **UserInterfaceProps**).

In the example above (involving the sports car class), the **ClassFields** table will be queried three times (once each for the **ClassID**s of **5000**, **5001**, and **5002**). In each case, we'll get back a series of fields that make up the class.

Along with the field identifiers, we'll also retrieve the **JoinFieldID**, **OperatorType**, **PersistentIOFlag**, **SortOrder**, and the **SortType**. The **JoinFieldID** identifies any other fields that we want to perform a SQL join against. When the Data Object actually goes to retrieve its data, any identified join fields must be included in the query. The join field IDs are later used to query the **PersistencyProps** table for their table and column names.

The next column, **OperatorType** is used when the field will be involved in the **WHERE** clause of the SQL statement. It determines whether the field should be specified as having an operator type of **<**, **>**, **=**, **IN**, or **BETWEEN**. For example, you might want your SQL statement to look like this:

```
SELECT LastName, FirstName FROM Customer
WHERE CustomerID > 1000
```

In this case, **OperatorType** has been set to **">"**. You can also get more complicated, by using the **IN** or **BETWEEN** syntax, as follows:

```
SELECT LastName, FirstName FROM Customers
WHERE CustomerID BETWEEN 1002 AND 3892
```

The **PersistentIOFlag** is used in a similar way to how we used the key flag value in the **RCDATA** structure described earlier. The column determines how the field should be used in any SQL statements. Basically, this will determine if the field is a column that should come back from the **SELECT** statement, if it should only be used in the **WHERE** clause, or both. The value contained in this column can be any of the following values OR-ed together:

```
enum PersistentIOFlags{
   piofColumn = 1,
   piofParam = 2,
   piofJoin = 4,
   piofPrimary = 8};
```

The last two columns determine the sort order of the retrieved data when the Data Object loads its data from the actual tables and columns specified. **SortOrder** determines the order of the fields as they appear in the **ORDER BY** clause and the **SortType** determines whether they're sorted in ascending or descending order.

As you can see, these values are things that might change from class to class, which is why I placed them

in this table. The next two tables, **`PersistencyProps`** and **`UserInterfaceProps`** will depend only on the field, not the class. Both tables are searched using the field IDs gathered from the **`ClassFields`** table. We'll look at the persistency properties first.

PersistencyProps

The **`PersistencyProps`** table contains the **`FieldID`**, **`TableName`**, **`ColumnName`** and **`ODBCDataType`** fields. The field ID is used in the search for the rest of the attributes. For example, getting the persistency properties might be done using SQL as follows:

```
SELECT TableName, ColumnName, ODBCDataType
FROM PersistencyProps
WHERE FieldID = ?
```

The statement will be executed once for each field ID. The table name and column name returned will later be used by the Data Object when it needs to go after its data. The Data Object will eventually need to create a Recordset Object containing the specified columns and parameters when it's time to load its data.

The ODBC data type can be one of several constants, as follows:

ODBC Data Type Constant	Corresponding C++ Data Type
`SQL_INTEGER`	**`long`**
`SQL_DOUBLE`	**`double`**
`SQL_REAL`	**`float`**
`SQL_DECIMAL`, **`SQL_CHAR`**	**`CString`**
`SQL_TIMESTAMP`,**`SQL_DATE`**, **`SQL_TIME`**	**`CTime`**
`SQL_SMALLINT`	**`int`**
`SQL_TINYINT`	**`BYTE`**
`SQL_BIT`	**`BOOL`**

UserInterfaceProps

The last table, **`UserInterfaceProps`**, contains the values used by the user interface layer (the front-end application and any objects it provides). The columns of this table are **`UIDescription`**, **`UIHasCommas`**, **`UIHasDollarSign`**, **`UIIsHidden`**, **`UIIsReadOnly`**, **`UIName`** and **`UIPrecision`**.

The **`UIName`** is designed for use in static controls that might appear to the left of the actual value from the Data Object in the user interface. Since the name might be so short that it might not be clear what the value is, the **`UIDescription`** field contains a more lengthy string. The description can be used in tooltip controls when the mouse flies over the value. **`UIHasCommas`**, **`UIHasDollarSign`**, and **`UIPrecision`** are used to provide formatting attributes to the value displayed. **`UIIsReadOnly`** determines whether the value should be displayed at all, and **`UIIsReadOnly`** determines whether the user can change the value in the Data Object.

Automate It

When I started to consider how I would write the code, many things crossed my mind, but one thing was obvious: I needed to incorporate OLE in this picture. Why? Well, because it's the wave of the future. Judging from the amount of OLE built into Windows 95, there's no doubt that Microsoft will soon make everything in the operating system a COM object, exposing interfaces that we can call to perform different tasks. For example, in the future we might see an interface named **IWindow**. We should then be able to call the members of **IWindow**, such as **IWindow::Show()** or **IWindow::Update()**.

In the meantime, I could start to prepare for the future now by incorporating pieces of OLE into my implementation. I wanted to be able to use my implementation, not only from Visual C++ applications, but also from Visual Basic, or any other development environment that supports OLE. Since most development platforms have support for OLE Automation, I thought that it would be ideal to provide my implementation as a series of OLE automation classes.

I used Visual C++ and MFC, along with ClassWizard's support for OLE Automation to provide just want I needed. The end result was placed in two in-process servers: one for the transient layer and one for the component layer. The next few sections describe the code bed in more detail. As you read through the code samples, keep in mind that some of the error handling has been removed from what you see in these pages. The accompanying CD contains the complete code.

The Transient Layer

The first automation object that a front-end application will ever need to create is a Database Object. As I mentioned before, this object, along with the Recordset Object, can be used without any of the objects from the component layer to create complete applications, but it's best to use them in conjunction with the objects in the component layer to provide a more flexible and richer system.

You'll find the code described in the next few sections in an in-process server, called `TransientLayer.dll`.

The Database Object

Let's take a look at the class definition of the Database automation class:

```
class CWhDatabase : public CCmdTarget
{
   DECLARE_DYNCREATE(CWhDatabase)
   CWhDatabase();

// Attributes
public:
   CDatabase* GetDatabase()
      { return &(m_Database); }

protected:
   CDatabase m_Database;
   CString m_strDSN;

  // Array of Recordset objects
   CObArray m_RsArray;

// Operations
public:
   void OnRecordsetDestroyed(CObject* pObj);
   void SendConnectionDestroyed();
   void SendConnectionChanged();
```

```
// Overrides
   // ClassWizard generated virtual function overrides
   //{{AFX_VIRTUAL(CWhDatabase)
   public:
   virtual void OnFinalRelease();
   //}}AFX_VIRTUAL

// Implementation
protected:
   virtual ~CWhDatabase();

   DECLARE_OLECREATE(CWhDatabase)

   // Generated OLE dispatch map functions
   //{{AFX_DISPATCH(CWhDatabase)
   afx_msg BOOL Connect(LPCTSTR lpszDSN);
   afx_msg void Disconnect();
   afx_msg LPDISPATCH CreateRecordset(LPCTSTR lpszTableName);
   //}}AFX_DISPATCH
   DECLARE_DISPATCH_MAP()
};
```

Let's start with the data members of the class. The **m_Database** member is of type **CDatabase**, because our automation class uses an MFC **CDatabase** object to perform its communication with the database (as we'll see shortly in the member functions). Why didn't I simply derive a class from the **CDatabase** class directly? Well, to expose a class as an Automation server using MFC, the class must be derived from **CCmdTarget** and the constructor must call **CCmdTarget::EnableAutomation()**. Since **CDatabase** is not derived from **CCmdTarget**, I had to provide my own class and simply create an object of type **Cdatabase**, which I then use from my implementation.

The **m_strDSN** member is set by the client application when it calls the automation class' **Connect()** method (one of the three methods exposed by this automation class). This member provides the Data Source Name that should be used for opening a connection to a database using ODBC. Eventually, this value will be passed to the **CDatabase::Open()** function.

If the application is using the Database Object to create Recordset Objects itself, it will eventually call the **CreateRecordset()** method (which we investigate below). Each Recordset Object is added to a **CObArray** named **m_RsArray**. This is done so that if the creator of the Database Object later decides to switch to a different data source by recalling the **Connect()** method again, it can alert the Recordset Objects it holds in its array. Also, if the creator of the Database Object destroys it, without destroying the Recordset Objects, the Database Object can destroy them automatically.

Now, let's investigate the member functions of this class, starting with the exposed automation methods. The first one is the **Connect()** method. This method first closes any opened connection, calls the **CDatabase::Open()** function, and alerts the recordsets (if any have been created) of a possible data source change. Here's the code for the function:

```
BOOL CWhDatabase::Connect(LPCTSTR lpszDSN)
{
   BOOL bResult = TRUE;

   // Disconnect before opening another connection.
   Disconnect();

   // Attempt to open new connection.
   m_strDSN = lpszDSN;
   m_Database.Open(lpszDSN);

   // Change all CWhRecordsets' m_pDatabase.
   SendConnectionChanged();
```

```
    return bResult;
}
```

The next method is **Disconnect()**. This method closes any opened connection and alerts the recordsets that the connection has been closed:

```
void CWhDatabase::Disconnect()
{
   if (m_Database.IsOpen())
      m_Database.Close();

   // NULL out all CWhRecordsets' m_pDatabase.
   SendConnectionDestroyed();
}
```

And last but not least is the **CreateRecordset()** method. This method takes a single parameter: a table name to be used by the Recordset when it builds the **FROM** clause for the SQL statement. The individual fields can, of course, override this behavior by providing a table name for each field. In other words, each column or parameter in the recordset can be assigned a different table name to be used exclusively for the fields. The end result would generate SQL code that looks like this:

```
SELECT Customer.LastName, Customer.FirstName, Orders.TotalAmountDue
FROM Customer, Orders
WHERE Customer.CustomerID = ?
AND Customer.CustomerID = Orders.CustomerID
```

As you can see, a recordset can span across several tables at the same time, bringing back fields from the different tables. The **CreateRecordset()** function is implemented as follows:

```
LPDISPATCH CWhDatabase::CreateRecordset(LPCTSTR lpszTableName)
{
   // .
   // . Code to search for existence of recordset
   // .
   if (fFound)
      return pTestObj->GetIDispatch(TRUE);

   // Create a new CWhRecordset (default to lpszTableName) and return
   // the IDispatch of the object to the caller with one reference
   // count on it.
   CWhRecordset* pRS = new CWhRecordset(this, lpszTableName);
   m_RsArray.Add(pRS);
   if (pRS)
      return pRS->GetIDispatch(FALSE);

   return NULL;
}
```

The function begins by looking in the object array for the existence of the recordset. If it's found, the reference count of the object it incremented (since it's a COM object), and its **IDispatch** interface pointer is returned. Otherwise, a new Recordset Object is created, with one reference count on it, and its **IDispatch** interface pointer is returned.

The other members of the **CWhDatabase** class are **OnRecordsetDestroyed()**, **SendConnectionDestroyed()** and **SendConnectionChanged()**. These are helper functions used either by the Database Object or by the Recordset Objects. When a Recordset Object is destroyed directly by the front-end application (or the component layer), it needs to alert the Database Object so that it can remove the Recordset from its array. The Recordsets do this by calling the **OnRecordsetDestroyed()** function.

You may have noticed that the **CreateRecordset()** method sends the Database Object's **this** pointer to the Recordset Object when it is first created. The Recordset Object will hold on to this pointer so that it can call the **OnRecordsetDestroyed()** function through it.

The **SendConnectionDestroyed()** function sends notifications to the Recordset Objects held in the array when the **Disconnect()** function is called. The **SendConnectionChanged()** function sends notifications to the Recordsets when the **Connect()** function is called. These two functions allow the Recordset Objects to change any necessary state information.

The Recordset Object

Next in line is the Recordset Object. This automation object is implemented by the **CWhRecordset** class. Again, I needed to create a class derived from **CCmdTarget**, since I wanted to expose the functionality as an automation class exposing an **IDispatch** interface. My implementation, of course, takes advantage of another class, named **CGenericRecordset**, which is directly derived from MFC's **CRecordset** class.

The real work is done in the **CGenericRecordset** class, but since I couldn't expose its functionality as an OLE Automation class (because it's not derived from **CCmdTarget**), I needed a wrapper class which exposed the automation methods. When the automation methods are called, they in turn call the member functions of the **CGenericRecordset** class. Before we see the actual functionality of the **CGenericRecordset** class, let's meet the automation class, **CWhRecordset**.

There are three reasons why the **CWhRecordset** class exists. One is to expose the Recordset Object's functionality as an automation class, the second is to hold on to the Database Object's **this** pointer that created the Recordset Object, and the third is to maintain and call the functionality of a **CGenericRecordset** object as appropriate. When the automation methods of the **CWhRecordset** class are called, they call the appropriate member functions of the **CGenericRecordset**, as you can see from **CWhRecordset::AddCol()**, for example:

```
BOOL CWhRecordset::AddCol(long lODBCType, LPCTSTR lpszName)
{
   return m_Recordset.AddCol(lODBCType, lpszName);
}
```

It really does nothing more than simply provide a wrapper for the same function in the **CGenericRecordset** class.

Since the **CWhRecordset** class is actually what gets exposed as an automation class, it might at one point or another be released with a call to its **IDispatch::Release()** function inherited from the **CCmdTarget** class. When this occurs, the Recordset Object will delete itself, causing its destructor to be called which in turn calls a function in the Database Object that created the Recordset Object so that it can be aware that this Recordset Object no longer exists:

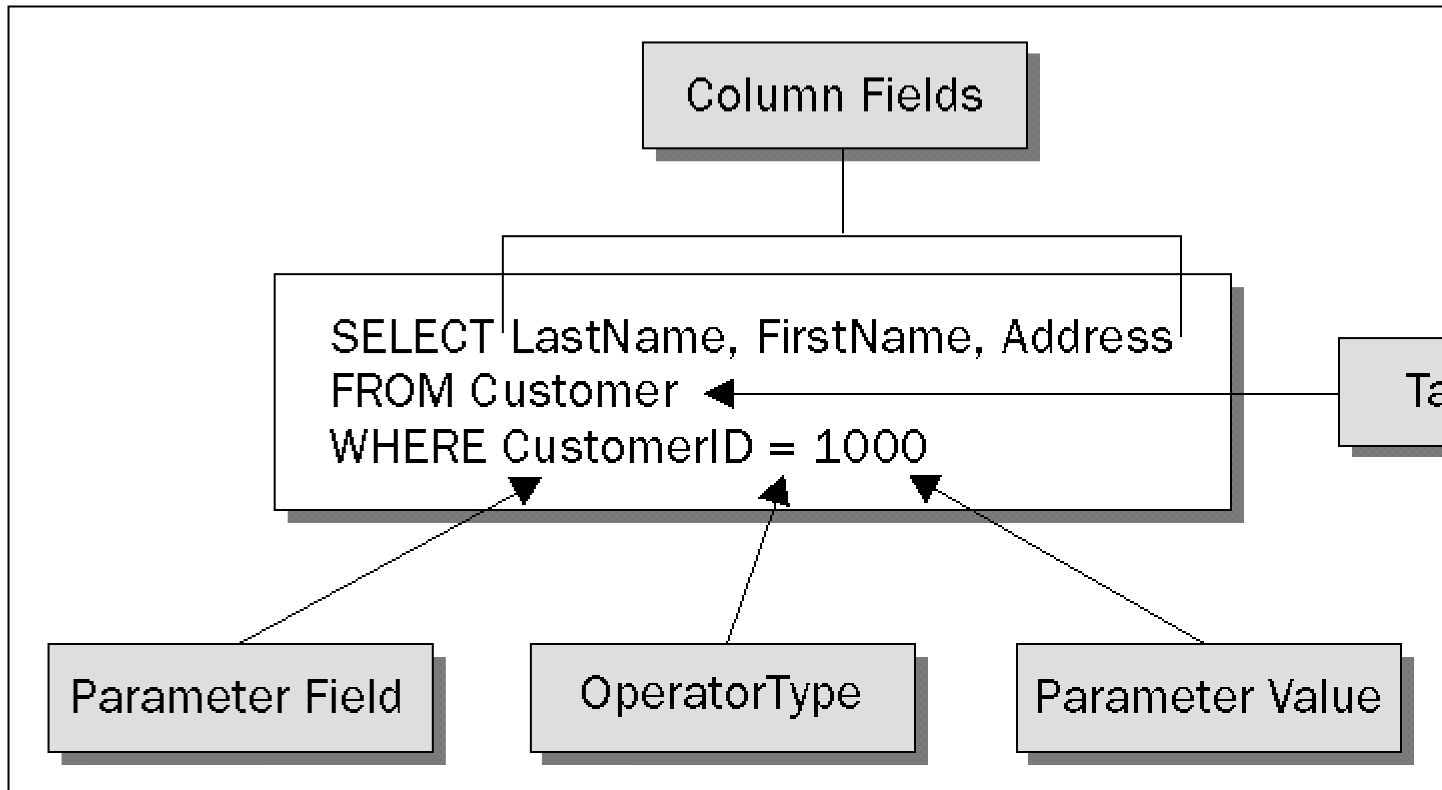
```
CWhRecordset::~CWhRecordset()
{
   // Advise the parent CWhDatabase of our destruction.
   m_pDbObj->OnRecordsetDestroyed(this);
}
```

Since the **CWhRecordset** class is nothing more than a wrapper providing OLE Automation support, the class that we really need to investigate is the **CRecordset**-derived class. This provides a very generic implementation for the **DoFieldExchange()**, **GetDefaultConnect()**, and **GetDefaultSQL()** functions,

which is exactly why I decided to call this class **`CGenericRecordset`**. Whereas you normally need to provide static information for these functions, with my class, you don't do this until runtime. The idea is that you create a generic recordset, then tell it the columns, parameters and any other pieces of information that it will need to return or update data from a database.

When it comes down to it, a recordset is only as intelligent as you make it. You have to teach it what to go after in a database, and it will attempt to do as it's told, returning data from a query simply to please you. But how do you teach it what it needs to know? Well, you start off by telling it what database table it should perform its queries on. Next, you tell it what columns it should return and, finally, you might want to tell it what criteria it should base its query on.

These steps are normally performed very easily with a SQL **`SELECT`** statement, as shown:



In the figure above, there are three column fields: a table name, one parameter field and a value to compare with the parameter field. Together, parameter fields and their values determine the criteria used on the given table. As I said before, in order to make a recordset intelligent, we must provide this information to it. Shortly, we'll meet the methods that help us to do this, but first let's meet the internal C++ structures that the recordset will need to maintain in order to provide this information later, when it performs the **`SELECT`** statement.

The FIELD Structure

The first structure is called **`FIELD`**. An instance of this structure is created each time that a column field or a parameter field is added to the recordset.

```
typedef struct tagFIELD
{
   long       lODBCType;
   CString    strTableName;
   CString    strName;
   void*      pValue;
   CString    strOperator;
   BOOL       bSortAsc;
   JOINFIELD* pJoinField;
} FIELD, *FAR LPFIELD;
```

The first member of this structure, **lODBCType,** determines the type of column or parameter (I will refer to both as simply *field*). When a field is added to the recordset, I use the value in this member to determine how much space to allocate or deallocate for the field. For example, if the **lODBCType** is **SQL_CHAR**, I know that I need to allocate enough space for a **CString** and treat it appropriately through the rest of the code. For a list of possible values, refer back to the table showing the ODBC data type constants and the corresponding C++ data types shown earlier in the chapter.

The next member, **strTableName**, contains the table within which the field is located. Does this mean that a recordset can contain fields that are retrieved from different tables? Absolutely. This is what makes the Recordset Object so powerful.

The **strName** member is used to hold the name of the field. The **pValue** member contains the allocated memory. I created this member as a **void***, since I have no way of predetermining what type of value the field will hold. If the field is a column, **pValue** will be used to hold the data of the rows returned from a query. If the field is a parameter, **pValue** will be used to hold the value to compare with the field.

The **strOperator** determines which operator should be used against the associated parameter field (this value is normally only used for parameter fields since that is where it makes the most sense). Possible values include **"="**, **"<"**, **">"**, **"LIKE"**, **"IN"**, and **"BETWEEN"**.

The next field, **bSortAsc**, determines whether the field should be sorted in ascending or descending order. This field is used when the **ORDER BY** clause is built for the SQL **SELECT** statement.

The last field points to a structure of type **JOINFIELD**, which is used to hold the table name and column name of a column used as an r-value to perform a join. The field is first tested for **NULL**. If it's not **NULL** and the current field happens to be a parameter, the values contained in the **JOINFIELD** are assigned to the r-value of the join. For example, "**WHERE** *param* = *strTableName*.*strName*". The **JOINFIELD** structure is organized as follows:

```
typedef struct tagJOINFIELD
{
   CString   strTableName;
   CString   strName;
} JOINFIELD, *FAR LPJOINFIELD;
```

m_ColFields, m_ParamFields and m_SortFields

Now for the data members of the **CGenericRecordset** class. Since we can create fields, we need somewhere to place them. This is managed with two arrays which hold pointers to column and parameter **FIELD** structures, **m_ColFields** and **m_ParamFields**, respectively.

I mentioned sorting before, but we never did see a Boolean in the **FIELD** structure to determine whether a

column should be included in the **ORDER BY** clause. So how do I determine that? I used another array, named **m_SortFields**, which simply references existing fields that are already in the **m_ColFields**. In other words, I don't use up any extra memory with a silly Boolean and I don't create another **FIELD** structure. Instead, I simply reference a **FIELD** structure that must already exist. I later traverse through the **m_SortFields** array, adding each field's name to the **ORDER BY** clause.

Since it's possible that all the columns, as well as the parameters, might belong to the same database, I give the caller the opportunity of setting a default table name, which is then used each time that a field is created. The default table name is stored into the **FIELD** structure's **strTableName** member. This value is filled in the constructor of the **CGenericRecordset** class, which is originally called from the **CWhRecordset** class. Of course, the caller has the opportunity to change this by calling a method on the recordset. In fact, the caller can change the table name of each and every field (parameter or column).

m_strDefaultSQL

As you'll soon see, the code is smart enough to create and execute several SQL statements of the same type if necessary. This is especially important for updates to the database, since only one table can be updated at a time. The Recordset has to be smart enough to create one SQL **UPDATE** statement for each table referenced in the fields.

Since a **SELECT** statement created by the generic recordset can span across several tables, the **FROM** clause of the SQL statement should reflect this. MFC will call the **GetDefaultSQL()** function whenever it's building the **FROM** clause of the SQL statement. I take the time at that point to traverse the fields and create a string that contains all of the names of the tables involved in the SQL statement. This string is then returned from the **GetDefaultSQL()** function when it is called. I stored this string into a data member named **m_strDefaultSQL**.

m_nNumOfCols

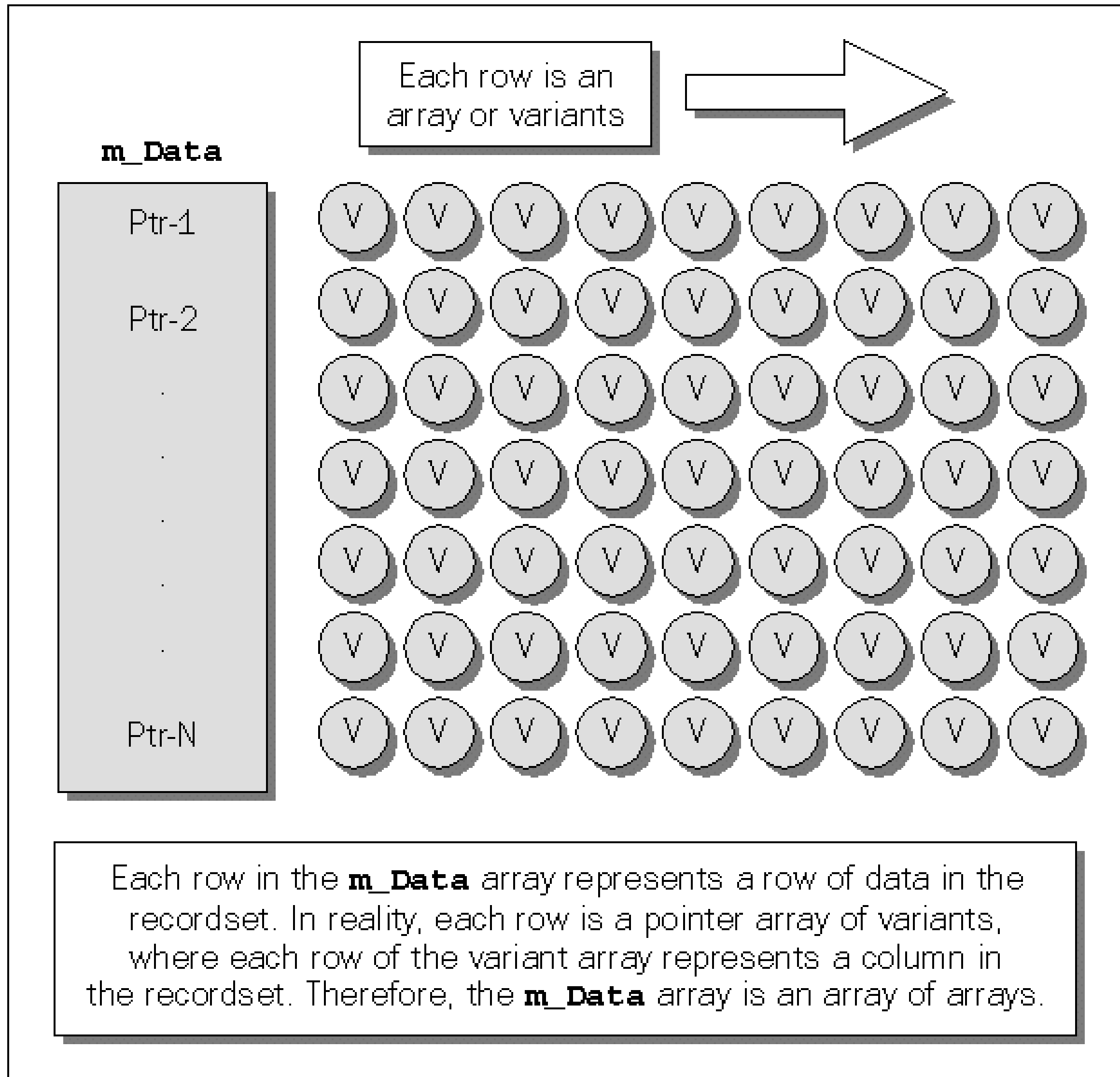
Since the generic recordset is dynamic by nature, the caller can add more columns after a query has returned, so how do we know how many columns were involved in the last query? I keep this in a data member, named **m_nNumOfCols**. Let's say that the calling application adds five columns to the recordset, then tells the recordset to fetch the data. Next, it adds five more columns, but before fetching more data, it decides to read the data from the first fetch. Since the last five columns don't contain data as of yet, only the first five columns should be read.

One of the things that I wanted for my Recordset Object is the ability to close down the MFC recordset when an operation has completed. Why? Well, it seems that, in my experience, MFC recordsets (using the underlying ODBC API) are messy creatures. They open several connections (for each recordset) on LAN based databases and these connections remain open until the recordset is closed. Now I really couldn't put my figure on who the culprit was. Is it the driver, or MFC, or the underlying ODBC API? Every time I asked questions, I always got the run around and was told that it was the other guy's fault.

I decided that I would once and for all fix the problem myself. The solution was to open a recordset (using MFC), which causes the **SELECT** statement to be executed, read in all of the data into a tabular volatile data storage area and close the MFC recordset. You're probably wondering what happens when you update the data. Don't you need the recordset to remain opened for updates, additions, or deletions?. The answer is *no*. I can build the **UPDATE** statement (or any other SQL statement) myself and execute it using the **CDatabase::ExecuteSQL()** function.

m_Data and m_iRow

I don't know whether you noticed or not, but the **FIELD** structure only holds one value, which means that the column fields can only ever hold one row of data. This is why I created the **m_Data** member. This member is a pointer array, and each element holds an array of **VARIANTs**.



The recordset contains functions which allow the caller to traverse this array. When these functions are called, they move a row of data from the **m_Data** array into the column fields' **pValue** member. For example, if the column fields currently contain the values from the first row in the **m_Data** array and the application tells the recordset to move to the next row, the recordset goes to the **m_Data** array and copies the values of the next row into the columns' **pValue** member. Next, the caller accesses the data directly from the columns' **pValue** member via another member function.

I used **VARIANTs** because it's impossible to tell what type of data will be returned to the application from each column at compile time, I need to make the recordset as generic as possible and **VARIANTs** are perfect for this type of situation.

To keep track of which row of data we're looking at, I created a data member, called **m_iRow**, which simply keeps the row number of the **m_Data** array which was last copied to the column fields.

m_bEOF and m_bBOF

Since recordsets allow callers to move up and down the rows of data, there must be a way that the caller can detect when it has reached the end or the beginning. The following pseudo code describes what an application might do:

```
move to the first recordset
begin loop
  if we've reached the end of the recordset
    end the loop
  else
    access data from the current row
    move to the next row
go to "begin loop"
```

This loop can also take place in the opposite direction. To keep track of when the recordset has gone past the end of the rows of data, or past the beginning of the recordset, I use the members, **m_bEOF** and **m_bBOF**, respectively.

m_nOperation

When they're updating or adding new rows, users of MFC's recordset class use a protocol that goes something like this:

For updates:

Call **CRecordset::Edit()**, which makes a copy of the current row for later optimization of the update. This allows MFC to determine which columns have actually changed and only include those fields in the update.

Change the necessary columns.

Call **CRecordset::Update()** to complete the update and commit the changes to the database.

For additions:

Call **CRecordset::AddNew()**, which creates a new row in memory and initializes the columns.

Store the appropriate values into the columns.

Call **CRecordset::Update()** to complete the addition and commit the changes to the database.

If, at any point, the caller wants to cancel the edits or the addition, it should call **CRecordset::Cancel()**. This protocol works as long as the recordset remains opened and connected to the database.

I wanted to mimic this protocol in my generic recordset and since I actually close the recordset (as I explained above), I needed to do all of the work myself. I first created a data member, named **m_nOperation**, which would hold one of several values from an enumeration:

```
enum OPERATIONS  {opNone, opUpdate, opInsert};
```

If the application calls **Edit()**, I set **m_nOperation** to **opUpdate**. If the application calls **AddNew()**, I set **m_nOperation** to **opInsert**. When the **Update()** function or the **Cancel()** functions are called I reset the **m_nOperation** member to **opNone** (its default value).

m_UpdateCmds

The last data member, **`m_UpdateCmds`**, is related to updates and inserts. Remember that I mentioned that my recordsets are smart enough to create separate SQL statements if they need to. This needs to be done if the columns being updated or inserted contain different table names. For example, there might be two columns from **`TableA`**, and three columns from **`TableB`**. The **`UPDATE`** statement needs to know what table to write the changes to, but there can only be one table specified in the statement. The statement for **`TableA`** looks something like this:

```
UPDATE TableA SET Col1 = Value1, Col2 = Value2
```

Another statement would need to be created for **`TableB`**. I use **`m_UpdateCmds`** to contain all the generated statements before they are executed, then I simply traverse through this array of commands, executing each one as I move from row to row.

CGenericRecordset Member Functions

So far, we've discussed the data members of the **`CGenericRecordset`** class, but we haven't seen any of the member functions yet. There are only really a handful of functions that actually perform a lot of work. The rest are trivial and can be explained with a sentence or two.

Let's first look at the member functions as they are exposed by the **`CWhRecordset`** class. In other words, these are the member functions of **`CGenericRecordset`** which are exposed as automation methods:

Query Automation Methods	Description
`BOOL AddCol(long lODBCType, LPCTSTR lpszName)`	Adds a column for data input from the data source.
`BOOL AddParam(long lODBCType, LPCTSTR lpszName)`	Adds a parameter for data querying from the data source. This eventually becomes the **`WHERE`** clause for the **`SELECT`** statement.
`BOOL SetColValue(LPCTSTR lpszName, const VARIANT FAR& Value)`	Sets the column value for writing to the data source.
`BOOL SetParamValue(LPCTSTR lpszName, const VARIANT FAR& Value)`	Sets the parameter value for the SQL query to the data source.
`BOOL SetParamOperator(LPCTSTR lpszName, LPCTSTR lpszOperator)`	Changes the operator used by the parameter from the default **`=`** operator to something else for querying. e.g.: **`>=`**, **`<=`**, **`IN`**, **`BETWEEN`**, **`LIKE`**, etc.
`BOOL SortCol(LPCTSTR lpszName, BOOL bAscending)`	Creates the list of columns to be sorted by calling this function. For each, also specify the direction. This will eventually become the **`ORDER BY`** clause of the **`SELECT`** statement.
`BOOL AddJoinToParam(LPCTSTR lpszName, LPCTSTR lpszJoinTableName, LPCTSTR lpszJoinName)`	Specifies a **`Table.ColName`** to be added as a join to a parameter. Note that only one join can be added to each parameter field. e.g.:

	`TableX.FieldX = JoinTableName.JoinFieldName`
`BOOL SetColTableName(LPCTSTR lpszName, LPCTSTR lpszTableName)`	Looks up the column name and changes its table name to something other than the default.
`BOOL SetParamTableName(LPCTSTR lpszName, LPCTSTR lpszTableName)`	Looks up the parameter name and changes its table name to something other than the default

.

Navigation Automation Methods	Description
`BOOL MoveTo(short iRow)`	Allows you to move to a particular row in the result set. In reality, it moves the given row from the `m_Data` array to the columns' `pValue` member.
`void MoveFirst()`	Moves the data from the first row into the fetchable area.
`void MoveNext()`	Moves the data from the next row (if any) into the fetchable area.
`void MovePrev()`	Moves the data from the previous row (if any) into the fetchable area.
`void MoveLast()`	Moves the data from the last row into the fetchable area.
`BOOL IsBOF()`	Returns `TRUE` if we moved backwards past the first row.
`BOOL IsEOF()`	Returns `TRUE` if we moved forwards past the last row in ascending direction.

Data Retrieval Automation Methods	Description
`BOOL Fetch()`	Executes the composed SQL statement on the data source.
`VARIANT GetColValue(LPCTSTR lpszName)`	Fetches a value from the given column and returns the value at that column from the current row.

Edit State Automation Methods	Description
`BOOL AddNew()`	Directs the recordset to add a new row to the data source. The new row is not actually added to the database until `Update()` is called. Once `AddNew()` has been called, the caller *cannot* call `Edit()`. The caller should call `Update()` or `Cancel()`, to change the state.
`BOOL Edit()`	Directs the recordset to prepare for changes to the fields. The new changes are not performed on the data source until `Update` is called. Once `Edit()` has been called, the caller *cannot* call `AddNew()`. The caller should call `Update()` or `Cancel()`, to change the state.
`BOOL Update()`	Commits any new records (created with `AddNew()`) or

	changes (created with **Edit()**) to the data source and returns the state of the recordset to normal.
void Cancel()	Cancels any new records or changes to the data source and returns the state of the recordset to normal.

In order to implement some of these functions, I created several helper functions. For example, when the application calls the **AddCol()** or **AddParam()** methods, these functions both call a helper function, named **CreateField()**, to return a newly created and initialized **FIELD** structure. Once the **AddCol()** or **AddParam()** functions receive back the **FIELD** structure, they add the object to the appropriate field array (**m_ColFields** or **m_ParamFields**, as explained above). Here you can see the **CreateField()** function:

```
LPFIELD CGenericRecordset::CreateField(long lODBCType, LPCTSTR lpszName)
{
   // Create a new FIELD object and initialize it.
   FIELD* pField = new FIELD;

   pField->lODBCType = lODBCType;
   pField->strTableName = m_strTableName;
   pField->strName = lpszName;
   pField->strOperator = _T("=");
   pField->bSortAsc = TRUE;
   pField->pValue = NULL;
   pField->pJoinField = NULL;

   // Create place to store data coming from data server in case this
   // field is being created for a col. Parameters' pValues are reset
   // when value is assigned each time. The reason that we don't use
   // VARIANT type instead, is because the TransferData() function needs
   // to work with the native data type transfering data values from
   // ODBC into our Recordset.
   switch(pField->lODBCType)
   {
      case SQL_INTEGER:
         pField->pValue = new long;
         break;

      case SQL_DOUBLE:
         pField->pValue = new double;
         break;

         .
         . Cases for other types
         .

      default:   // No supported type found.
         delete pField;
         return NULL;
   }

   return pField;
}
```

The **AddField()** function first creates a **FIELD** object, then initializes it with default values and, finally, it allocates memory for the appropriate type of value that will be stored later when data is returned from a query, or when the application stores values to be used as the parameter's criteria.

When a value is stored into one of these fields, another generic function performs the work. This function is the **SetFieldValue()** function and it looks like this:

```
BOOL CGenericRecordset::SetFieldValue(FIELD* pField, const COleVariant&
   Value)
```

```
{
   COleVariant va = Value;

   DeleteField(pField, FALSE);

   // Must be string if operator is "IN" or "BETWEEN".
   if (pField->strOperator.CompareNoCase(_T("IN")) == 0 ||
       pField->strOperator.CompareNoCase(_T("BETWEEN")) == 0)
   {
      if (va.vt != VT_BSTR)
         return FALSE;

      pField->pValue = new CString;
      pField->lODBCType = SQL_CHAR;
      *((CString*)pField->pValue) = va.bstrVal;
      return TRUE;
   }

   switch(pField->lODBCType)
   {
      case SQL_INTEGER:
         // Convert numeric types to a known type.
         VariantChangeType(&va, &va, 0, VT_I4);
         pField->pValue = new long;
         *((long*)pField->pValue) = va.lVal;
         break;

      case SQL_DOUBLE:
         // Convert numeric types to a known type.
         VariantChangeType(&va, &va, 0, VT_R8);
         pField->pValue = new double;
         *((double*)pField->pValue) = va.dblVal;
         break;
         .
         . Cases for other types
         .
      default:
         return FALSE;
   }

   return TRUE;
}
```

This function begins by deallocating any memory that the **pValue** member might have, in case the operator has been changed to an **"IN"** or **"BETWEEN"**, which always require the value to be a string. That is, the application will need to pass the value for an **"IN"** or a **"BETWEEN"** as a string because that's the way I've designed it. Next, the function tests for this situation and acts appropriately. Finally, if the operator has not been changed, it recreates the **pValue** field and stores the given value into it. This function could probably be optimized, but it works for now.

The **CGenericRecordset::GetFieldValue()** is called from the **GetColValue()** function. I didn't implement a **GetParamValue()** function because I didn't see any reason for one, but you can feel free to add one if you see the need for it. The **GetFieldValue()** function performs a similar **switch** statement on the field passed to it and returns the value as a **VARIANT**. Before an application calls **GetColValue()**, the data must be moved from the **m_Data** array to the columns' **pValue** member. Let see an example of this:

```
BOOL CGenericRecordset::MoveTo(short iRow)
{
   // Get the number of rows in the tabular data.
   int nSize = m_Data.GetSize();
   if (iRow >= 0 && nSize > iRow)    // Is it a legal row?
   {
      m_bBOF = FALSE;
      m_bEOF = FALSE;
      m_iRow = iRow;
```

```
        // Retrieve the given row in the tabular data
        // and copy it to the obtainable area.
        COleVariant* pva = (COleVariant*)m_Data[m_iRow];
        FIELD* pField;
        for(int i = 0; i < m_nNumOfCols; i++)
        {
           pField = (FIELD*)m_ColFields.GetAt(i);
           SetFieldValue(pField, pva[i]);
        }

        return TRUE;
     }

     // Reset to indicate there are no rows.
     m_iRow = -1;
     m_bBOF = TRUE;
     m_bEOF = TRUE;

     return TRUE;
  }
```

The other functions that move data from the **m_Data** array into the columns behave in a similar manner.

Once the recordset has been told to execute its **SELECT** statement (by calling the **Fetch()** function), the recordset traverses through all of the data in the MFC recordset and stores it into the **m_Data** array. Let's examine this a bit closer with some code:

```
while (CRecordset::IsEOF() != TRUE)
{
   pva = new COleVariant[m_nNumOfCols];
   for (int i = 0; i < m_nNumOfCols; i++)
       pva[i] = GetFieldValue((FIELD*)m_ColFields.GetAt(i));
   m_Data.Add(pva);
   CRecordset::MoveNext();
}
```

GetFieldValue() will end up picking up whatever is placed into the columns' **pValue** member by **TransferData()** which is called from my override of **CRecordset::DoFieldExchange()**, which is called by the framework each time I call **CRecordset::MoveNext()**. Let's look at these two functions next to get a clearer picture. First, the **DoFieldExchange()** function:

```
void CGenericRecordset::DoFieldExchange(CFieldExchange* pFX)
{
   // The columns coming back in the recordset.
   if (m_ColFields.GetSize() > 0)
   {
      pFX->SetFieldType(CFieldExchange::outputColumn);
      TransferData(m_ColFields, pFX);
   }

   // The values used to build the query statement.
   if (m_ParamFields.GetSize() > 0)
   {
      pFX->SetFieldType(CFieldExchange::param);
      TransferData(m_ParamFields, pFX);
   }
}
```

This isn't what you're used to seeing in a **DoFieldExchange()** function, is it? That's because I made it as generic as possible. The real work is performed in the **TransferData()** function:

```
BOOL CGenericRecordset::TransferData(CPtrArray &array,
                                     CFieldExchange* pFX)
```

```
{
   int nSize = array.GetSize();
   for(int i = 0; i < nSize; i++)
   {
      FIELD* pField = (FIELD*)array[i];

      // Check for operator = 'IN' or 'BETWEEN'
      if (pField->strOperator.CompareNoCase(_T("IN")) == 0 ||
          pField->strOperator.CompareNoCase(_T("BETWEEN")) == 0)
         continue;   // This one is handled in
                     // the Fetch() function

      // Handle all other transfers.
      switch (pField->lODBCType)
      {
         case SQL_INTEGER:
            RFX_Long(pFX, pField->strName, *((long*&)pField->pValue));
            break;

         case SQL_DOUBLE:
            RFX_Double(pFX, pField->strName, *((double*&)pField->pValue));
            break;
         .
         . Other cases handled here
         .
         default:
            return FALSE;
      }

   }

   return TRUE;
}
```

As you can see from the code, the abilities of the class are pretty extensive. The ability to build the column information and parameter information at run time for the `DoFieldExchange()` is what makes this whole thing work.

So, now you've seen how the data gets into the `FIELD` structures, moves into the `m_Data` array and back into the `FIELD` structures when needed for returning values to the application via the `GetColValue()` function. The other function implementations are fairly trivial, so you should be able to understand their implementation from walking through the rest of the code.

The Component Layer

This layer is where the data warehousing really begins. The transient layer is just a stepping stone for the data to get from the storage medium to the component layer. Whereas the transient layer provides the pipe for the data to move from one place to another, the component layer provides a home full of intelligence about the data. Think of the component layer as data that has gone to college.

The code you are about to see was implemented in an in-process automation server named `ComponentLayer.dll`.

The Container Object

Container Objects exist to produce components for applications. Container Objects are intelligent enough to create and maintain Data Objects and Class Objects (we'll meet the implementation of these objects soon enough). An application only needs to create a single Container Object, unless it wants to create a different Container Object for each database connection it has established. The implementation for my Container Object is in the form of a class named **`CWhClassContainer`**. This class is derived from MFC's **`CCmdTarget`** and exposes several methods via OLE Automation: **`Initialize()`**, **`CreateComponent()`**, and **`DestroyAll()`**.

A Container Object can't do anything until it's told where it should look for its information. In other words, it can't build components until it knows where to point the components for their information. This is done by passing the Container Object a Database Object that has already been connected to a data source. The connection object will hold onto this Database Object in a data member named **`m_Database`**. The Database Object is passed from the application to the Container Object as a parameter to the **`Initialize()`** function:

```
void CWhClassContainer::Initialize(LPDISPATCH pDbObj)
{
   pDbObj->AddRef();

   // Convert to wrapper class and store it.
   m_Database.AttachDispatch(pDbObj);
}
```

Since this server needs to use the Database and Recordset Objects introduced in the transient layer, I got ClassWizard to read the type library of that server and generate the appropriate wrapper classes for me. Here you can see that I attach the dispatch pointer passed in to the function to an object of the Database wrapper class.

The next thing that an application will want to do with a Container Object is to have it create components and return them for use by the application. An application can accomplish this by calling the **`CreateComponent()`** method of the Container Object. This is what the code looks like:

```
LPDISPATCH CWhClassContainer::CreateComponent(long lClassID, long lKey)
{
   CWhDataObject* pDataObj = NULL;

   // Lets go after the Data-Object.
   WH_KEY_PROLOGUE(lClassID, lKey) // Make strDataObjKey

   // If we do not find it then we create it.
   if (!m_WhDataObjMap.Lookup(strDataObjKey, (CObject*&)pDataObj))
   {
      CWhClassObject* pClassObj = NULL;
```

```
            // If not found Class object, create it.
            if (!m_WhClassObjMap.Lookup(lClassID, (CObject*&)pClassObj))
            {
               pClassObj = new CWhClassObject(this, lClassID));
               // Store it in the map.
               m_WhClassObjMap.SetAt(lClassID, (CObject*&)pClassObj);
            }
            else
               pClassObj->ExternalAddRef();

            // Create the Data-Object next.
            pDataObj = new CWhDataObject(this, pClassObj, lKey));
            // Store it in the map.
            m_WhDataObjMap.SetAt(strDataObjKey, (CObject*&)pDataObj);
         }
         else   // If found, add a reference and return it.
            pDataObj->ExternalAddRef();

         return pDataObj->GetIDispatch(FALSE);
      }
```

Let's start at the top of this function. If you remember from our discussion of the component identifier key, we learned that an application must provide something to uniquely identify a component. The component will need to know what class information it should use, and whether or not to return a Data Object if it has the data that the application is interested in. Let me clarify this some more with an example. Say that an application wishes to create a customer component for the customer whose customer ID is **`1000`**. It might, for example, call **`CreateComponent()`**, passing it **`CUSTOMER_CLASS`** (an ID that represents the class) and **`1000`** for the **`lKey`** value.

The **`CreateComponent()`** function creates a hashed value out of the class ID-key combination, which uniquely identifies the Data Object. If, at a later time, the application calls **`CreateComponent()`** again, asking for the same component (which is specified by the same class ID/key combination), the application would get back exactly the same Data Object.

The hashed value is created with a call to **`WH_KEY_PROLOGUE`**, which looks like the following:

```
      #define WH_KEY_PROLOGUE(lClassID, lKey) \
         CString strDataObjKey;           \
         strDataObjKey.Format(_T("%ld-%ld"), lClassID, lKey);
```

This macro should only be called once from a function, since it creates an automatic variable inside the macro.

The **`CreateComponent()`** function next looks in an MFC map, **`m_WhDataObjMap`**, for the existence of a Data Object with a matching hashed value. If it doesn't find one, it looks for the requested Class Object in another MFC map, **`m_WhClassObjMap`**. If it doesn't find that either, it creates the Class Object, then the Data Object. Both objects are then added to the appropriate maps for further searches. However, if the objects are located, their reference counts are increased and the objects are returned. In fact, only the Data Object's **`IDispatch`** is returned; the Class Object is simply passed to the Data Object when the Data Object is created.

The last function, **`DestroyAll()`**, simply goes through the maps and destroys the Class Objects and the Data Objects.

The Class Object

Now we'll meet the Class Object in much more detail. The Class Object is implemented by a

CCmdTarget-derived class named **CWhClassObject**. If you look at the **CreateComponent()** code, you'll notice that the Container Object passes its **this** pointer to the Class Object's constructor. This is done so that the Class Object can access the Database Object from the container when it needs to load its fields and attributes. The following code shows the constructor for the **CWhClassObject**:

```
CWhClassObject::CWhClassObject(CWhClassContainer* pClassContainerObj,
   long lClassID)
{
   EnableAutomation();

   // Initialize
   m_lClassID = lClassID;
   m_pClassContainerObj = pClassContainerObj;
   m_posEnumField = NULL;

   CWhDatabase* pDbObj = m_pClassContainerObj->GetDatabase();

   // 1. Build a CompType Inheritance List by traversing
   // the Parent-IDs.
   LoadInheritanceMap(pDbObj);

   // 2. Load all the Field-IDs. Overload fields if
   // necessary.
   LoadFields(pDbObj);
}
```

Again, let's take it from the top. The Class Object holds onto its class ID for later use if necessary (it will use it when loading its fields and finding the classes that it inherits from if there are any). Next, it holds on to the Container Object in case it needs to access a different Database Object later on.

m_posEnumField needs some explaining. The **m_posEnumField** data member is a **POSITION** object used in conjunction with a map that contains Field Objects of type **CWhClassField** (We'll come back to this class shortly. We haven't mentioned it before because it's an implementation detail of the component layer.) Since the Class Object is a collection of fields, these fields might need to be enumerated by the application or the Data Object (when loading its data). This is easily performed by the functions **EnumFirstFieldID()** and **EnumNextFieldID()**. When you first call **EnumFirstFieldID()**, the function returns the ID of the first field and initializes the **m_posEnumField** to the position of the first Field Object.

Once you've called **EnumFirstFieldID()**, you don't call it again unless you wish to access the ID of the first field again, causing the **m_posEnumField** member to be reset. The other function, **EnumNextFieldID()**, is called repeatedly to traverse through the rest of the field IDs. Each time you call it, it updates the **m_posEnumField** so that it points to the position of the next field until, of course, you reach the end (in which case **m_posEnumField** is set to **NULL**).

The next thing to happen in the constructor is to access the Database Object from the Container Object and pass it to the **LoadInheritanceMap()** and **LoadFields()** functions.

We've discussed how inheritance works with the Class Objects, but we haven't seen it actually implemented. If you remember from our discussion, we need to go out to the **ClassInheritance** table and find all of the parent classes from which the current class should inherit fields. Here's how I do it:

```
BOOL CWhClassObject::LoadInheritanceMap(CWhDatabase* pDbObj)
{
   // This class' ID will definitely be needed, so let's add it now.
   m_InheritArray.Add(m_lClassID);

   CWhRecordset ClassInheritRS;
   VARIANT varg;
```

```
   VariantInit(&varg);
   varg.vt = VT_I4;
   varg.lVal = m_lClassID;

   // Create the "ClassInheritance" recordset.
   ClassInheritRS.AttachDispatch(pDbObj->CreateRecordset(
                                          _T("ClassInheritance")));
   ClassInheritRS.AddCol(SQL_INTEGER, _T("ParentClassID"));
   ClassInheritRS.AddParam(SQL_INTEGER, _T("ClassID"));

   // Keep going until we traverse to the top of the hierarchy.
   while (TRUE)
   {
      // Set the class ID to the last class ID read.
      ClassInheritRS.SetParamValue(_T("ClassID"), varg);
      // Read the next one up in the hierarchy.
      ClassInheritRS.Fetch();

      // If nothing was returned, we're finished.
      if (ClassInheritRS.IsEOF())
         break;

      // Fetch the one and only value returned.
      varg = ClassInheritRS.GetColValue(_T("ParentClassID"));

      m_InheritArray.Add(varg.lVal);
   }

   return TRUE;
}
```

Note how this function uses our OLE Automation Recordset Object to access the data. I make use of the automation wrapper class created by ClassWizard for the Recordset Object and attach the **IDispatch** returned from **CreateObject()** to an instance of this class. I then continue as normal, calling the automation methods of the Recordset Object.

There's one other thing that I want to point out about this function: it makes use of a data member named **m_InheritArray**. This array member will hold the IDs of the classes involved in the class hierarchy when the function has completed. That way, the next function, **LoadFields()**, can search this array and load all the fields for each class ID, causing the current Class Object to inherit all of the fields necessary. At the very least, this array will contain the class ID of the current Class Object that we're trying to load. This ID is assigned to the array at the very top of the **LoadInheritanceMap()** function.

Once **m_InheritArray** has been filled, the **LoadFields()** function is called. This function begins by creating a Recordset to communicate with the **ClassFields** table in the given database. It will use this table to access all of the fields. As I explained earlier in the chapter, this table also contains the properties that are dependent on the Class Object. The function next loads the fields for each class ID in the **m_InheritArray** and loads the fields and the properties from the database using the Recordset Object. A **CWhClassField** object is created for each field. This class maintains data members for all of the properties. The Field Objects are created like this:

```
pField = new CWhClassField(pDbObj, varg.lVal);
```

When the Field Object has been created, the properties accessed from the recordset are assigned to the field and, finally, the Field Object is put in the **m_FieldMap**:

```
pField->m_lOperatorType = vtConvertToLong(
   ClassFieldRS->GetColValue(_T("OperatorType")));
pField->m_lSortOrder = vtConvertToLong(
   ClassFieldRS->GetColValue(_T("SortOrder")));
pField->m_bSortType = vtConvertToBool(
```

```
      ClassFieldRS->GetColValue(_T("SortType")));
   pField->m_lIOFlag = vtConvertToLong(
      ClassFieldRS->GetColValue(_T("PersistentIOFlag")));
   pField->m_lJoinFieldID = vtConvertToLong(
      ClassFieldRS->GetColValue(_T("JoinFieldID")));
   m_FieldMap[varg.lVal] = pField;
```

These Field Objects are only used internally within this layer; they are never exposed to the outside world. Any interested parties will see the values held by the Field Objects via the Class Objects. This is possible because the Class Objects expose several methods that allow access to the properties held by the Field Objects. In reality, the methods exposed by the Class Objects do nothing more than to look up the appropriate Field Object and access the requested property.

The following is a list of the automation methods exposed by the Class Object. Most of these methods simply return the value for the properties accessed from the database tables. You should now be familiar with these properties, since I mentioned each one when I described the database tables above. The only functions here that don't return a property gathered from the database are **`GetClassID()`** (which returns the class ID of the Class Object, which it is holding on to) and **`EnumFirstFieldID()`**, and **`EnumNextFieldID()`** (which I have already explained).

```
long CWhClassObject::GetClassID()
BSTR CWhClassObject::GetName(long lFieldID)
BSTR CWhClassObject::GetDescription(long lFieldID)
long CWhClassObject::GetPrecision(long lFieldID)
BOOL CWhClassObject::IsHidden(long lFieldID)
BOOL CWhClassObject::HasDollarSign(long lFieldID)
BOOL CWhClassObject::HasCommas(long lFieldID)
BOOL CWhClassObject::IsReadOnly(long lFieldID)
BSTR CWhClassObject::GetTableName(long lFieldID)
BSTR CWhClassObject::GetColumnName(long lFieldID)
long CWhClassObject::GetDataType(long lFieldID)
long CWhClassObject::GetIOFlag(long lFieldID)
long CWhClassObject::GetJoinFieldID(long lFieldID)
long CWhClassObject::EnumFirstFieldID()
long CWhClassObject::EnumNextFieldID()
BOOL CWhClassObject::GetSortType(long lFieldID)
long CWhClassObject::GetOperatorType(long lFieldID)
BSTR CWhClassObject::GetOperator(long lFieldID)
long CWhClassObject::GetSortOrder(long lFieldID)
```

Now let's take a close look at the **`CWhClassField`** class used by the Class Object. When its constructor is called, it calls two other member functions: one to load the persistent properties and the other to load the user interface properties from the tables mentioned above. This is what the constructor looks like:

```
CWhClassField::CWhClassField(CWhDatabase* pDbObj, long lFieldID)
{
   // Initialize this object.
   m_lFieldID = lFieldID;

   // Load its common characteristics.
   LoadUIProperties(pDbObj);
   LoadPersistentProperties(pDbObj);
}
```

By now it should be pretty obvious what the **`LoadUIProperties()`** and the **`LoadPersistentProperties()`** functions will do. They simply create Recordset Objects that point to the appropriate tables, then load the properties for the given field ID. Since the Field Object holds onto its field ID, the process of loading the properties becomes trivial. (I love it when that happens. Goes to show what a good object-oriented design can do for you!)

The Data Object

The last object (but definitely not the least) we need to discuss is the Data Object. The code I implemented for the Data Object can be found in the form of a class, named **CWhDataObject**.

We already understand the role that the Data Object plays in the data warehouse, but how is it implemented? Before we answer this question, let's step back a second and study the Data Object in a brighter light.

We've learned that it's the Class Object that contains the fields of a class definition. But it's the Data Object that maintains the data returned from the data source. This data is gathered based on information stored in the associated Class Object. If the Class Object says that the first field contains a column name of **FirstName** and the table name is the **Customer** table, it's the responsibility of the Data Object to create an appropriate recordset to access the specified column within the specified table.

Class objects are not only responsible for determining the columns that should be used in a **SELECT** clause, they also contain the columns to be used as parameters in the **WHERE** clause. As a Data Object loads it data (pertaining to the columns specified by the Class Object), it needs to know what parameters should be set up before it loads the data. That way, the application will have a Data Object that contains exactly the information that it's interested in.

For example, let's say that the application makes a request for an Order object to be created. The application calls the Container Object's **CreateComponent()** function and specifies **1000** for the class ID (if this is the class ID of the Order class in the database) and **1234** for the unique identification key for the Data Object. A Class Object is created and initialized.

In its initialization process, it loads its fields from the **ClassFields** table and ends up with three fields. The first two, **OrderID** and **OrderDate**, are columns, but the third, **CustomerID**, is a parameter. All the fields are taken from the same table: **Orders**. After building a **SELECT** statement from the information, the statement might look something like the following:

```
SELECT OrderID, OrderData
FROM Orders
WHERE CustomerID = ?
```

This object is most likely used to create a Data Object containing all of the orders for a given customer. But how does the Data Object know which customer we want the information returned for? The answer is that the application must pass it to us, which means that applications must have some knowledge of the objects that they are trying to create, so that they can be intelligent enough to specify appropriate values for the Data Object's parameter fields.

This means that before an application can tell a Data Object to load its data, it must first tell it *how* to do it. I actually allocate room for the values that will be used in assigning values to parameter fields once the recordsets have been created. I do this in the constructor of the Data Object's C++ class, like this:

```
CWhDataObject::CWhDataObject(CWhClassContainer* pClassContainerObj,
   CWhClassObject* pClassObj, long lKey)
{
   EnableAutomation();

   m_bLoaded = FALSE;
   m_lKey = lKey;

   m_pClassContainerObj = pClassContainerObj;
```

```
      m_pClassObj = pClassObj;

      // Setup the param map for RS-Params future use.
      // Define variables for fields.
      long lFieldID = m_pClassObj->EnumFirstFieldID();
      long lIOFlag;
      VARIANT* pva;
      ASSERT(lFieldID != -1);

      while (lFieldID != -1)
      {
         // Fetch the persistent IO flag.
         lIOFlag = m_pClassObj->GetIOFlag(lFieldID);

         // Check for Param field.
         if (lIOFlag & piofParam)
         {
            // Create a variant to hold the param's data value.
            pva = new VARIANT;
            VariantInit(pva);
            m_ParamMap[lFieldID] = pva;   // Place the variant in the map.
         }

         // Fetch the next field ID.
         lFieldID = m_pClassObj->EnumNextFieldID();
      }
   }
```

To begin with, I enter a **while** loop, traversing through all of the fields of the associated Class Object. Next I check their IO flag for the existence of **piofParam**. If this bit is present in the flag, I allocate a **VARIANT** to hold the value. This value will later be initialized with a call to the Data Object's **SetParamValue()** function. This function receives two parameters: the field ID and the parameter's value as a **VARIANT**. The Data Object can now assign these values to the parameter fields of the Recordset it builds to load its data:

```
RecordsetObj.SetParamValue(OLE2T(bstrColumnName), *pvarg);
```

While we're on the subject of the recordsets needed to load the data for a Data Object, let me remind you that a single Class Object might contain fields from different tables. To make matters even more complicated, the fields might just return one row or several rows of data. This means that a single Data Object should be prepared to maintain data for different fields from different tables, with different row counts (repeat this sentence three times really fast. I bet you can't!). It took me a while to figure out a clean solution for this one.

The first thing I needed was to separate the fields into tables. In other words, I needed some way of creating a Recordset for each table name. If I could accomplish this, I could then keep all of the fields that belong to one table in a single recordset. I created a map collection to map table names to a new structure, called **RECORDSETINFO**, which looks like this:

```
typedef struct tagRECORDSETINFO
{
  CWhRecordset RecordsetObj; // Constructed Recordset.
  CLongList ColumnList; // Chain of fields in recordset.
} RECORDSETINFO;
```

The map is named **m_RSMap**, simply because it maintains a map of Recordsets. This structure contains the Recordset for the table, as well as pointers to the fields maintained by the Recordset (in **ColumnList**). Once I create my map and my Recordset structure, I only need to traverse through the Class Object's fields, determine what table they belong in and add them to the appropriate recordset. Here's the code from **CWhDataObject::CreateRecordsets()** that achieves this:

```
while (lFieldID != -1)
{
  .
  . Fetch all the necessary field properties.
  .
  // Retrieve or create the necessary table.
  if (lIOFlag & piofColumn || lIOFlag & piofParam)
  {
    if (m_RSMap.Lookup(OLE2T(bstrTableName), pRS) == FALSE)
    {
      pRS   = new RECORDSETINFO;
      .
      . Create new recordset, assign it to RecordsetObj
      .
      // Add the new RECORDSETINFO to the map
      // using the table name as the key.
      m_RSMap[OLE2T(bstrTableName)] = pRS;
    }
  }

  // Add Columns to the recordset.
  if (lIOFlag & piofColumn)
  {
    pRS->ColumnList.AddTail(lFieldID);
    pRS->pRecordsetObj->AddCol(lODBCDataType, OLE2T(bstrColumnName));
    .
    . Assign other properties such as table and
    . sort column.
    .
    // Add Params to the recordset.
    if (lIOFlag & piofParam)
    {
      pRS->pRecordsetObj->AddParam(lODBCDataType, OLE2T(bstrColumnName));
      .
      . Assign other properties such as table name,
      . operator type, parameter value, join fields.
    }
}
```

When you're adding the fields to the Recordset, you must determine whether the field is a column field or a parameter field, because you'll need to call different functions on the Recordset Object, depending on the type of field. Note that the `RECORDSETINFO` structure is only created once per table. The structure is then added to the map, using the table name as the key. When the Data Object actually loads its data, it will need to tell each Recordset to fetch its data. This is done with the following code from the exposed Data Object's automation method, named `Load()`:

```
BOOL CWhDataObject::Load(LPDISPATCH pDbObj)
{
   if (m_bLoaded)
      return FALSE;

   pDbObj->AddRef();
   m_DbObj.AttachDispatch(pDbObj);

   VERIFY(CreateRecordsets());

   // Load the data.
   POSITION pos;
   RECORDSETINFO* pRS;
   CString strKey;

   for (pos = m_RSMap.GetStartPosition(); pos != NULL; /* NO LOOP-EXPR */)
   {
      m_RSMap.GetNextAssoc(pos, strKey, pRS);
      VERIFY(pRS->RecordsetObj.Fetch());
   }
```

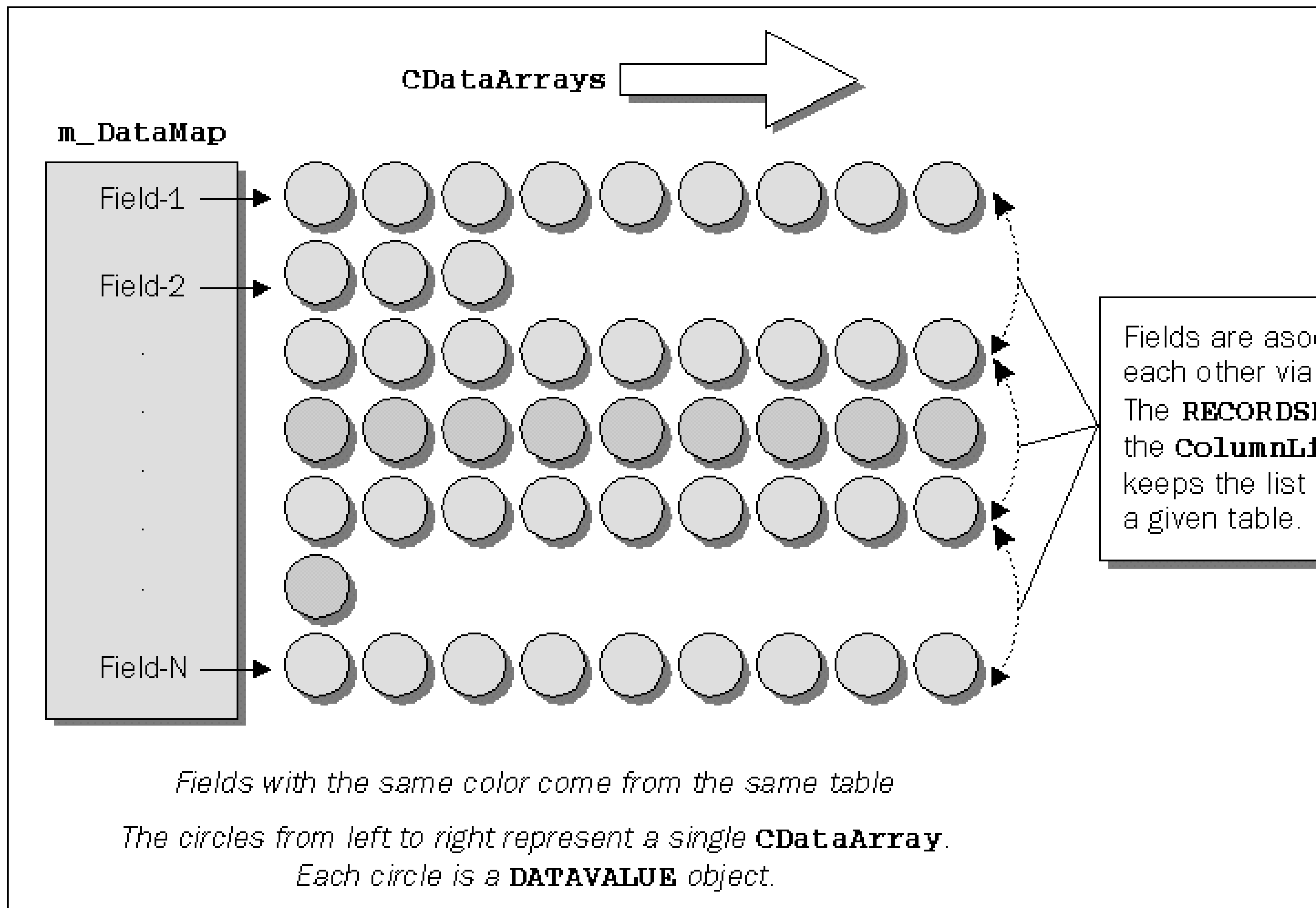
```
   VERIFY(LoadData());

   return m_bLoaded = TRUE;
}
```

`Load()` iterates through the Recordsets, calling each one's `Fetch()` method. Once the data has been loaded into the Recordsets, the Data Object copies the values held by the Recordsets into memory allocated by the Data Object. That way, it can destroy the Recordsets without destroying the data returned by them. The copying of the data is performed by the `CWhDataObject::LoadData()` function like this:

```
BOOL CWhDataObject::LoadData()
{
  USES_CONVERSION;
  // Define variables for fields.
  Long lFieldID = m_pClassObj->EnumFirstFieldID();
  BSTR bstrTableName;
  BSTR bstrColumnName;
  long lIOFlag;
  RECORDSETINFO* pRS;
  CDataArray* pArray;
  DATAVALUE* pDataValue;

  while(lFieldID != -1)
  {
    // Fetch all the necessary field properties.
    lIOFlag = m_pClassObj->GetIOFlag(lFieldID);

    // Retrieve the necessary table.
    if (lIOFlag & piofColumn)
    {
      bstrTableName = m_pClassObj->GetTableName(lFieldID);
      bstrColumnName = m_pClassObj->GetColumnName(lFieldID);

      m_RSMap.Lookup(OLE2T(bstrTableName), pRS);

      // Add an array of data values if necessary.
      if (m_DataMap.Lookup(lFieldID, pArray) == FALSE)
      {
        m_DataMap[lFieldID] = new CDataArray;
      }

      // Loop through recordset and load data.
      for(pRS->RecordsetObj.MoveFirst();
          pRS->RecordsetObj.IsEOF() != TRUE;
          pRS->RecordsetObj.MoveNext())
      {
        // Create new DataValue item for adding to array.
        pDataValue = new DATAVALUE;
        pDataValue->lModType = dmfUnchanged;
        pDataValue->varValue = pRS->RecordsetObj.GetColValue(
                                          OLE2T(bstrColumnName));

        // Store the data in the map.
        m_DataMap[lFieldID]->Add(pDataValue);
      }
    }

    // Release the system strings.
    SysFreeString(bstrTableName);
    SysFreeString(bstrColumnName);

    // Get the next field.
    lFieldID = m_pClassObj->EnumNextFieldID();
  }

  return TRUE;
}
```

This function begins by iterating through the fields. As it reaches a new field, it checks the field's IO flag to determine whether or not the field is a column field. If it's a column field, the field's table name is used to locate the **RECORDSETINFO** structure in the map **m_RSMap**. Once the appropriate Recordset has been located, another map, **m_DataMap** (a collection that maps field IDs to **CDataArray** objects), is searched. The reason I needed an array for the field is because, as I mentioned before, a field (or column) can contain one, or several rows, so it's safer to go with an array for the field's values.

Finally, I traverse through the Recordset, pulling out all of the rows' values for the current field. The value is then stored in a structure, named **DATAVALUE**, and the structure is stored in the **CDataArray**. From top to bottom, each field ID maps to a **CDataArray**, which maintains all of the values (row by row) for a single field. In case you're confused, let me try to clear things up with a diagram:



The **lModType** member of the **DATAVALUE** structure determines whether the value has been modified by the application. This is later used to determine which fields need to be updated to the database. This is normally done with a call to the Data Object's **Save()** function, which the application needs to call when it wants to update the database with any changes that might have been made to a Data Object.

I'll spare you the code for the **Save()** function, since it simply pulls out the values, looks up the

appropriate Recordsets, calls each Recordset Object's **`Edit()`** function or **`AddNew()`**, sets the new values for the columns and calls the **`Update()`** function of the Recordset Objects to complete the updates.

In case you were wondering how the application can determine how many rows of data a particular field was loaded with, it can call the Data Object's **`GetRowCount()`** function.

The Final Verdict

So there you have it folks. I've explained both concepts and code, so now it's up to you to absorb the information and come up with some applications to use the warehouse. I'm sure you'll waste no time in turning your database applications into applications that can practically maintain themselves.

Although I've tried to provide as much detail about the principles and the implementation as possible, I just don't have space to answer all of the questions you might have about how I implemented every feature. To get the most from the data warehouse, I recommend that you spend some time investigating the code (which has been conveniently provided on the accompanying CD-ROM). But just so that I don't leave you out in the cold without showing you some sort of client for the warehouse, I've also provided a small sample application, which I'll describe in the next section.

A Sample to Break in the House

You can find the client application on the CD in the **`Warehouse\Testbed`** directory. It uses the `Warehouse.mdb` database which must be registered with a data source name of 'Warehouse'.

In this dialog-based application, I made use of the type libraries provided by the transient and component layer servers to create OLE Automation wrapper classes using ClassWizard. I provided member variables instantiated from these classes in my client application's main window class as follows:

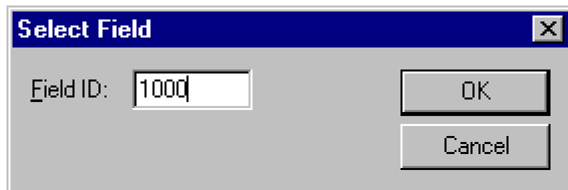
```
class CTestBedDlg : public CDialog
{
// Construction
public:
   CTestBedDlg(CWnd* pParent = NULL);   // standard constructor

// Attributes
   CWhDatabase m_DbObject;
   CWhClassContainer m_ClassContainer;
   CWhDataObject m_DataObject;
   CWhClassObject m_ClassObject;
.
. more code
.
}
```

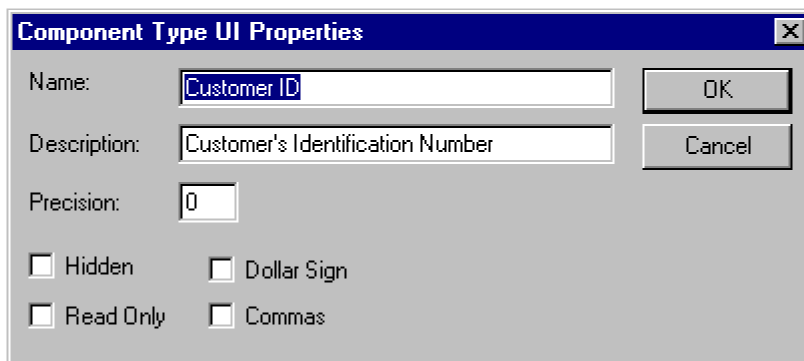
The client application that we'll meet in this section has a main screen that looks like this:

The fields that you see on the screen, **`CustomerID`**, **`FirstName`**, and **`LastName`**, and the values associated

with them, were generated on the fly, based on the information that came back for the Class Object and the Data Object. We'll shortly see how I picked the information from the objects.

The application also gives us the ability to display information about the fields if we know the field identifier. For example, if the `CustomerID` field has an identifier of `1000`, we could display the field's user interface properties by selecting the UI Props... button and entering the value `1000`.





All of this information is gathered at run time. When you look at the code, you'll find that there's no mention of `CustomerID` or any other field.

Creating a Database Object

Before you can use a Container Object, your application should first create a Database Object. I do this in `CTestBedDlg::OnInitDialog()`:

```
// Create and initialize the Database Object.
m_DbObject.CreateDispatch("Warehouse.Database");
m_DbObject.Connect("Warehouse");
```

The first line calls the `COleDispatchDriver::CreateDispatch()` function to load the automation server into memory and create an instance of the automation class associated with the program identifier of `"Warehouse.Database"`. Once the object has been created successfully, MFC returns and attaches the `IDispatch` returned from OLE to the `m_DbObject`.

The next function, `IWhDatabase::Connect()`, calls the `Connect()` method of the Database class in the automation server. The function requires that you pass it a valid ODBC data source name. It will attempt to open the data source and connect to it, getting back an ODBC connection handle. The handle is stored in the automation server for safe keeping.

Creating a Container Object

Once a Database Object has been created and successfully connected to a data source, you should construct your Container Object. You'll need this object in order to request Data Objects or Class Objects from the data warehouse.

I created my Container Object using the following code:

```
// Create and initialize the Container Object.
m_ClassContainer.CreateDispatch("Warehouse.ClassContainer");
m_ClassContainer.Initialize(m_DbObject.m_lpDispatch);
```

Again, the first line calls the **`COleDispatchDriver::CreateDispatch()`** function to create an instance of the automation class associated with the program identifier of **`"Warehouse.ClassContainer"`**. Once the object has been created successfully, MFC returns and attaches the **`IDispatch`** returned from OLE to the **`m_ClassContainer`** data member.

The second line initializes the Class Container Object by passing the dispatch interface of the Database Object to the Class Container Object. The Class Container Object holds on to the Database Object's **`IDispatch`** pointer so that it can also create Recordset Objects or load data from the data source as it needs.

Creating a Data Object

When an application wishes to create a Data Object, it must call **`m_ClassContainer.CreateComponent()`**. The Container Object is passed the class ID of the Data Object to create. This creates several objects such as the Class Object and the Data Object itself, then attaches them to each other. Several Recordset Objects are created for loading the appropriate fields, events and methods for the Class Object.

In the client code, I do this in the **`OnInitDialog()`** function of the main window:

```
// Create the Data Object.
m_DataObject.AttachDispatch(m_ClassContainer.CreateComponent(100, 1));
```

Notice that I passed the value returned from the Container Object to the **`COleDispatchDriver::AttachDispatch()`** function. Since Data Objects are not registered in the registry, the only way to create them is via the Container Object.

There are two parameters that are passed to the **`CreateComponent()`** method. The first is the identifier for the type of object you wish to create. As I said before, whoever creates an object in your database must document the object's identifier and the fields' identifier and must publish them to the rest of your team.

The second parameter passed to **`CreateDataObject()`** is a unique identifier which will be used to identify the component that you're creating. Internally, the automation server combines the Class Object's identifier with the Data Object's identifier (the key you pass) and creates a hashed value. Here's the code used by the server to create this key once again:

```
// Build DataObject key using ClassID + DataObject's Key Value.
#define   WH_KEY_PROLOGUE(lClassID, lKey)         \
   CString strDataObjKey;                     \
   strDataObjKey.Format(_T("%ld-%ld"), lClassID, lKey);
```

If the server needs to access or return this key for any reason, it simply needs to call this macro and pass it the Class Object's identifier and the Data Object's identifier. As a matter of fact, when you call the Data Object class's **GetKey()** function, the key is first constructed using this technique and then the composed key is returned.

Once you have created both the Class Object and the Data Object, it's time to load the data into the Data Object. We do this by simply calling the **Load()** method of the Data Object class. Here you can see an extract from the client application that demonstrates the use of **Load()**:

```
m_DataObject.Load(m_DbObject.m_lpDispatch);
```

Depending on how the object's class information was set up in the database, you might need to set some parameters in order to load the object. You'll know if you need to do this based on the object's documentation. For example, let's say that the object was created so that you must supply a **CustomerID** before it's loaded. Furthermore, let's say that the field identifier for the **CustomerID** field is 1000. You would need to call the **SetParamValue()** of the Data Object before calling the **Load()** method. Here's some code to perform this:

```
COleVariant va(1);
m_DataObject.SetParamValue(1000, va);
```

This code would result in a SQL statement similar to the following:

```
SELECT * FROM Customers WHERE CustomerID = 1
```

If the **Load()** function is successful, you can begin to retrieve values from the Data Object. Again, you would need to know the identifiers of the fields to pull out the data. The exception to this rule is if you simply enumerate through the fields by using the Class Object's **EnumFirstFieldID()** and **EnumNextFieldID()** functions. Next, you would need to pass the identifier to the **GetData()** function, as specified by the following code:

```
m_strValue1 = vtConvertVariantToString(m_DataObject.GetData(nFieldID, 0));
```

In the above code, I tell the Data Object that I want to retrieve the value stored at **nFieldID**, row 0 (in case there are several rows of data). Since the values come as a **VARIANT**, I created a function to convert it from a **VARIANT** to a **CString**, so that I could store it directly into a string variable.

Summary

So there you have it folks. Again, you won't get the whole picture until you study the code. Unfortunately, since the warehouse is a major piece of code, I couldn't print it all here, but I would suggest that you take a look at the source code and study it, using this chapter as a reference.

With every solution I have ever seen (no matter how good), there is always room for improvement, so if you do come up with some good ideas, send me an e-mail. I'd love to hear about them.

Enhancing Windows UI Components

Overview

Windows has always provided the programmer with a good variety of controls and dialogs, and with Windows 95 this choice has broadened considerably (probably putting several small custom control manufacturing companies out of business in the process!).

Along with the traditional buttons, edit controls and list boxes, we now have image lists, masked edit controls, property sheets, tree lists and a host of other, sometimes weird and wonderful, controls and dialogs. It's always the case, though, that you can find something you need to do which is outside the scope of the controls you've got, which means that you need to go hacking. When this happens, there are two things you can do: either write yourself a new control from scratch, or modify the behavior of an existing control.

Obviously, if the control you need is completely unlike anything currently available under Windows, such as, say, a stop/go signal or a graphing control, you'll need to write it yourself. Actually writing such a control using MFC and implementing it as an OCX isn't that hard, but it *is* outside the scope of this chapter.

But what if an existing control nearly fits what you need? Say, an edit control which only accepts numbers, or a status bar with a bitmap in one of its panes? In these, and similar cases, you can hack into the existing controls, modifying their behavior and making them do what you want them to do. It's exciting stuff, and will certainly teach you a lot more about how Windows (and MFC) works, as well as giving you some neat user-interface widgets.

This chapter shows how to go about modifying and extending the controls and dialogs provided with Windows 95 (and NT 3.51) through a series of case studies, rather than pure theory or abstract ideas. I've done this in the hope that you may find something that's actually of use to you in your own applications, and to give you some ideas which you can extend and modify in your own time.

If you do invent (or come across) some great ideas for modifying the standard controls or dialogs, let me know!

Getting Started

Before we start on the coding, let's get some of the more theoretical stuff over and done with.

Although in some cases we're going to be altering the appearance and function of a control or dialog, most of the work in this chapter is accomplished by **subclassing**.

What is subclassing? Let's begin by considering it in the context of a traditional SDK-style Windows program. (Before anyone complains, I'm well aware that there may be some people who have never seen an old-style Windows program, but don't worry, we aren't going too far back in time…)

In a Windows program, every window has a function associated with it, called the **window procedure**,

which processes the messages for all windows of that type. This means that for, say, edit controls, there's a function down in the bowels of Windows whose job it is to process all the messages passed to edit controls. Every window 'object' has stored within it a pointer to this function, which is how Windows knows where to route messages.

We can't get at this window procedure, because Microsoft doesn't hand out the source code, but we do have a pointer to it inside our edit control. This means that we can replace that pointer with one to a new custom window procedure, which does what we want. In other words, we hook into the control's message processing.

Our new procedure may do one of four things with each message it receives. It may:

Do nothing at all.
Pass it to the original procedure.
Do something without passing it on.
Do something and then pass it on.

Most messages will get passed on unmodified, like they always do. Some, we may wish to block, as when we examine **WM_CHAR** messages to stop an edit control accepting digits. In other cases, we might want to take some action before passing the message on; when the user resizes our application's main window, we might want to move or resize a control within the window before passing the message on so that Windows can resize the frame.

Before we go on to see how this is done using MFC, I'll just mention that there's also a technique, called **superclassing**, which is similar to subclassing, only different (if you see what I mean). Both involve altering the window procedure of a control, but whereas subclassing involves changing the procedure in a single instance of a control, superclassing involves creating a new procedure which affects an entire class of controls, such as all the edit controls.

How to Subclass a Control Using MFC

As you might expect, subclassing using MFC bears little outward resemblance to the procedure I've just described, because all the low-level Windows work is done for you by MFC itself.

The idea is simple. MFC provides classes to wrap all the control types provided by Windows, from **CButton** to **CStatusBar** and beyond, and the member functions of each of these classes together make up the 'window procedure' which processes the messages for the control. We subclass controls by deriving new classes from those in MFC and overriding member functions in order to customize their functionality. As in the example above, we are at liberty not to override certain functions, so that the message gets processed by the base class, or we can provide our own functions and either process messages ourselves, or partially process them and pass them on.

Supposing that I do subclass a control, say a **CButton**, by providing my own class **CMyButton**. How do I use this in my MFC program?

There are basically two ways. The first is to use ClassWizard, and the second is to use the **CWnd** function, **SubclassDlgItem().**

When you place a control in a dialog or a **CFormView**, ClassWizard allows you to associate a variable with it, using the Member Variables tab in the ClassWizard dialog. This may be a simple variable, such as an

integer or **CString**, but it can also be a 'control', in which case you specify a control class to be associated with the actual control on the screen.

ClassWizard then creates an instance of your control class and associates it with the control, so that all messages for the control will get routed to your control object, rather than the original. We'll see how this works in practice later in the chapter, when we subclass an edit control.

The second way is to add a handler to your dialog to process the **WM_INITDIALOG** message and in that handler, add calls to **SubclassDlgItem()** in order to associate control IDs with control classes. It has the same effect as using ClassWizard, but is more, how shall I put it… manual.

Playing with Menus

We'll start by looking at a couple of ways in which we can modify menus.

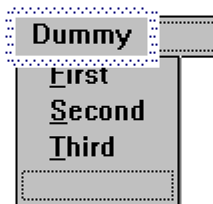
How to Create a Right-button Pop-up

Many applications are now making use of the right mouse button pop-up menu; it can be used in most places in Developer Studio to give shortcut access to commonly used commands, which are relevant to the part of the screen on which you click.

If you're using Visual C++ 4.0 or above, you can add this support directly from the Component Gallery. If you're interested in understanding how it works, read on.

To provide a right mouse menu for a window, you could install a handler for the **WM_RBUTTONDOWN** or **WM_RBUTTONUP** messages, but for Windows 95 and NT 3.51, it's better to use the new **WM_CONTEXTMENU** message. This has been specially provided for right mouse button menus, and will work in situations where the **WM_RBUTTONDOWN** and **WM_RBUTTONUP** messages aren't generated, such as clicking over controls in a dialog.

Start by defining a menu resource using the menu editor in Developer Studio and give it a suitable ID, such as **IDR_RT_POPUP**. The item at the top of the pop-up can be anything; it only serves as a place marker and you'll never actually see it. Here's a pop-up menu with three items:



Now code up the context menu handler. We need to load the pop-up menu from the app's resources and get a pointer to its first submenu. To avoid loading the menu from the resource each time the user presses the right mouse button, we can load it once when the application starts and save it in one of our application's classes. It doesn't really matter which one, but I've chosen to add a **CMenu** data member to the application class, and load it with the menu resource in **InitInstance()**.

```
class CRtMenuApp : public CWinApp
```

```
{
public:
   CRtMenuApp();

// ...
public:
   CMenu m_Popup;
};

BOOL CRtMenuApp::InitInstance()
{
   // ... All the other initialization

   m_Popup.LoadMenu(IDR_RT_POPUP);
}
```

In the handler, we get the submenu associated with the pop-up as a pointer to a **`CMenu`**. Instead of finding it each time, why not store this directly in the application class? The answer is that **`CMenu`** operations, like **`GetSubMenu()`**, return pointers which are not designed to be stored, and should be regenerated each time they are to be used.

```
void CRtMenuView::OnContextMenu(CWnd* pWnd, CPoint point)
{
   CRtMenuApp* pApp = (CRtMenuApp*)AfxGetApp();

   CMenu* pSubMenu;
   pSubMenu = pApp->m_Popup.GetSubMenu(0);

   pSubMenu->TrackPopupMenu(TPM_LEFTALIGN | TPM_RIGHTBUTTON,
             point.x, point.y, AfxGetMainWnd(), NULL);
}
```

Now that we've got the submenu, we use its **`TrackPopupMenu()`** member function to display the pop-up and handle its messages, and we can add handlers for these menu items just as you would for any other.

Using Dialogs

Using **`WM_CONTEXTMENU`** rather than the right button messages really comes into its own in forms and dialogs. In this situation, the parent won't get a button message when the user right-clicks over a control, but it will get a **`WM_CONTEXTMENU`**. The message handler gets passed a pointer to the window where the event occurred, and you can use this to find out what the user clicked on:

```
void CTestDlg::OnContextMenu(CWnd* pWnd, CPoint point)
{
   if (this == (CTestDlg*)pWnd)
      AfxMessageBox("Clicked on dialog");
   else if (pWnd == GetDlgItem(IDCANCEL))
      AfxMessageBox("Clicked on cancel button");
}
```

One final point—you'll notice that you get a standard menu when you right-click over a standard edit control. If you want to override this to provide your own menu, you can subclass the **`CEdit`** class and provide a handler for the **`WM_CONTEXTMENU`** message. The section later in this chapter on how to write a masked edit control shows you how to subclass **`CEdit`**.

Modifying the System Menu

Let's turn our attention now to modifying the system menu. Why would you want to do this? One

particular use is in applications which run in a minimized state, where the only access to any menu items is their system menu; in these cases, you may well want to add commands to this menu.

It's not at all difficult to modify the system menu. The **CWnd** class has a member function **GetSystemMenu()**, which returns you the system menu in the form of a pointer to a **CMenu** object.

Once you've got that, you can use the standard **CMenu** member functions to modify the menu. In the following code fragment, we've added About... to the bottom of the system menu and given it the same ID as the About... item on the standard menu:

```
BOOL CMyApp::InitInstance()
{
   // Stuff omitted

   // Get handle to the system menu of the app's main window
   CMenu* pSysMenu;
   pSysMenu = m_pMainWnd->GetSystemMenu(FALSE);
   // Add a separator and an 'About' item
   pSysMenu->AppendMenu(MF_SEPARATOR);
   pSysMenu->AppendMenu(MF_STRING, ID_APP_ABOUT, "About...");
}
```

We still need to handle the selection of the menu, though. When the system menu is selected, Windows generates a **WM_SYSCOMMAND** message, but ClassWizard won't add a handler; you have to code one yourself. The best place to put this is in the **CMainFrame**. The class definition needs a member prototype:

```
class CMainFrame : public CFrameWnd
{
...
// Generated message map functions
protected:
   //{{AFX_MSG(CMainFrame)
   afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
   afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
      // NOTE - the ClassWizard will add and remove member functions here.
      //    DO NOT EDIT what you see in these blocks of generated code!
   //}}AFX_MSG
   DECLARE_MESSAGE_MAP()
};
```

The message map needs a new entry:

```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
   //{{AFX_MSG_MAP(CMainFrame)
      // NOTE - the ClassWizard will add and remove mapping macros here.
      //    DO NOT EDIT what you see in these blocks of generated code !
   ON_WM_CREATE()
   ON_WM_SYSCOMMAND()
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

And finally, the handler itself:

```
void CMainFrame::OnSysCommand(UINT nID, LPARAM lParam)
{
   if ((nID & 0xFFF0) == ID_APP_ABOUT)
   {
      ((CMyApp*)AfxGetApp())->OnAppAbout();
   }
   else
      CWnd::OnSysCommand(nID, lParam);
}
```

The **nID** argument contains the ID of the menu option selected, along with other information in the lowest four bits, hence the need to AND it with **0xFFF0**. You must remember to call the base class implementation of **OnSysCommand()**, otherwise your application will stop working properly (just try it if you don't believe me!).

If you need to modify the system menu from elsewhere in the program, you can use **AfxGetApp()->m_pMainWnd** or **AfxGetMainWnd()** to get to the app's main window.

Notice the **FALSE** parameter to **GetSystemMenu()**. If called with **FALSE**, the function returns you a pointer to the menu so that you can modify it. If called with **TRUE**, on the other hand, it returns the system menu to its default state.

Looking at Toolbars

In the next two sections, we're going to look at how to modify two of the most popular of the new Windows 95 common controls: toolbars and status bars.

What are Toolbars?

A toolbar is basically a window which contains an array of buttons that are mostly used as shortcuts for common menu items. The standard toolbars can either be free-floating or 'docked' to the edge of the window.

Prior to the release of the common controls, anyone wanting to use toolbars or status bars had to either roll their own, or buy a third-party control. Neither of these options is particularly good from the point of view of portability and maintainability, so it's good to have 'standard' versions at last.

Two for the Price of One

MFC provides two classes that work with toolbars: **CToolBar** and **CToolBarCtrl**. Why two separate classes? This is because versions of MFC prior to the release of the common controls provided a toolbar class called **CToolBar**. With Visual C++ 4.x, MFC provides a new class, **CToolBarCtrl**, that provides a thin wrapper over the Windows common control and **CToolBar** has had its implementation changed so that it too uses the new common control.

CToolBarCtrl, directly descended from **CWnd**, provides almost direct access to all the functions of the common control, whereas **CToolBar**, which inherits from **CControlBar**, is more fully integrated with the rest of MFC.

We'll be concentrating mainly on using **CToolBar** in the context of MFC, but if you need to talk directly to the underlying common control, you can call the **CToolBar::GetToolBarCtrl()** member function to get a reference to the actual control.

Control Bars

Status bars and toolbars are specialized types of control bar, represented in MFC by the **CControlBar** class. Control bars are windows that may contain either standard, **HWND**-derived controls or non-**HWND**

items. They are usually owned by the frame window in which they appear.

The **CControlBar** class contains functionality for aligning the bar to the top, bottom or side of the frame, for allocating arrays of control items, and for helping you to derive new classes.

Adding a Combo Box to a Toolbar

A toolbar basically consists of buttons and separators, but you can add other sorts of objects. Let's look at how to add a combo box to the beginning of the standard MFC toolbar, the one which AppWizard adds to the skeleton project.

Create a Project

Create a new project using AppWizard; either an MDI or SDI example will do. Make sure you select the Docking toolbar option in Step 4.

Modifying the Toolbar Creation Code

In the frame window class' source code, in `Mainfrm.cpp`, separate out the code which creates the toolbar from **OnCreate()**, and put it into its own function, **CreateToolBar()**.We aren't going to be calling it from anywhere else, so make it **private** to the **CMainFrame** class.

```
BOOL CMainFrame::CreateToolBar()
{
   if (!m_wndToolBar.Create(this) ||
      !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
   {
      TRACE0("Failed to create toolbar\n");
      return FALSE;       // fail to create
   }

   return TRUE;
}
```

You can then call this function from **OnCreate()**, to create the toolbar:

```
   if (!CreateToolBar())
      return -1;
```

Formatting the Toolbar

We need a placeholder for the drop-down list at the start of the toolbar, and a separator between the list and the next group of buttons, but the toolbar editor doesn't allow you to put spaces in at the start of the toolbar. We could construct our own array of button IDs, putting two **ID_SEPARATOR** IDs (one for the combo and one for the space after it) at the start, and call **SetButtons()**, but that gives maintenance problems. Pity the poor programmer who adds a new button using Developer Studio and then can't find out why it doesn't work because they've overridden the IDs in the code. A neater solution would be to use the toolbar IDs provided and dynamically add the two separators.

How would we go about this? There doesn't seem to be any method in **CToolBar** to add just one button. Well, the **CToolBar** class is a wrapper for the system supplied toolbars, implemented in **CToolBarCtrl**.

You can get a pointer to this class using **CToolBar::GetToolBarCtrl()**, and **CToolBarCtrl** has a method called **InsertButton()**, which will do what we want.

Once the toolbar has been created, we call **InsertButton()** twice to add the separators. First, we have to set up a structure for defining the button:

```
BOOL CMainFrame::CreateToolBar()
{
   if (!m_wndToolBar.Create(this) ||
      !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
   {
      TRACE0("Failed to create toolbar\n");
      return FALSE;      // fail to create
   }

   TBBUTTON   button;       // prepare to call InsertButton

   button.iBitmap = NULL;    // set up the structure
   button.idCommand = 0;
   button.fsState = 0;
   button.fsStyle = TBSTYLE_SEP;  // a separator
   button.dwData = 0;
   button.iString = NULL;
   m_wndToolBar.GetToolBarCtrl().InsertButton(0, &button);
   m_wndToolBar.GetToolBarCtrl().InsertButton(0, &button);

   return TRUE;
}
```

Test the Program

If you build and run the program at this point, you'll see that the toolbar has a blank space at the start. We'll now go on to fill it with a combo box.

Creating the Combo Box

We need to provide an ID by which we can refer to the combo box. Creating a string resource will enable us to both label the combo box and provide a status bar prompt and tool tip. I called it **IDS_COMBO**, and gave it a value of **"Combo box\nCombo"**.

The toolbar's **SetButtonInfo()** member function is used to set various items of data associated with buttons on the toolbar. Note that, although the function is associated with buttons, it also works with separators; maybe **SetItemInfo()** would have been a better name!

In this case, we're interested in two properties in particular: the command ID associated with the button and the width of the button.

Button 0 represents the separator where we're going to display the combo box, so we set it to have the ID of our combo string resource and a width of 100 pixels. Button 1 stays as a separator, with a width of 12:

```
BOOL CMainFrame::CreateToolBar()
{
   ...
   m_wndToolBar.SetButtonInfo(0, IDS_COMBO, TBBS_SEPARATOR, 100);
   return TRUE;
}
```

Note that we leave the style as **TBBS_SEPARATOR** for both items, because, as far as the toolbar itself is concerned, they are both areas which the toolbar doesn't have to draw.

Now that we've got enough room on the toolbar, we actually need to create the combo box and display it. We need an MFC combo box object to do this, so add a combo box member to the frame window class:

```
CComboBox m_toolBarCombo;
```

To draw the box, we find the size of the space in the toolbar and extend it downwards to allow for the drop of the combo box, say, 100 pixels:

```
BOOL CMainFrame::CreateToolBar()
{
   ...

   CRect rect;
   m_wndToolBar.GetItemRect(0, &rect);
   rect.top = 3;
   rect.bottom = rect.top + 100;
```

Once we've got the dimensions, we can create the combo box as a child of the toolbar:

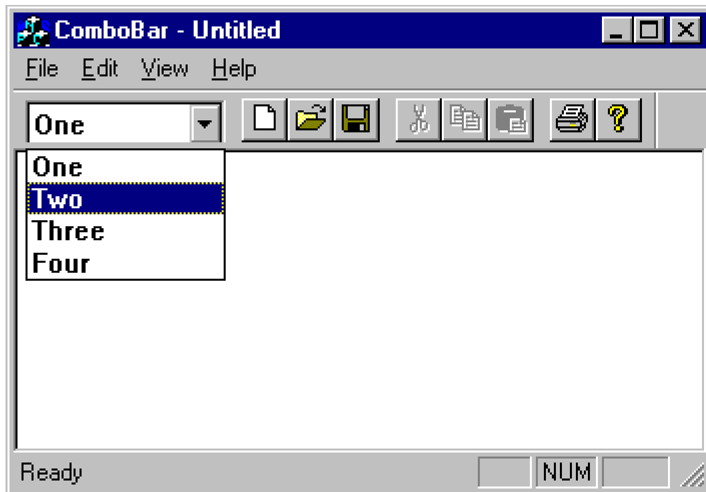
```
   if (!m_toolBarCombo.Create(CBS_DROPDOWNLIST | WS_VISIBLE |
      WS_TABSTOP, rect, &m_wndToolBar, IDS_COMBO))
   {
   TRACE0("Failed to create combo-box\n");
   return FALSE;
   }
```

After that, it's simple to fill the combo box with strings:

```
   m_toolBarCombo.AddString("One");
   m_toolBarCombo.AddString("Two");
   m_toolBarCombo.AddString("Three");
   m_toolBarCombo.AddString("Four");

   return TRUE;
}
```

Testing the Program

When you build and test the program, you should find a working combo box on your toolbar, like this:



Handling Notifications

Now that we've got the combo box there, how do we use it? The easiest thing to do is to add a suitable message handler.

Say we want to intercept each time the user changes the combo selection. We can do this by adding a handler for the combo's **CBN_SELCHANGE** notification message. Because we've added the combo box manually, we'll also add the message map entries manually.

First, add a suitable function to the frame class declaration:

```
//{{AFX_MSG(CMainFrame)
afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
   // NOTE - the ClassWizard will add and remove member functions here.
   //    DO NOT EDIT what you see in these blocks of generated code!
//}}AFX_MSG

// Handler for combo box messages
afx_msg void OnComboChange();
DECLARE_MESSAGE_MAP()
```

Then add the entry to the message map in the frame class' source file:

```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
   //{{AFX_MSG_MAP(CMainFrame)
   // NOTE - the ClassWizard will add and remove mapping macros here.
   // DO NOT EDIT what you see in these blocks of generated code!
   ON_WM_CREATE()
   //}}AFX_MSG_MAP

   ON_CBN_SELCHANGE(IDS_COMBO, OnComboChange)
END_MESSAGE_MAP()
```

Once again, don't put it in ClassWizard's section. The handler for the **CBN_SELCHANGE** message is, unsurprisingly, **ON_CBN_SELCHANGE**; MFC defines a host of handlers for all manner of control notification messages. See the online help for a full list.

The last thing is to implement the handler itself, which isn't too hard, as the routine takes no arguments. So you can see how to do it, here's some simple code which gets the selection and uses the **TRACE** macros to output to the debugger window:

```
void CMainFrame::OnComboChange()
{
   int nSel = m_toolBarCombo.GetCurSel();
   if (nSel == CB_ERR)
      TRACE0("Combo: no selection\n");
   else
   {
      CString cs;
      m_toolBarCombo.GetLBText(nSel, cs);
      TRACE1("Combo: selection is now '%s'\n", (LPCTSTR)cs);
   }
}
```

Run the program under the debugger, and you'll see a message being printed each time the selection changes.

You can use this method to insert any control you wish into the toolbar (including OCX's if you turn on OCX containership in AppWizard), although you do have to be careful about the height of the control!

A Word about Status Bars

A **status bar** is a horizontal window that is usually displayed at the bottom of a parent window and is used by the application to display status information.

Status bars may contain a number of separate display areas, called **panes** or **indicators**. Panes pack to the right of the window, so the first one, pane 0, expands to take up all the space on the left of the window which is unused by the other panes.

Pane 0 is usually used for displaying text messages, such as menu item prompts, while the other panes are used for status information, such as key states (*CapsLock*, *NumLock* and so on).

Version 2 Status Bars

This book is about programming using Visual C++ version 4.x, but it's worth noticing that there are differences between the way status bars are handled in 4.x and version 2.x.

The reason for this is that, in version 2.x, status bars were an MFC invention, whereas in version 4.x, designed as it is to run under Windows 95 and NT, the status bar classes use the built-in status bar common control. This means that we have the same situation as with toolbars, with two classes for status bars: `CStatusBar` and `CStatusBarCtrl`.

What this means in practice is that some of the techniques that could be used to customize status bars under 2.x no longer work under 4.x. If you really do want to emulate a 2.x status bar, there is some sample code included with the Visual C++ distribution, which you'll find in `\Msdev\Samples\Mfc\Advanced\Oldbars`.

Putting a Bitmap in a Status Bar Pane

Often, it would be neat to be able to put a bitmap in one of the panes in the status bar, such as a logo, or maybe a stop/go light. The problem is that while it's quite easy to modify the text in a status bar pane

(using **SetWindowText()** to update the text in pane 0, and **SetPaneText()** for any other pane), no functions are provided for adding or manipulating graphics. However, you're now given the option of making panes owner-draw, so it becomes fairly easy to draw a bitmap yourself in a pane.

Create a Project

The first thing to do is to create a project. For the purposes of this exercise, you can use an SDI or MDI project, but just make sure you tell AppWizard to add a status bar!

Add a New Pane to the Status Bar

We'll add a new pane to the status bar to hold the bitmap. The number and size of the panes is determined by the framework when it creates the status bar, and MFC uses the information it finds in the static array **indicators[]** in the `Mainfrm.cpp` file, which you'll find just below the message map:

```
static UINT indicators[] =
{
   ID_SEPARATOR,           // status line indicator
   ID_INDICATOR_CAPS,
   ID_INDICATOR_NUM,
   ID_INDICATOR_SCRL,
};
```

The number of items determines the number of panes. Each item is the ID of a string resource, with the length of the associated string determining the size of the pane. Panes are numbered from the left, starting from zero, and the left-most pane is 'elastic', so that it will take up all the space not allocated to the other panes.

As well as the default pane 0, the default status bar added to your project by AppWizard has three panes, which are used to show the state of the *CapsLock*, *NumLock* and *ScrollLock* keys respectively.

By adding another item to this array, and to the string table in the program's resources, we can put another pane into the status bar at the position we choose.

Adding the New Pane to the Code

Use the string table editor to create a new string, **ID_INDICATOR_BMP**, in the same segment of the string table as the other indicator IDs.

I mentioned above that the length of the string will determine the size of the pane when the status bar is created, so it's important to give the string a value which will result in a pane large enough to hold your bitmap. In this example, a value such as MMM will be sufficient. It doesn't actually matter what the string is. It's just a dummy, which the status bar creation routine uses to calculate a size for the pane.

Now add the ID to the **indicators[]** array in `Mainfrm.cpp`, putting it wherever you want the new pane to appear. I've put it as the first one after the separator, so that it'll appear before the key-state panes:

```
static UINT indicators[] =
{
   ID_SEPARATOR,           // status line indicator
   ID_INDICATOR_BMP,
   ID_INDICATOR_CAPS,
   ID_INDICATOR_NUM,
```

```
    ID_INDICATOR_SCRL,
};
```

Adding a Bitmap

Now that we've added the pane, we need a way to draw the bitmap into it. The way to do this is to override the **CStatusBar**'s **DrawItem()** function; this will get called whenever an owner-draw pane needs to be updated.

If you look at the messages which a **CStatusBar** can handle, you'll find one called **WM_DRAWITEM**, and you might wonder why this isn't used. The answer is that **WM_DRAWITEM** is the message sent to an owner-draw control to tell it to draw itself, and in this case, you're not drawing the complete control, but using a special mechanism to override the drawing of one pane.

To do this, we can create a class (using ClassWizard), which inherits from **CStatusBarCtrl**, so that we can override **DrawItem()**. We don't actually want to have our class based on the control, so we need to change the references to **CStatusBarCtrl** to **CStatusBar** (there are two: one in the class declaration, the other in the message map). We can now override the **DrawItem()** member, as well as adding another member to hold the bitmap resource ID.

```
class CBitmapBar : public CStatusBar
{
...
   public:
      virtual void DrawItem(LPDRAWITEMSTRUCT lp);
      UINT m_BitmapID;
};
```

We add a line to the constructor to initialize the **m_BitmapID** member, otherwise we would be asking for trouble when we use it later:

```
CBitmapBar::CBitmapBar()
{
   m_BitmapID = IDB_GREEN;
}
```

When you use AppWizard to create an application that has a status bar, a data member is added to the application's frame window class. In the case of an SDI application, a data member, called **m_wndStatusBar**, is added to the **CMainFrame** class. We can replace it with one of our derived objects by changing the class declaration in `Mainfrm.h`, and including `Bitmapbar.h` in `Mainfrm.cpp`

```
#include "bitmapbar.h"

class CMainFrame : public CFrameWnd
{
protected: // create from serialization only
   CMainFrame();
   DECLARE_DYNCREATE(CMainFrame)

...

protected:  // control bar embedded members
   CBitmapBar  m_wndStatusBar;
   CToolBar    m_wndToolBar;

// Generated message map functions
protected:
```

```
   //{{AFX_MSG(CMainFrame)
   afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
   afx_msg void OnActionsRedraw();
   afx_msg void OnUpdateBmpPane(CCmdUI *pCmdUI);
   //}}AFX_MSG
   DECLARE_MESSAGE_MAP()
};
```

Create the Bitmaps

Before going any further, we need to create something to display in the pane. The length of string we've given in the table means that we need a bitmap about 11 pixels high and 22 wide. We'll create one with two 'lights', representing a stop/go signal. Use the resource editor to create two bitmaps—one with red and gray lights, and the other with gray and green—to represent the stop and go states. Call them **IDB_RED** and **IDB_GREEN**, using the Edit/Properties... option to change the ID.

Draw the Bitmap

There are two stages to drawing the bitmaps. First, we have to tell the status bar that the pane is owner-draw, and then we need to provide the drawing code.

The status bar is set up in **CMainFrame::OnCreate()**. We can add a call to **CStatusBar::SetPaneStyle()**,to set our bitmap pane to be owner-draw:

```
   m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
         CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);

   m_wndStatusBar.SetPaneStyle(1, SBT_OWNERDRAW);
```

Now, whenever the status bar needs to refresh pane 1, it will call **DrawItem()**, which our class needs to override. If you look in the MFC source, in `Barstat.cpp`, you'll see why:

```
// Derived class is responsible for implementing all of these
// handlers for owner/self draw controls.
void CStatusBar::DrawItem(LPDRAWITEMSTRUCT)
{
   ASSERT(FALSE);
}
```

Overriding DrawItem()

Here's our implementation of **DrawItem()**:

```
void CBitmapBar::DrawItem(LPDRAWITEMSTRUCT lp)
{
   CRect r(lp->rcItem);
   CBitmap bmp;

   // Blit the bitmap into the pane
   CDC bmpDC, paneDC;

   // Attach the paneDC object to the DC passed in
   paneDC.Attach(lp->hDC);

   // create a memory DC in which to prepare the bitmap
```

```
        bmpDC.CreateCompatibleDC(NULL);

        // load a bitmap into the DC
        bmp.LoadBitmap(m_BitmapID);
        CBitmap* pOldBmp = bmpDC.SelectObject(&bmp);

        // stretch the bitmap into the pane
        paneDC.StretchBlt(r.left, r.top,
            r.Width(), r.Height(),
            &bmpDC, 0, 0, 22, 11, SRCCOPY);

        bmpDC.SelectObject(pOldBmp);
    }
```

The first thing to look at is the data item passed in, which is a pointer to a Windows data structure, called a **DRAWITEMSTRUCT**. This is used for all of the owner-draw controls, and looks like this:

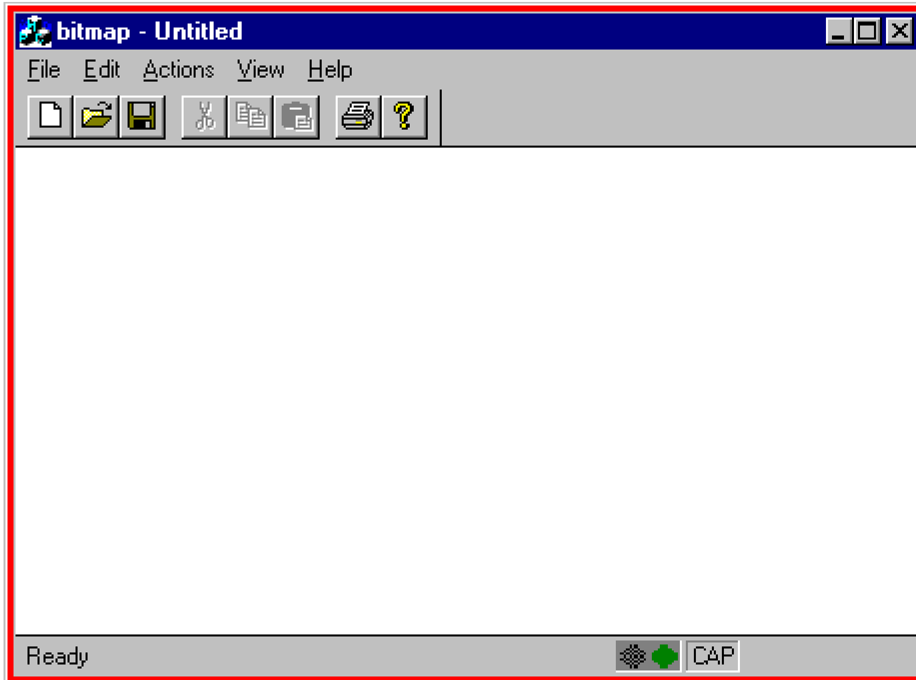
```
typedef struct tagDRAWITEMSTRUCT {
    UINT   CtlType;
    UINT   CtlID;
    UINT   itemID;
    UINT   itemAction;
    UINT   itemState;
    HWND   hwndItem;
    HDC    hDC;
    RECT   rcItem;
    DWORD  itemData;
} DRAWITEMSTRUCT;
```

Many of the fields don't concern us for now; those which do in this example are **hDC**, which is the device context into which we can draw, and **rcItem**, which is the rectangle occupied by the control. In other words, the **DRAWITEMSTRUCT** contains all the information we need to be able to decide how to draw the control and then to draw it.

We first create two MFC **CDC** objects—the first to hold the bitmap and the second to represent the pane—so we attach it to the DC passed in as part of the **DRAWITEMSTRUCT**. Next, the bitmap resource is loaded and placed into the first **CDC**.

We've now got two **CDC** objects, and we can use the **CDC::StretchBlt()** function to copy the contents of the bitmap DC into the pane DC.

When you code all this up and build the program, you'll get the result shown below.



Changing the Bitmap

Depending on what your application is doing, you may well want to change the bitmap at some point,; examples might include indicating the status of a connection to a server application, or the status of a print job.

We'll simulate this by allowing you to switch between the red and green bitmaps using a menu item, although you'd use exactly the same method if you did it completely under program control. First, add a suitable menu item, give it an ID and install a handler for the menu item. I put the handler in the mainframe class, because that's the class that owns the status bar, so it makes the coding easy for the purposes of this example. You may find that some other class makes a better owner for the handler.

The handler simply calls a routine, telling the status bar to switch the bitmaps:

```
void CMainFrame::OnActionsSwitchbitmaps()
{
   // Tell the status bar to switch the bitmaps
   m_wndStatusBar.SwapBmp();
}
```

The status bar does the switch and then causes a redraw:

```
void CBitmapBar::SwapBmp()
{
   m_BitmapID = m_BitmapID == IDB_GREEN ? IDB_RED : IDB_GREEN;
   InvalidateRect(NULL);
}
```

When you've implemented this, selecting the menu item will switch the bitmaps. You can see it more easily if you set up a toolbar button to trigger the same action.

Another neat trick is to allow the user to change the status of the application through the bitmap (Word 95

does something similar with its status bar; for example double-clicking on the MRK pane brings up the Revisions dialog), and doing this in our application is quite easy. All we need is to add a handler for the **WM_LBUTTONDBLCLK** message:

```
void CBitmapBar::OnLButtonDblClk(UINT nFlags, CPoint point)
{
   // TODO: Add your message handler code here and/or call default
   CRect rectBmpPane;
   GetItemRect(CommandToIndex(ID_INDICATOR_BMP), &rectBmpPane);
   if (rectBmpPane.PtInRect(point))
      SwapBmp();

   CStatusBar::OnLButtonDblClk(nFlags, point);
}
```

As you can see, we get the rectangle of the pane containing the rectangle, then make sure that the user actually clicked in that pane. We then call the **SwapBmp()** function that we set up in the class to swap the bitmaps.

Making a Masked Edit Control

One of the most frequently requested modifications of a standard control is to be able to apply some sort of masking to an edit control, so that it only (for instance) accepts numeric or alphabetic characters. It just so happens that Visual C++ 4.x includes an OLE control in the Component Gallery which does just this, but it's possible that the formats supported by this control may not do quite what you want (or even that you're using a version of Visual C++ that doesn't have the Component Gallery), so it's is still useful to see how to do it yourself.

In this section, we'll see how to develop an edit control into which you can plug any type of input filter you wish.

Subclassing CEdit

To filter out input characters that we don't want, we need to intercept them before they get displayed in the edit control.

Each time a character is entered into the control, it results in a **WM_CHAR** message being sent to the edit box. If we derive our own class from the MFC edit control class, **CEdit**, and provide a handler for **WM_CHAR**, we can check the character against a mask and only pass on to the edit control those which are acceptable. Doing it this way does have one disadvantage, which is that we have to derive a new **CEdit** class for each type of input we wish to filter.

To get around this, and to make the class more flexible, we won't do the checking directly in the **WM_CHAR** handler, but will push the checking out to another mask class, which does the checking for us. The edit class will contain a pointer to a mask object, and we can then write as many different mask classes as we want, and attach them to the edit control at run time. Working this way means that we only have to provide a simple, plug-in mask class for each type of input we wish to process, and we can even change them at run time should we ever wish to.

Our first step is to derive a new edit control class, so use ClassWizard to add a new class, inheriting from **CEdit**, and call it, say, **CExtEdit**.

The first modifications to make to this class are to add a handler for the **WM_CHAR** message, a data member which points to a masking object (which we'll discuss in a minute or two), and a means of setting it:

```
#include "MaskBase.h"

/////////////////////////////////////////////////////
// CExtEdit window

class CExtEdit : public CEdit
{
   // pointer to mask for this edit control
   CMaskBase* m_pMask;

// Construction
public:
   CExtEdit();

// Operations
public:
   void SetMask(CMaskBase* pM);

// Overrides
```

```
   // ClassWizard generated virtual function overrides
   //{{AFX_VIRTUAL(CExtEdit)
   //}}AFX_VIRTUAL

// Implementation
public:
   virtual ~CExtEdit();

   // Generated message map functions
protected:
   //{{AFX_MSG(CExtEdit)
   afx_msg void OnChar(UINT nChar, UINT nRepCnt, UINT nFlags);
   //}}AFX_MSG

   DECLARE_MESSAGE_MAP()
};
```

At this stage, we just need to arrange for the initialization and destruction of the data member:

```
CExtEdit::CExtEdit()
{
   m_pMask = NULL;
}

CExtEdit::~CExtEdit()
{
   // We're responsible for getting rid of any mask
   if (m_pMask)
      delete m_pMask;
}
```

Providing Masks

The input masking is going to be provided by a series of classes, all of which will derive from a vanilla base class, **CMaskBase**. Here's its definition:

```
class CMaskBase
{
public:
   virtual BOOL AddChar(UINT nChar) = 0;
};
```

The single member function, **AddChar()**, is called to process each character entered into the edit control, and returns **TRUE** or **FALSE**, depending on whether a character should be accepted or not. This function needs to be overridden by derived classes in order to provide their custom behavior, so it's made a pure **virtual** function.

How will we set and use the mask in the edit control? **CExtEdit::SetMask()** stores the pointer away for future use, but, first, it checks that the pointer passed in was valid, and then it deletes any existing mask object. It has to do this because we've decided that it's our edit class that is responsible for the ownership of the mask objects:

```
void CExtEdit::SetMask(CMaskBase* pM)
{
   ASSERT(pM);
   // If there is already a mask, zap it
   if (m_pMask)
      delete m_pMask;
   m_pMask = pM;
}
```

Deleting the old mask lets us change masks in mid-program, should we ever wish to. I can't really think of a time when I'd actually want to do this, but it's always a good idea to plan ahead and make features available, even if you can't see a good use for them right now. Note the use of an assert to check the pointer argument; this is fine during development, but, for production code, you might want to replace it with a better run-time check, such as throwing an exception, or passing back a return value showing that the function failed.

The mask is used in the **WM_CHAR** handler:

```
void CExtEdit::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
   // Check if we have a handler set up
   if (m_pMask == NULL)
      throw "No handler defined!";

   // Pass the character onto the mask object to see whether
   // it is acceptable
   if (m_pMask->AddChar(nChar))
      CEdit::OnChar(nChar, nRepCnt, nFlags);
   else
      MessageBeep(0);
}
```

The first thing is to check that a mask has been set. Here, we choose to throw an exception if it hasn't. This is the sort of place that you'd want to use exceptions; where a fatal error has been encountered in a low-level routine and you want to give the application code a chance to handle it. Incidentally, remember to remove the default call to **CEdit::OnChar()**, or your handler won't work as expected.

If the mask is in place, we pass it the character and see whether we get **TRUE** or **FALSE** returned. If **TRUE**, we pass the character on to **CEdit::OnChar()**, which results in it getting displayed; if **FALSE**, we beep to alert the user.

Mask Classes

Before we can use masked editing, we need to define some actual masks. The first one is a numeric edit control for simple integer values:

```
#include "MaskBase.h"

class CMaskNum : public CMaskBase
{
   int nCharPos;
public:
   CMaskNum() : nCharPos(0) {}

   // Override pure virtual base class function
   virtual BOOL AddChar(UINT nChar);
};
```

Most mask classes will have to save some notion of state, to know where they've got to in the input so that they can decide whether a given character is valid. As an example, in our numeric mask, it's valid to have a + or a - character entered, but only as the first one, so we need to save the number of characters entered.

We can see how this state is used in the implementation of **CheckChar()**:

```
#include "stdafx.h"
#include "masknum.h"

#define BKSPACE 8

BOOL CMaskNum::AddChar(UINT nChar)
{
   // Valid character set is '+' and '-' (as first character
   // only), '0'-'9' and backspace.
   BOOL bOK;

   switch(nChar)
   {
   case '+':
   case '-':
      // Only valid as the first character
      if (nCharPos == 0)
      {
         nCharPos++;
         bOK = TRUE;
      }
      else
         bOK = FALSE;
      break;

   case '0': case '1':
   case '2': case '3':
   case '4': case '5':
   case '6': case '7':
   case '8': case '9':
      nCharPos++;
      bOK = TRUE;
      break;

   case BKSPACE:
      // Handle backspace, adjusting character count (but don't
      // go back past zero!)
      nCharPos = (nCharPos > 0) ? nCharPos-- : 0;
      bOK = TRUE;
      break;

   default:
      bOK = FALSE;
   }

   return bOK;
}
```

Valid characters are digits, + and -, and backspace (ASCII character 8). When a backspace is entered, we need to take care to adjust the character count correctly.

This mask is one of the simplest, and most others require considerably more work. For instance, a general floating-point number mask may have to be able to process input like the following:

```
1.33
-2.06
+.311E2
-1.245e-7
```

Many such parsing problems are well handled by state machines, so if you find yourself having to provide complex filters, I suggest that you read up on this approach.

Using Masked Edit Controls

Finally, we get to use the mask in a program. Create a dialog containing one or more edit controls and use ClassWizard to attach a **CEdit** control object to one of them. Then manually edit the code that ClassWizard inserts into the dialog's class definition, changing the **CEdit** to a **CExtEdit**.

```
#include "ExtEdit.h"

/////////////////////////////////////////////////////////////////
// TestDlg dialog

class TestDlg : public CDialog
{
// Construction
public:
   TestDlg(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
   //{{AFX_DATA(TestDlg)
   enum { IDD = IDD_TESTDLG };
   CExtEdit m_NumEdit;
   //}}AFX_DATA

   // (Rest of class definition is unchanged...)
};
```

This change will result in the **CExtEdit** class getting all the messages for the edit control. Now all you have to do is to set the edit control's mask in the dialog initialization code.

```
BOOL TestDlg::OnInitDialog()
{
   CDialog::OnInitDialog();

   // Create a new numeric mask, and attach it to the edit
   // control
   m_NumEdit.SetMask(new CMaskNum);

   return TRUE;
}
```

Remember that we've made the edit control responsible for the mask object, so we don't have to delete it externally.

Extending the ExtEdit Class

There are quite a few ways to extend the edit class. Say you've implemented a date mask and the users of your program may live in any of several countries, then you could attach a right mouse button menu to the edit control by providing a handler for **WM_CONTEXTMENU**, as shown elsewhere in this chapter, and use it to allow the user to select their choice of date formats at run time.

A more ambitious extension is to use the mask class to control the appearance of the text in the edit control, so that it comes out properly formatted. To do this, you need to provide a handler for the **EN_UPDATE** notification message, which is sent just before an edit control is about to display altered text. This is usually sent to the parent of the control, but using message reflection (as discussed a couple of topics further on), you can arrange for the edit control to handle the message itself.

Making an Owner-draw Button

Some Windows controls can be made either wholly or partially **owner-draw**, meaning that the owner of

the control takes over responsibility for rendering it, rather than letting Windows draw it in its normal style. We've already seen an example of this when we drew a bitmap in one of the panes of a status bar, by making that pane an owner-draw pane.

As we mentioned when we were discussing that example, MFC control classes which support owner-draw implement a **virtual** function called **DrawItem()**. If you want to take over drawing, you need to derive a new class from the MFC control class, then override the **DrawItem()** function. This is enforced by the fact that the base class version of this function does nothing except generate an assertion error.

If you've declared your control to be owner-draw, each time Windows needs to redraw the control it will call your **DrawItem()** function, passing it information about the state of the control and the device context in which to draw.

Note that by making a control owner-draw, you're taking on all responsibility for drawing it yourself. For a button, you need to handle the selected, deselected and disabled states, with and without focus, drawing all the shading and highlights. It can be a lot of work.

Let's look at how you might go about taking over drawing a button, by drawing one which is subtly different from the usual sort, with button text which 'floats' over the surface of the button. The result is similar to some custom buttons which Borland used to use.

Creating the Basic Class

First, create a testbed project with a simple dialog. Put a button somewhere in the dialog, and check the Owner draw style in the dialog editor.

The next task is to use ClassWizard to create a new class which inherits from **CButton**, calling it **CSpecialBtn**, and override the **DrawItem()** function. Next, attach a **CSpecialBtn** control variable to the button in your dialog, then exit from ClassWizard. This will ensure that all the messages for that button will be sent to our **CSpecialBtn** class.

If you build and run the project at this stage, with a blank **DrawItem()** function, you'll see a dialog with nothing where the button should be. Windows is handing over all responsibility for drawing the button to our class. Since it isn't doing anything, nothing appears on the screen.

Let's go through the code to draw a rectangular button which looks (and behaves) pretty much like a normal Windows button.

Drawing the Button

The first thing to do is to fill in the background color of the button and to draw its outline. We have two cases to consider for the outline: when the button has the focus, in which case it has a thick border, and when it doesn't, in which case the border is thin:

```
void CSpecialBtn::DrawItem(LPDRAWITEMSTRUCT lpDS)
{
   CDC dc;
   dc.Attach(lpDS->hDC);

   // Fill the button with current button face colour
   CBrush btnBrush(::GetSysColor(COLOR_BTNFACE));
   CRect rct(&(lpDS->rcItem));
```

```
   dc.FillRect(rct, &btnBrush);

   // Draw the outside frame
   CBrush blkBrush(RGB(0,0,0));
   dc.FrameRect(rct, &blkBrush);
   rct.InflateRect(-1,-1);

   // Draw a thick border if we've got the focus
   if (lpDS->itemState & ODS_FOCUS)
   {
      dc.FrameRect(rct, &blkBrush);
      rct.InflateRect(-1,-1);
   }
}
```

As this is a routine which draws a control, we are passed a pointer to a **DRAWITEMSTRUCT**, which we looked at earlier in the status bar example. Here, we're concerned with three fields: **hDC**, which is the device context into which we can draw, **rcItem**, which is the rectangle occupied by the control, and **itemState**, which tells us what state the control is in, selected, has focus or whatever.

The device context passed into **DrawItem()** as part of the argument structure is a bare Windows **HDC**, and it will be easier to work with if we turn it into an MFC **CDC** object, which we do by creating a **CDC** and then using **CDC::Attach()**.

The next thing to do is to fill the button with the right background color, which we get from the system constant **COLOR_BTNFACE**. Using this rather than an actual color value will enable the button to respond to changes the user might make to their desktop color scheme.

We can provide the outline of the control by drawing a rectangle just inside the boundary of the button, and then shrinking the bounding rectangle slightly so that future drawing operations won't draw over what we've done. The simple way to do this, which we'll implement, is to use **FrameRect()** to draw round the boundary. However, if you look closely at ordinary buttons, you'll notice that the lines forming their boundaries don't join at the corners, so if you want to mimic Windows buttons exactly, you'll need to draw discontinuous lines.

If the control has the focus, we then provide the thicker border by drawing another rectangle immediately inside the first, and then shrinking the bounding rectangle yet again.

If you build and run the project now, you'll see the outline of the button. It will respond to gaining and losing the focus, but what it hasn't got yet is any sort of 3D look, which means that it won't show when the user clicks on it.

Adding a 3D Effect to the Button

The button's outer frame is static, so it looks the same, regardless of whether the button is pressed or not. We now need to add the drawing code to change the appearance of the button when it's selected.

Here's the code we use:

```
void CSpecialBtn::DrawItem(LPDRAWITEMSTRUCT lpDS)
{
   ...
   CPen shPen(PS_SOLID, 1, ::GetSysColor(COLOR_BTNSHADOW));
   CPen *pOldPen = dc.SelectObject(&shPen);

   if (lpDS->itemState & ODS_SELECTED)
```

```
    {
       dc.MoveTo(rct.left, rct.bottom-1);
       dc.LineTo(rct.left, rct.top);
       dc.LineTo(rct.right, rct.top);
    }
    else
    {
      // draw highlight
      CPen hiPen(PS_SOLID, 1, ::GetSysColor(COLOR_BTNHILIGHT));
      dc.SelectObject(&hiPen);
      dc.MoveTo(rct.left, rct.bottom-1);
      dc.LineTo(rct.left, rct.top);
      dc.LineTo(rct.right, rct.top);

      // draw shadow
      dc.SelectObject(&shPen);
      dc.MoveTo(rct.left, rct.bottom-1);
      dc.LineTo(rct.right-1, rct.bottom-1);
      dc.LineTo(rct.right-1, rct.top-1);

      dc.MoveTo(rct.left+1, rct.bottom-2);
      dc.LineTo(rct.right-2, rct.bottom-2);
      dc.LineTo(rct.right-2, rct.top);
    }
    ...
```

We use another system constant to determine the current color of the shadowing used on buttons. If you look at a button, you'll see that the shadow is drawn along the bottom and right edges when it's up, and (slightly thinner) along the top and left edges when it is selected. Accordingly, we test for selection and draw either a pair of single lines if the button is selected, or a pair of double lines if not.

All we're lacking now is the text on the button. We need to calculate the size and position of the bounding rectangle for the string, so that it will be drawn in the correct place, and to make allowances for the offset introduced when the button is selected:

```
...
   // Draw the text - first get the size of the text
   CString btnText;
   GetWindowText(btnText);
   CSize textSize = dc.GetOutputTextExtent(btnText);

   // Calculate the centered rect the text needs to be drawn in, and make
   // allowances for it being selected
   CRect textRect = rct;
   int xSize = (rct.Width() - textSize.cx)/2;
   int ySize = (rct.Height() - textSize.cy)/2;
   textRect.InflateRect(-xSize, -ySize);

   dc.SetBkMode(TRANSPARENT);

   // If the button is selected (i.e. pressed), write the
   // shadow of the text in gray, offset by two pixels.
   if (lpDS->itemState & ODS_SELECTED)
   {
      textRect.OffsetRect(2,2);
      dc.SetTextColor(::GetSysColor(COLOR_GRAYTEXT));
      dc.DrawText(btnText, textRect, DT_SINGLELINE | DT_CENTER);
      textRect.OffsetRect(-2,-2);
   }

   // Draw the actual text in red, to make it stand out
   dc.SetTextColor(RGB(255,0,0));
   dc.DrawText(btnText, textRect, DT_SINGLELINE | DT_CENTER);
   ...
```

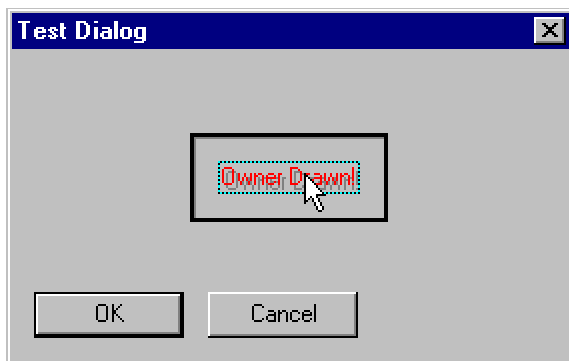
On normal buttons, the text moves when the button is pressed. Here, the text stays where it is, and a

'shadow' of the letters is displayed, offset by two pixels. This gives the effect of the text floating above the surface of the button.

The final touch is to add the focus rectangle —the small dotted rectangle which surrounds the text on a button when it has the focus. The **CDC** class provides a function to draw a focus rectangle, and we have something suitable already available, so by inflating the text rectangle slightly, we'll fit a focus rectangle neatly around the text.

```
      ...
      // Draw the focus rectangle if we need to
      if (lpDS->itemState & ODS_FOCUS)
      {
         textRect.InflateRect(1,1);
         dc.DrawFocusRect(textRect);
      }

      dc.SelectObject(pOldPen);
   }
```

And here's the result...



Using Message Reflection

Message reflection, as its name implies, involves sending a message back for processing to where it came from.

When is this useful? A lot of controls send notification messages to their parent window when something happens to them. For instance, when you click on a button, it sends a **BN_CLICKED** notification message to its parent, which is usually the window in which it is embedded. In Windows programs, it has traditionally been this parent window which has intercepted the message and performed the action.

There are many cases, however, where we might wish the control itself to handle the event. In an object-oriented world, it makes sense for the control object itself to handle messages, especially when forcing the parent to do so would result in code being duplicated in a number of other classes. There's also an added advantage that we can write self-contained control classes which can be reused much more easily than ones in which the parent window has to process messages.

This technique of getting a control to handle its own messages is called **message reflection**, and support for it was added to Visual C++ 4.0.

To show how it works, we'll develop a self-contained Help button which handles its own **BN_CLICKED**

message, and uses it to display a screen from a help file.

Create the Project

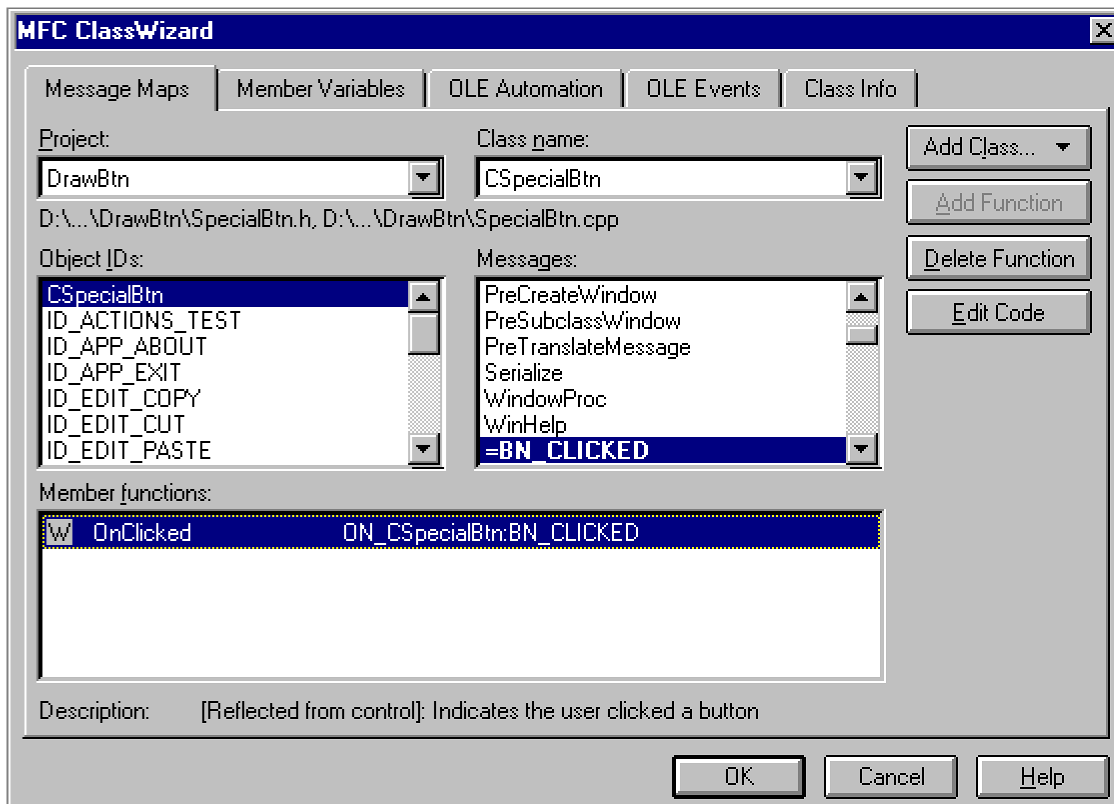
Create yourself a project as a test bed for this control; any type of project will do (SDI or MDI), but remember to turn on context-sensitive help!

Create a Custom Button Class

To create a button which can handle its own messages, we need to subclass MFC's **CButton** class in order to override its default behavior again.

The easiest way to do this is to use ClassWizard to add a class, called (say) **CSpecialBtn**, giving **CButton** as its parent. Once you've done this, you can use ClassWizard to provide message handlers for the reflected messages.

In ClassWizard's Message Maps tab, reflected messages are marked with a leading =…



If you examine the messages listed for the button class, you'll find that there are four reflected messages: **BN_CLICKED**, **BN_DOUBLECLICKED**, **WM_CTLCOLOR** and **WM_PARENTNOTIFY**.

In our case, we want to handle a single click on the button, so add a handler for the =BN_CLICKED message, as shown above.

Displaying Help

When you generate a project with AppWizard, you can ask for context-sensitive help to be included in the skeleton application.

As well as adding a Help menu to your app's menu bar, it also generates the `.rtf` source for a skeleton Windows help file, with entries for all the standard menu items, which you can then tailor to your particular requirements. You can use the `Makehelp.bat` file to compile the `.rtf` file into a `.hlp` file, or use the Help Workshop provided with Visual C++.

You can display help screens from this help file using the **`CWinApp::WinHelp()`** function, passing it the appropriate context ID. You don't have to supply a help file name, as the app knows which one it should be using.

So, for our button to display help, it only needs to be passed the context ID of a help page; it can get a pointer to the **`CWinApp`**-derived application object by calling **`AfxGetApp()`**.

Add to your button class a member function called **`SetContextID()`**, which will be used to set the ID used in help calls:

```
void CSpecialBtn::SetContextID(unsigned long lID)
{
   m_ContextID = lID;
   m_bGotContextID = TRUE;
}
```

Here, **`m_ContextID`** is an **`unsigned long`** which stores the ID, while **`m_bGotContextID`** is a Boolean flag which shows whether the ID has been set. I use a flag rather than set some default value for the ID, because you can't be sure that whatever value you choose won't be used sometime as a valid context ID. Remember to set its initial value to **`FALSE`** in the constructor!

Now we have a way to get the context ID into the object, we need to display the help page. The code is pretty simple:

```
void CSpecialBtn::OnClicked()
{
   // When the user clicks on the button, the notification will
   // come here, rather than to the parent window

   // For displaying help, make sure that the context ID
   // is set first.
   if (!m_bGotContextID)
      MessageBox("Help context ID not set!", "Error",
         MB_ICONHAND | MB_OK);
   else
      AfxGetApp()->WinHelp(m_ContextID);
}
```

Modify the About Dialog

Let's test it by adding a help button to the About dialog. Bring up the dialog in the dialog editor and add a button with the ID **`IDC_HELP_BTN`**.

Then bring up ClassWizard and add a control member variable to the Help button, of type **CSpecialBtn**. Remember to include the header for **CSpecialBtn** into Helpbtn.cpp.

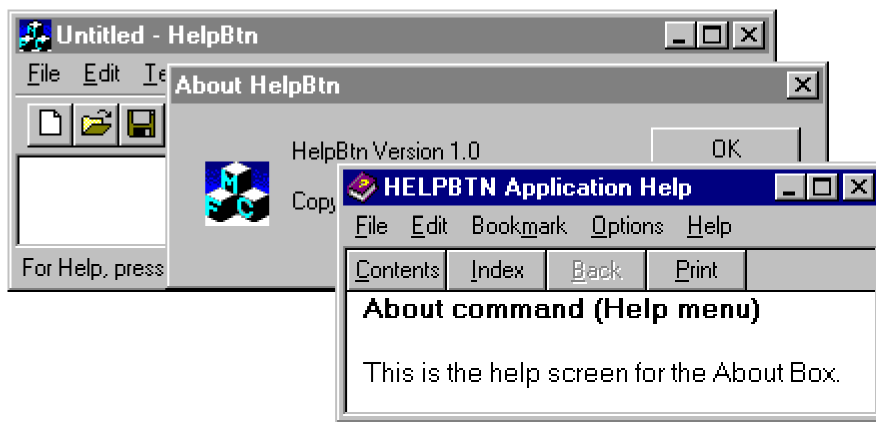
The only other thing we need to do is to ensure that the button is passed the correct context ID when the dialog is created, which we can easily do in the dialog's constructor:

```
CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
   //{{AFX_DATA_INIT(CAboutDlg)
   //}}AFX_DATA_INIT
   m_HelpBtn.SetContextID(0x20064);
}
```

You can find the correct context ID in the .hm file, in the Hlp subdirectory. This file, generated by the Makehelp.bat file which runs the help compiler, lists the context IDs which were generated for the dialogs, menu items and other resources in your program.

Build and Test the Program

Once you've coded up the button class and the changes to the About dialog, you can build the program and test out the button.



The important thing to realize about this example is that, in handling reflected messages, our button provides a self-contained solution, which you can easily include in other dialogs in the program. In fact, by adding the button class to the Component Gallery when you create it, you can use it in other projects—one more step along the way to reusable software components!

Modifying the Common Dialogs

The common dialogs are an incredibly useful resource, providing access to several frequently used operations.

Sometimes, though, we may wish to modify these dialogs. It may be a very simple change, such as changing the text of a label or button, or more major surgery, such as adding extra controls or changing the layout of the dialog.

In the next couple of sections, we'll look at how to modify the File Open/Save dialog, as this is one of the most popular dialogs for modification. You can, however, apply this technique equally well to any of the other common dialogs.

Modifying the Text in a CFileDialog

Let's first look at customizing the text. You can change the text associated with any of the controls in the common dialogs. For instance, in the next chapter, Ken Ramirez modifies the File Open dialog to use it for creating shortcuts, so that the OK button reads Create, and the Read Only check box label reads Save to desktop. This is fine, because the Read Only check box simply sets a flag, which you can interpret as you wish.

There are two steps to customizing the text in a common dialog. First, we need to find the IDs of the controls that we're going to change, and then subclass the dialog class so that we have somewhere to do the changing. (You knew it was going to involve subclassing somewhere, didn't you?)

Common Dialog Resource Files

The key to finding the control IDs lies in two files—`Dlgs.h` and `Fileopen.dlg` —which live in the main include directory (i.e. `\Msdev\Include`).

`Fileopen.dlg` holds the templates of the File Open common dialogs, while `Dlgs.h` gives the ID of each control, including static strings.

If you open and browse through `Fileopen.dlg`, you'll find two dialog templates. We're interested in **`FILEOPENORD`**, the standard file selection dialog. If you examine the template, you'll see that each control (even static text items) has an ID, and it's these that we can use to customize the text.

Creating a Modified Dialog

Once you've used ClassWizard to create a new project, create a new class which inherits from **`CFileDialog`**. Call it **`CExtFileDlg`**.

We want to alter the controls after the Windows dialog has been created, but before it's displayed, so the place to do this is in the dialog's initialization code.

Use ClassWizard to provide a handler for `WM_INITDIALOG`, which will get called just before the dialog is displayed. Leave the call to the base class `OnInitDialog()` routine as the first thing in the handler, because we want to override what the **`CFileDialog`** base class sets up.

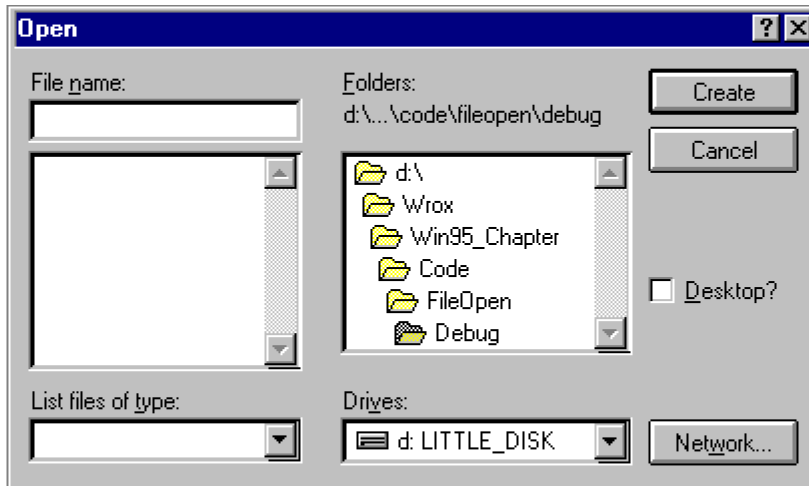
It's now a matter of using **SetWindowText()** for any control you want to modify; you'll need to add **#include <dlgs.h>** in your dialog class source file in order to be able to use the symbolic names for the control IDs.

Unfortunately, setting the text of the controls is complicated by the differences between old style (Windows NT 3.51) and new style (Windows 95) dialogs. If you're using old style dialogs, the controls with the text we want to change are part of the dialog we're overriding. However, if you use the new Explorer-style dialogs, these controls actually belong to the parent of our dialog. That makes the code a little more complicated:

```
BOOL CExtFileDialog::OnInitDialog()
{
   CFileDialog::OnInitDialog();

   CWnd* pParent = GetParent();

   // The items we are looking for may be in this dialog or its
   // parent, depending on whether we're using new (Win95) or old style
   // dialogs.
   CWnd* pOK = pParent->GetDlgItem(IDOK);    // Win95-style
   if (!pOK)
      pOK = GetDlgItem(IDOK);                // Old style
   if (pOK)
      pOK->SetWindowText("Create");

   // If we're using new style dialogs, we have more room for
   // our text string
   CWnd* pChk1 = pParent->GetDlgItem(chx1);
   if (pChk1)
      pChk1->SetWindowText("Add to &Desktop");
   else
   {
      pChk1 = GetDlgItem(chx1);
      if (pChk1)
         pChk1->SetWindowText("&Desktop?");
   }

   return TRUE;  // return TRUE unless you set the focus to a control
                 // EXCEPTION: OCX Property Pages should return FALSE
}
```

In this example, we'll change the text on the OK button (whose ID is **IDOK**) and on the Read-only check box (ID **chx1**). Note how we use **GetDlgItem()** to get a pointer to the **CWnd**-derived object behind the control and set its text.

You can easily test this by adding a menu item that brings up a **CExtFileDlg**.



Note that the dialog and its controls will still appear at the same size, so you have to be careful not to make your text strings longer than the originals, else they'll get clipped.

More Major Surgery to CFileDialog

We can get around this by modifying the dialog template itself. As with all dialogs, as long as we leave the controls with the right IDs and properties, we're free to change their position and size.

We need to do a little bit of coding, because we now need to tell the base common dialog code to use our new, modified dialog template.

What we'll do is to create a version of the File Open dialog which adds information about the file currently selected in the list box at the bottom of the dialog. The information presented will consist of the filename, its size, the last modification date, and file attributes, like this,

```
MainFrm.h, 1763 bytes, 07/03/96 (a)
```

where the (a) shows that the archive bit is set.

Create the Project

Once again, create yourself a project as a test bed for this control; any type of project will do (SDI or MDI). Then add a menu handler for File/Open..., which will be used to display the modified common dialog.

Add the Dialog Resources to the Project

So that we can edit the layout of the dialog with the tools in the Visual C++ IDE, we need to add the File Open dialog resources to our project resources. This happens in three steps:

1. Select the View/Resource Includes... menu item.

2 Add **#include <dlgs.h>** and **#include <fileopen.dlg>** to the Read-only symbol directives list box.

3 Save the project away and reopen it.

You will now have the two File Open dialogs as editable resources in your project.

Now go back into the View/Resource Includes... dialog and remove the **#include <fileopen.dlg>** line. You need to do this because the dialogs were added to the project when we first reopened it, but the **#include**s will also be parsed each time we compile, so if you don't take the reference out, you'll get the linker complaining about multiple inclusions.

Subclassing the File Dialog

Add a new class, **CNewFileOpen**, inheriting from **CFileDialog** as before. To be able to use our new dialog template, we need to give it a dialog ID , but ClassWizard won't give us one (the Dialog ID combo box is disabled) because it doesn't reckon we need one. So, hack into `Resource.h` and add,

```
#define IDD_NEWFILEOPEN  103
```

(or some other suitable ID).

We can now add the dialog ID to the source for **CNewFileOpen**, in the public section of the class definition:

```
enum { IDD = IDD_NEWFILEOPEN};
```

We need to rename the File Open dialog to give our modified template a unique name, so rename it **IDD_NEWFILEOPEN**. The fact that the ID is the same as that of the dialog ID will enable the system to load the correct template at run time.

We need to tell this new class to use our dialog template rather than the default one, so add **OFN_ENABLETEMPLATE** to the default flags in the constructor:

```
class CNewFileOpen : public CFileDialog
{
   DECLARE_DYNAMIC(CNewFileOpen)

public:
   CNewFileOpen(BOOL bOpenFileDialog,
      LPCTSTR lpszDefExt = NULL,
      LPCTSTR lpszFileName = NULL,
      DWORD dwFlags = OFN_OVERWRITEPROMPT | OFN_ENABLETEMPLATE,
      LPCTSTR lpszFilter = NULL,
      CWnd* pParentWnd = NULL);

      enum { IDD = IDD_NEWFILEOPEN };

protected:
   //{{AFX_MSG(CNewFileOpen)
      // NOTE …
   //}}AFX_MSG
   DECLARE_MESSAGE_MAP()
};
```

Now fill in the **hInstance** and **lpTemplateName** fields of the **OPENFILENAME** structure, so that the common dialog knows where to find the new template, and OR the **OFN_ENABLETEMPLATE** flag with the

flags passed to the base class constructor, to ensure that it gets through. That will ensure that our modified dialog template will get called.

```
CNewFileOpen::CNewFileOpen(BOOL bOpenFileDialog, LPCTSTR lpszDefExt, LPCTSTR lpszFileName,
      DWORD dwFlags, LPCTSTR lpszFilter, CWnd* pParentWnd) :
      CFileDialog(bOpenFileDialog, lpszDefExt, lpszFileName,
         // Ensure flag is passed to base class
         dwFlags |= OFN_ENABLETEMPLATE,
         lpszFilter, pParentWnd)
{
   // Set template information
   m_ofn.hInstance = AfxGetResourceHandle();
   m_ofn.lpTemplateName = MAKEINTRESOURCE(CNewFileOpen::IDD);

   m_ofn.Flags &= ~OFN_EXPLORER;
}
```

Note that we remove the **OFN_EXPLORER** bit from the **Flags** member of the **OPENFILENAME** structure so that Windows knows we're using the old style interface.

Now let's modify the template and add the code to use our modifications.

Modifying the Template

Next, you need to open the dialog in the editor and add a static text control to the bottom of the window. It'll be the full width of the dialog, and because we're going to update it, it will need an ID, say, **IDC_FILE_INFO**. If you want to make it stand out, you can place a group box around the text.

Handling List Box Notifications

The **CFileDialog** class has a function, **OnLBSelChangedNotify()**, which gets called when the selection in any of the dialog's list boxes or combo boxes changes.

We can override this and add the code which will update the file information in the static text control.

Add an override to your dialog class, either by right-clicking on the class in the ClassView pane, or manually:

```
class CNewFileOpen : public CFileDialog
{
   DECLARE_DYNAMIC(CNewFileOpen)

public:
   virtual void OnLBSelChangedNotify(UINT nIDBox, UINT iCurSel,
         UINT nCode);
   CNewFileOpen(BOOL bOpenFileDialog,
      LPCTSTR lpszDefExt = NULL,
      LPCTSTR lpszFileName = NULL,
      DWORD dwFlags = OFN_OVERWRITEPROMPT | OFN_ENABLETEMPLATE,
      LPCTSTR lpszFilter = NULL,
      CWnd* pParentWnd = NULL);

      enum { IDD = IDD_NEWFILEOPEN };

protected:
   //{{AFX_MSG(CNewFileOpen)
      // ...
   //}}AFX_MSG
   DECLARE_MESSAGE_MAP()
};
```

We only want to update the text when the user is changing files, so we need to check the control ID that is passed to the function. The file selection list box has the ID **lst1**, while that of the directory selection box is **lst2**. The code below shows the bare bones of what we want to do. This function is called when various events happen in the list boxes, so we need to filter the arguments for both the list box and the correct event.

```
void CNewFileOpen::OnLBSelChangedNotify(UINT nIDBox,
            UINT iCurSel, UINT nCode)
{
   // Get a pointer to the static text control
   CWnd* pText = GetDlgItem(IDC_FILE_INFO);
   ASSERT_VALID(pText);

   // If we're changing selection in the file list box,
   // change the text, otherwise blank it out.
   if (nIDBox == lst1 && nCode == CD_LBSELCHANGE)
      pText->SetWindowText("File:");
   else
      pText->SetWindowText(NULL);
}
```

Getting File Information

There are a number of ways to get information for a file. We're going to use the Win32 API call, **FindFirstFile()**. Just pass this function a string containing a filename and a pointer to a **WIN32_FIND_DATA** structure and the function will fill the structure with all the information our users might need to know about a file. Once the data's in the structure, it's a relatively simple task to extract it and present it to the user of our dialog.

Here's the code for the complete function:

```
void CNewFileOpen::OnLBSelChangedNotify(UINT nIDBox, UINT iCurSel,
                                        UINT nCode)
{
   // Get a pointer to the static text control
   CWnd* pText = GetDlgItem(IDC_FILE_INFO);
   ASSERT_VALID(pText);

   // Ditto for the file list box
   CListBox* pList = (CListBox*)GetDlgItem(lst1);
   ASSERT_VALID(pList);

   // If we're changing selection in the file list box,
   // change the text, otherwise blank it out.
   if (nIDBox == lst1 && nCode == CD_LBSELCHANGE)
   {
      // Get the full file path
      int nSel = pList->GetCurSel();

      if (nSel != LB_ERR)
      {
         // Get the status info
         CString sFile;
         pList->GetText(nSel, sFile);

         WIN32_FIND_DATA fd;
         HANDLE hRet = FindFirstFile((LPCTSTR)sFile, &fd);

         if (hRet == INVALID_HANDLE_VALUE)
            AfxMessageBox("Error getting file status info");
         else
         {
```

```
                CString sMsg;
                CString sDate;

                if (fd.ftLastAccessTime.dwLowDateTime == 0 &&
                   fd.ftLastAccessTime.dwHighDateTime == 0)
                   sDate = "<unknown>";
                else
                {
                   COleDateTime tModDate(fd.ftLastAccessTime);
                   sDate = tModDate.Format();
                }

                CString sAtt('(');
                if (fd.dwFileAttributes & FILE_ATTRIBUTE_READONLY)
                   sAtt += 'r';
                if (fd.dwFileAttributes & FILE_ATTRIBUTE_ARCHIVE)
                   sAtt += 'a';
                if (fd.dwFileAttributes & FILE_ATTRIBUTE_HIDDEN)
                   sAtt += 'h';
                sAtt += ')';

                sMsg.Format("%s, %ld bytes, %s %s", (LPCTSTR)sFile,
                    fd.nFileSizeLow, (LPCTSTR)sDate,
                    (sAtt != "()") ? (LPCTSTR)sAtt : "");

                // Update the text
                pText->SetWindowText(sMsg);
             }
          }
       }
       else
          pText->SetWindowText(NULL);
    }
```

The first task is to get the name of the file from the list box, so that we can find its attributes. We do this by getting a pointer to the list box object in the dialog, then using **`CListBox::GetCurSel()`** and **`CListBox::GetText()`** to get the name of the file.

The next step is to use **`FindFirstFile()`** to get the file information. We pass it a path, and it fills in a structure with information. You may wonder how we can pass it an unqualified filename, rather than the full path; this works because the File Open dialog resets the current directory when you change from one directory to another, so that we should always be in the right directory to use a filename without path information.

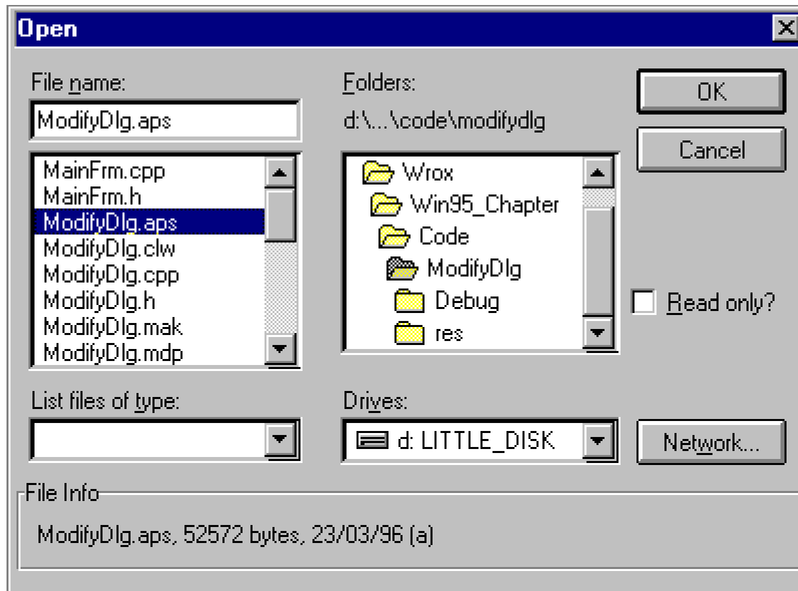
Displaying the File Information

The final stage is to format up the information and display it. This is easily done using the **`CString::Format()`** function to lay out a formatted string, but note how the date is handled.

MFC includes a date/time class, **`CTime`**, but this is rather limited in what it can do. This example uses the more versatile **`COleDateTime`** class, introduced in MFC 4.0, which is preferable for two reasons: first, it has a finer granularity and covers a wider range of dates than **`CTime`**, ranging from 1st January 100 to 31st December 9999, and secondly, it can automatically format the date and time based on the local language settings. To use **`COleDateTime`**, you need to **`#include <afxdisp.h>`** in your source file.

The Result

When you build and run the example, you should see a dialog looking like this:



What Can't You Do?

People often ask for things that turn out to be difficult to manage. Here, I'll point out a couple of common ones of which I'm aware.

It may seem rather strange to have such a negative section in the chapter, but who knows, at some point in the future, having these details to hand might save you some head scratching and wasted hours!

The first request is to put a common dialog on a property page. Imagine that you have a tabbed dialog which handles all the setup details for your word-processor application; one of the tabs is used to select fonts and another to select colors. Wouldn't it be neat if you could use the font and color picking common dialogs on the property sheet?

You can't! Many people have asked, and the answer is 'No'. The simple answer is that individual pages of a tabbed dialog need to be modeless dialogs, but the common dialogs are modal, so they won't work.

The second one also involves property sheets. We all know that they display with the tabs along the top, but is there a way to get the tabs in some other place, such as down the side (like a filofax), or along the bottom, like the Output window at the bottom of Developer Studio, or the equivalent in Excel?

It appears not. The word is that property sheets were designed to have the tabs along the top, and that no thought was given to them being customized. The tabbed windows in Developer Studio and Excel are apparently Microsoft's own custom controls.

Summary

So there you are, a few examples of how you can interact with the stock Windows controls and dialogs. In this chapter, we've covered:

- Menus
- Toolbars

- Status bars
- Subclassing
- Common dialogs

Of course, this list hasn't exhausted the potential of MFC by any means, but I hope it's given you some ideas.

Windows 95 Shell Programming

I hope that you're someone who can accept and adapt to change and move on, because, in this industry, things change often and in great leaps. The Windows interface is a case in point. No doubt as a computer user, you'll have been pleased, or even excited, to use the new features of the Windows 95 shell. Long file names, shortcuts, file viewers and many of the other characteristic elements of the new shell are a huge improvement over the previous Windows interface. As my parents once said, "You kids have it easy today".

However, it's as developers that you'll be reading this book, so in this chapter, you'll see how your applications can exploit the huge variety of new features that have arrived with the next generation of operating systems (Windows 95 and Windows NT 4.0).

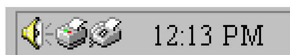
> *Since Windows NT 4.0 is still in beta at the time of writing, it makes sense to refer to the new interface as the Windows 95 shell, although all the techniques described here should be applicable to Windows NT 4.0 when it's released.*

Most of the features covered in this chapter rely on a knowledge of COM and OLE, which you should have built up through the previous chapters. Throughout the rest of the chapter we'll focus in turn on the following features:

- The taskbar notification area (system tray)
- Access bars (appbars)
- File viewers
- The shell namespace
- Shortcuts (shell links)
- Shell extensions

The Notification Area

Windows 95 includes an area on the taskbar that allows an application to display an icon to show a task the application is performing. Windows 95 itself uses this area to inform the user of printing or battery status, for example, as well as displaying the time. This area is called the **taskbar notification area** (or **system tray**) and lies at the right end of the taskbar (if the taskbar is aligned horizontally) or at the bottom (if it's aligned vertically).



The icons in this area can have a tooltip which is displayed when the user moves the cursor over them. For example, when I move the cursor over my printer icon, I get a message indicating whether the printer is idle or printing. An application provides the icon, as well as the string used for the tooltip. In return for providing an icon, an application can receive notification messages when the mouse moves or clicks over the notification area.

To perform the notification magic, the application calls a special function, called **`Shell_NotifyIcon()`**. (You'll need **`#include <shlobj.h>`** to use this.) The function accepts two parameters. The first is a message that indicates what the application wants to do to the notification area. You can indicate whether you want to add, modify, or delete an icon by specifying either **`NIM_ADD`**, **`NIM_MODIFY`** or **`NIM_DELETE`**.

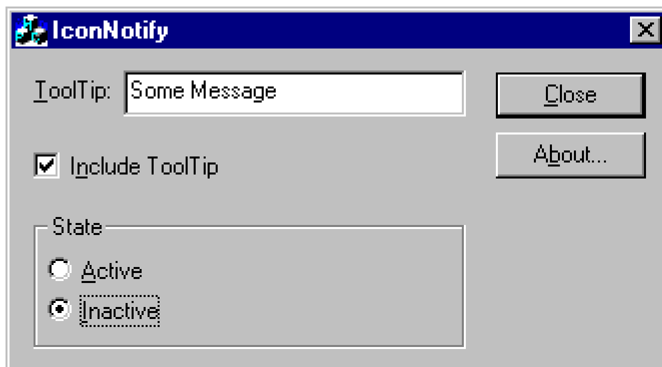
The second parameter is a pointer to a structure, called **NOTIFYICONDATA**, that the application must fill before passing it to the function. The **NOTIFYICONDATA** structure contains the following fields:

Data Member	Description
cbSize	This should be set to **sizeof(NOTIFYICONDATA)** by the caller.
hWnd	Set this to be the handle of the window that you want to receive notification messages associated with the icon.
uID	Set this to associate the icon with an application-defined identifier which may be passed back as the **wParam** parameter of the notification message.
uFlags	Flags that indicate which of the other structure members contain valid data. Can be a combination of **NIF_ICON**, **NIF_MESSAGE**, **NIF_TIP**.
uCallbackMessage	Application-defined message identifier. The operating system sends this identifier as the message parameter to the window procedure associated with the **hWnd** every time a notification is sent.
hIcon	The handle of the icon to add, modify or delete.
szTip	The tooltip text to display for the icon.

The window procedure is called when a mouse event occurs. The system sends the message to the window procedure identified by the **hWnd**. The message parameter will contain the **uCallbackMessage** identifier and **wParam** will contain the **uID** identifier. The **lParam** will contain the message identifier such as **WM_MOUSEMOVE**.

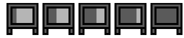
Creating a Taskbar Notification Icon

On the CD with this book, you'll find an example application, called **IconNotify**, which illustrates the steps necessary to show a notification icon on the taskbar. My dialog-based application allows you to enter a text string to be used as the tooltip for the notification icon and provides a check box to allow you to turn the tooltip support on and off. It also lets you select how the icon is displayed on the taskbar with a set of radio buttons. Here you can see the application's main window:



The **CIconNotifyDlg** class's constructor loads several icons that will be used for the notification icon.

When you select the Active radio button, it creates a timer that fires off about once a second and cycles through five different icons:



When the Inactive radio button is selected, the inactive icon is displayed:

Since the application is dialog-based, I initialize and display the notification icon in the **OnInitDialog()** function by calling a function, named **NotifyIcon()**, that I added to my dialog class:

```
BOOL CIconNotifyDlg::OnInitDialog()
{
   // other code...
   NotifyIcon(NIM_ADD, m_hInactiveIcon);
   // other code...
}
```

The **NotifyIcon()** function receives the message that will be sent to the **Shell_NotifyIcon()** function as well as the icon that should be displayed in the notification area. I first begin by declaring a **NOTIFYICONDATA** object, initializing it with the message sent to the **NotifyIcon()** function. Next, I call the **Shell_NotifyIcon()** function, sending both the message and the **NOTIFYICONDATA** object:

```
void CIconNotifyDlg::NotifyIcon(UINT uMessage, HICON hIcon)
{
   NOTIFYICONDATA nid;

   switch (uMessage)
   {
      case NIM_ADD:
         nid.uFlags = NIF_ICON | NIF_MESSAGE;
         break;

      case NIM_MODIFY:
         nid.uFlags = NIF_ICON;
         break;

      case NIM_DELETE:
         nid.uFlags = 0;
         break;
   }

   nid.cbSize = sizeof(NOTIFYICONDATA);
   nid.uID = ID_ICONNOTIFY;   // app-specific, defined in IconNotifyDlg.h
   nid.hWnd = m_hWnd;
   nid.uCallbackMessage = WM_ICONNOTIFY;   // defined in IconNotifyDlg.h
   nid.hIcon = hIcon;

   Shell_NotifyIcon(uMessage, &nid);
}
```

The next important function is **OnIncToolTip()**, which is called when the user clicks the Include Tooltip check box. We start out by retrieving the state of the check box (whether it's checked or not), which will determine whether or not we should display the tooltip. Next, we initialize a **NOTIFYICONDATA** object, stuffed with the appropriate string for the tooltip, and finally call the **Shell_NotifyIcon()** function.

```
void CIconNotifyDlg::OnIncToolTip()
{
```

```
   BOOL bIncToolTip =
                  (BOOL)((CButton*)GetDlgItem(IDC_INCTOOLTIP))->GetCheck();

   NOTIFYICONDATA nid;
   nid.cbSize = sizeof(NOTIFYICONDATA);
   nid.uID = ID_ICONNOTIFY;
   nid.uFlags = NIF_TIP;
   nid.hWnd = m_hWnd;
   if (bIncToolTip)
   {
      CString strText;
      GetDlgItem(IDC_TOOLTIP)->GetWindowText(strText);
      lstrcpyn(nid.szTip, strText, sizeof(nid.szTip));
   }
   else
   {
      strcpy(nid.szTip, _T(""));
   }

   Shell_NotifyIcon(NIM_MODIFY, &nid);
}
```

The last thing I want to discuss is the call-back function that is called when a mouse operation occurs on the notification icon. If you look at the **NotifyIcon()** function, when the message is **NIM_ADD**, it sets the flag to **NIF_ICON** and **NIF_MESSAGE**. The **NIF_MESSAGE** tells it that we are supplying an identifier, **WM_ICONNOTIFY** in the example, which will be sent as the message identifier to the **hWnd** we specify. The message is handled in an **ON_MESSAGE** handler and is sent to the **OnIconNotify()** function. We first check to make sure that the message is a right mouse button click and that the message is intended for our notification icon. If it is, I create a pop-up menu on the fly and append some dummy menu items to the menu. I then call **TrackPopupMenu()** to retrieve the menu choice from the user. The following code illustrates this:

```
LRESULT CIconNotifyDlg::OnIconNotify(WPARAM wParam, LPARAM lParam)
{
   if (lParam == WM_RBUTTONDOWN && wParam == ID_ICONNOTIFY)
   {
      CMenu popup;
      CPoint point;

      GetCursorPos(&point);
      popup.CreatePopupMenu();
      popup.AppendMenu(MF_STRING, ID_OPTION1, _T("Choice #1..."));
      popup.AppendMenu(MF_STRING, ID_OPTION2, _T("Choice #2..."));
      popup.AppendMenu(MF_STRING, ID_OPTION3, _T("Choice #3..."));

      SetForegroundWindow();
      popup.TrackPopupMenu(TPM_RIGHTALIGN | TPM_LEFTBUTTON,
                           point.x, point.y, this, NULL);

      PostMessage(WM_USER, 0, 0);
   }
   return Default();
}
```

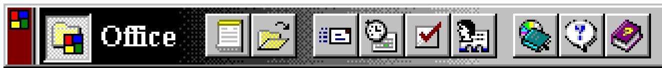
Notification icons are a great tool to use for notifying users of the status of your application, but they can also be abused if used improperly or inappropriately. Justify the use of these icons before you implement them, as nobody wants to end up with their notification area full of our icons; after all, where would they place the icons for the tasks that are currently running, which also need to share the taskbar with notification icons?

Note the strange looking SetForegroundWindow() and PostMessage() calls surrounding the call to TrackPopupMenu(). These are necessary to ensure the correct behavior of the context menu for notification icons. The call to SetForegroundWindow() is required to ensure that the menu

disappears when the user clicks anywhere beside the menu. This is caused by the behavior of Windows. The call to `PostMessage()`, on the other hand, is required by `TrackPopupMenu()`, which needs a task switch to the application that called the function shortly after it's called.

Access Bars

I remember being impressed when I first looked at the Microsoft Office desktop toolbar, which allowed easy access to the Microsoft Office applications as well as any other application you chose to add to the toolbar. Windows 95 now includes the ability to create this type of toolbar for any application you write. These are known as **desktop toolbars**, **access bars** or **appbars**. They behave similarly to the taskbar and can be docked to any side of the desktop window (or next to another access bar if one is already docked). They can even support auto-hiding and 'always on top' behavior, just like the taskbar. A desktop toolbar can also be undocked and displayed as a palette window.



Although access bars can be put to any use and are ideal for anything that requires a permanent presence on the screen, the main reason access bars exist today is to execute applications, especially when they are part of a suite of applications from one vendor. Showing an icon for each application and allowing your users to execute applications from your access bars is a great convenience for them.

Just as you do with notification icons, you should review the situation thoroughly before implementing an access bar. Remember that they take up screen real estate, and if everyone implemented an access bar, there would be no screen real estate left for our applications to run in. However, if you *do* decide to implement one anyway, always give the user the ability to close it down directly from the access bar itself. You should also consider providing choices for resizing and moving the access bar.

Enough of the theory. Lets find out how to implement an access bar with some source code.

Creating an Access Bar

On the CD that comes with the book, you'll see that I've provided a sample application to demonstrate how to create an access bar (imaginatively titled **`AccessBar`**). The most important function we need to use is the shell function, called **`SHAppBarMessage()`**. The **`SHAppBarMessage()`** function receives two parameters.

The first is a message that allows the application to register an appbar with the shell, to set an appbar's size, position and state, to retrieve information about the Windows taskbar, and so on. The message that you pass will depend on what you wish to do. The following table contains a list of the messages, with a short description of each:

Messages	Description
`ABM_ACTIVE`	Notifies the shell of an access bar's activation. The access bar should send this message when ever it

	receives the **WM_ACTIVATE** message.
ABM_GETAUTOHIDEBAR	Retrieves the handle of the autohide access bar associated with a given edge of the screen.
ABM_GETSTATE	Retrieves the autohide and always-on-top states of the shell's taskbar.
ABM_GETTASKBARPOS	Retrieves the bounding rectangle of the shell's taskbar.
ABM_NEW	Registers a new access bar and specifies the message identifier that the shell will use when it needs to send notifications to the access bar.
ABM_QUERYPOS	Retrieves the size and position for a given access bar.
ABM_REMOVE	Unregisters an access bar (which removes the bar from the shell's list of access bars).
ABM_SETAUTOHIDEBAR	Registers the access bar with the shell as an autohide bar for which the application provides the edge of the screen to be used. You don't need to send **ABM_NEW** if you use this message.
ABM_SETPOS	Specifies the size and position in screen coordinates for a given access bar.
ABM_WINDOWPOSCHANGED	Allows the application to notify the shell of any positional changes for the access bar. Your access bar should send this message whenever it receives **WM_WINDOWPOSCHANGED**.

The second parameter to **SHAppBarMessage()** is a pointer to an **APPBARDATA** structure which contains information the shell uses to manipulate the access bar.

The **APPBARDATA** structure is made up of several members:

```
struct APPBARDATA
{
    DWORD  cbSize;           // sizeof(APPBARDATA)
    HWND   hWnd;             // handle of appbar
    UINT   uCallbackMessage; // message sent to appbar
    UINT   uEdge;            // docking edge
    RECT   rc;               // bounding rectangle of appbar
    LPARAM lParam;           // used for autohide
};
```

The first member, **cbSize**, is the size of the structure and is used for version checking the structure. The second member is an **hWnd** which identifies the access bar window that will receive all the notification messages for the access bar.

The next member is named **uCallbackMessage** and is an application-defined message which will later be used by the shell to send notifications to the access bar application. The application will receive one of several possible values as the **wParam** of this message to let the access bar know why it received the message. We'll see the actual values in a minute.

The **uEdge** member is used to determine which side of the screen to dock the access bar against. Possible values include **ABE_BOTTOM**, **ABE_LEFT**, **ABE_RIGHT**, and **ABE_TOP**. This member is used when you send the **ABE_GETAUTOHIDEBAR**, **ABM_SETAUTOHIDEBAR**, **ABM_QUERYPOS**, or **ABM_SETPOS** messages.

The next member is a **RECT** structure, named **rc**, that is used to contain the bounding rectangle of the access bar in screen coordinates. This member is used when the message is **ABM_GETTASKBARPOS**, **ABM_QUERYPOS** and **ABM_SETPOS**.

The last member is an **LPARAM**, named **lParam**. This member is used with the **ABM_SETAUTOHIDEBAR** message. If the value of **LPARAM** is **TRUE**, the access bar is registered as an **AUTOHIDE**, and **FALSE** specifies that it should be unregistered.

Appbar Notifications

Notifications are received via the **uCallbackMessage** that you provide when you first create and register an access bar. In other words, the window will receive the **uCallbackMessage** message that you specify in your application. The **wParam** will contain one of several values (described in the table below), depending on the reason for the notification. The **lParam** will sometimes contain additional information (depending on the notification received).

Notifications	Description
ABN_FULLSCREENAPP	This notifies the access bar that a full screen application is opening or closing. The access bar needs to manage its Z order at this point. The **lParam** contains **TRUE** if an application is opening or **FALSE** otherwise.
ABN_POSCHANGED	This is sent when the shell taskbar's position or size has changed. The access bar should recalculate its position and/or size.
ABN_STATECHANGE	This is sent when the state of 'always on top' or 'auto-hide' has changed for the shell's taskbar.
ABN_WINDOWARRANGE	This is received when the shell is about to cascade or tile the current application windows. It's received twice for each window rearrangement: once before the windows are arranged (**lParam** equals **TRUE**) and again once they've been arranged (**lParam** equals **FALSE**).

The access bar that I created was generated with VC++ as an MFC application. I ripped out the document/view architecture because this is one of those situations where it just doesn't make sense to use it.

Since I took out the document/view architecture (including the document template), I needed to create the main window (which is also the access bar window) myself. I did this work in the application's **InitInstance()**, as follows:

```
BOOL CAccessBarApp::InitInstance()
{
   CMainFrame* pMainWnd = new CMainFrame();
   if (pMainWnd == NULL)
```

```
      return FALSE;

   m_pMainWnd = pMainWnd;

   if (!pMainWnd->Create(NULL, _T(""), WS_POPUP | WS_DLGFRAME |
       WS_CLIPCHILDREN, CFrameWnd::rectDefault, NULL, NULL,
       WS_EX_TOOLWINDOW))
      return FALSE;

   pMainWnd->ShowWindow(SW_SHOW);

   return TRUE;
}
```

First, I create the main window based on the **CMainFrame** class that AppWizard generated for me. Next, I perform the second step of the two step process (necessary for MFC windows) which is to call the **Create()** function. Notice that I use the extended style **WS_EX_TOOLWINDOW**. This style keeps my access bar from showing up in the task list. Finally, I show the window.

Now let's look at the **CMainFrame** window, which is where most of the work takes place. I needed to specify a handler for the **WM_CREATE** message so that the **CMainFrame** object could register itself as an access bar with the shell. I also took advantage of this opportunity to create several buttons on the access bar. Here's the code:

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
   if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
      return -1;

   // Add ourselves as an access bar.
   APPBARDATA abd;
   abd.cbSize = sizeof(APPBARDATA);
   abd.hWnd = GetSafeHwnd();
   abd.uCallbackMessage = WM_ACCESSBAR;
   if (!SHAppBarMessage(ABM_NEW, &abd))
      return -1;

   // Add five buttons to our access bar.
   m_button1.Create("&Right", BS_PUSHBUTTON | BS_CENTER | WS_VISIBLE |
                WS_CHILD, CRect(0,0,0,0), this, IDC_RIGHT);
   m_button2.Create("&Left", BS_PUSHBUTTON | BS_CENTER | WS_VISIBLE |
                WS_CHILD, CRect(0,0,0,0), this, IDC_LEFT);
   m_button3.Create("&Top", BS_PUSHBUTTON | BS_CENTER | WS_VISIBLE |
                WS_CHILD, CRect(0,0,0,0), this, IDC_TOP);
   m_button4.Create("&Bottom", BS_PUSHBUTTON | BS_CENTER | WS_VISIBLE |
                WS_CHILD, CRect(0,0,0,0), this, IDC_BOTTOM);
   m_button5.Create("&Close", BS_PUSHBUTTON | BS_CENTER | WS_VISIBLE |
                WS_CHILD, CRect(0,0,0,0), this, IDC_CLOSE);

   // Set initial position to the top.
   CalcPosition(m_uEdge = ABE_TOP);

   return 0;
}
```

I use the **ABM_NEW** message to register the access bar with the shell. Once I've created the button, I call a function, **CalcPosition()**, which calculates and sets the position and size of the access bar and its buttons. I use a member, called **m_uEdge**, to retain the edge of the screen that the access bar is docked against. Several functions need access to this piece of information, so I decided to store it in a data member. **CalcPosition()** is also called from several places, for example, whenever I receive a **ABN_POSCHANGED** message, I call **CalcPosition()** to reset the position and/or size of the access bar. The source code for **CalcPosition()** is quite simple:

```
void CMainFrame::CalcPosition(UINT uEdge)
{
   APPBARDATA abd;
   abd.cbSize = sizeof(APPBARDATA);
   abd.hWnd = GetSafeHwnd();
   abd.uEdge = uEdge;

   // Default to the whole screen size.
   CRect rc(0, 0, GetSystemMetrics(SM_CXSCREEN),
            GetSystemMetrics(SM_CYSCREEN));
   abd.rc = rc;

   // Call ABM_QUERYPOS to alter the position if
   // necessary.

   SHAppBarMessage(ABM_QUERYPOS, &abd);

   // Set the size.
   switch (uEdge)
   {
      case ABE_TOP:
         abd.rc.bottom = abd.rc.top + DEF_BAR_HEIGHT;
         break;

      case ABE_BOTTOM:
         abd.rc.top = abd.rc.bottom - DEF_BAR_HEIGHT;
         break;

      case ABE_LEFT:
         abd.rc.right = abd.rc.left + DEF_BAR_WIDTH;
         break;

      case ABE_RIGHT:
         abd.rc.left = abd.rc.right - DEF_BAR_WIDTH;
         break;
   }
   SHAppBarMessage(ABM_SETPOS, &abd);

   // Reposition the access bar with the newly calculated coordinates.
   // The system will not do this for you.
   MoveWindow(&abd.rc);

   // Recalc the child buttons.
   CalcChildren(uEdge);
}
```

When we're calculating the position and size of an access bar, it gets a little hairy, because there may be other access bars already docked to the edges of the desktop. However, I can use the **APPBARDATA** structure to tell the shell which side I'd like my access bar to be docked and pass a default **rect** into which to fit the access bar.

If I send the shell the **ABM_QUERYPOS** message, the shell will adjust the position of the rectangle in the **APPBARDATA** structure, but I need to refine the size of the access bar, depending on the edge that it's being docked to. If the access bar is being docked to the top, I need to determine the bottom edge. If it's being docked to the bottom, I need to determine top edge, and so on. Once I've calculated the final position and size for the access bar, I send the **ABM_SETPOS** message and move the window myself with a call to **MoveWindow()**. (When the **ABM_SETPOS** message that my application sends has completed, my access bar will receive a **ABN_POSCHANGED** message.)

Finally, I call **CalcChildren()**, which simply recalculates the positions and sizes for the buttons on the access bar.

```
void CMainFrame::CalcChildren(UINT uEdge)
{
```

```
    CRect rc;
    if (uEdge == ABE_TOP || uEdge == ABE_BOTTOM)
    {
       rc = CRect(10, 0, BUTTON_WIDTH, BUTTON_HEIGHT);
       m_button1.MoveWindow(rc);
       rc.OffsetRect(BUTTON_WIDTH * 2, 0); m_button2.MoveWindow(rc);
       rc.OffsetRect(BUTTON_WIDTH * 2, 0); m_button3.MoveWindow(rc);
       rc.OffsetRect(BUTTON_WIDTH * 2, 0); m_button4.MoveWindow(rc);
       rc.OffsetRect(BUTTON_WIDTH * 2, 0); m_button5.MoveWindow(rc);
    }
    else
    {
       rc = CRect(0, 10, BUTTON_WIDTH, BUTTON_HEIGHT + 10);
       m_button1.MoveWindow(rc);
       rc.OffsetRect(0, BUTTON_HEIGHT * 2); m_button2.MoveWindow(rc);
       rc.OffsetRect(0, BUTTON_HEIGHT * 2); m_button3.MoveWindow(rc);
       rc.OffsetRect(0, BUTTON_HEIGHT * 2); m_button4.MoveWindow(rc);
       rc.OffsetRect(0, BUTTON_HEIGHT * 2); m_button5.MoveWindow(rc);
    }
}
```

Notifications from the access bar arrive as a message. The message ID is the one that I provided when I registered the access bar. Using MFC, I defined a message map entry with the **ON_MESSAGE()** handler. I then coded a function, called **OnAccessBar()**, which receives the **WPARAM** and the **LPARAM** for the message:

```
LRESULT CMainFrame::OnAccessBar(WPARAM wParam, LPARAM lParam)
{
   UINT uState;
   APPBARDATA abd;

   abd.cbSize = sizeof(APPBARDATA);
   abd.hWnd = GetSafeHwnd();

   switch (wParam)
   {
      case ABN_STATECHANGE:
         uState = SHAppBarMessage(ABM_GETSTATE, &abd);
         SetWindowPos((ABS_ALWAYSONTOP & uState) ? &wndTopMost :
                      &wndBottom, 0, 0, 0, 0, SWP_NOMOVE | SWP_NOSIZE |
                      SWP_NOACTIVATE);
         break;

      case ABN_FULLSCREENAPP:
         uState = SHAppBarMessage(ABM_GETSTATE, &abd);
         if (lParam)    // Is it opening a Full-Screen App?
         {
            SetWindowPos((ABS_ALWAYSONTOP & uState) ? &wndTopMost :
                         &wndBottom, 0, 0, 0, 0, SWP_NOMOVE | SWP_NOSIZE |
                         SWP_NOACTIVATE);
         }
         else
         {
            // Retain old position.
            if (uState & ABS_ALWAYSONTOP)
               SetWindowPos(&wndTopMost, 0, 0, 0, 0, SWP_NOMOVE |
                       SWP_NOSIZE | SWP_NOACTIVATE);
         }

      case ABN_POSCHANGED:    // Pass through from above.
         CalcPosition(m_uEdge);
         break;
   }

   return 0;
}
```

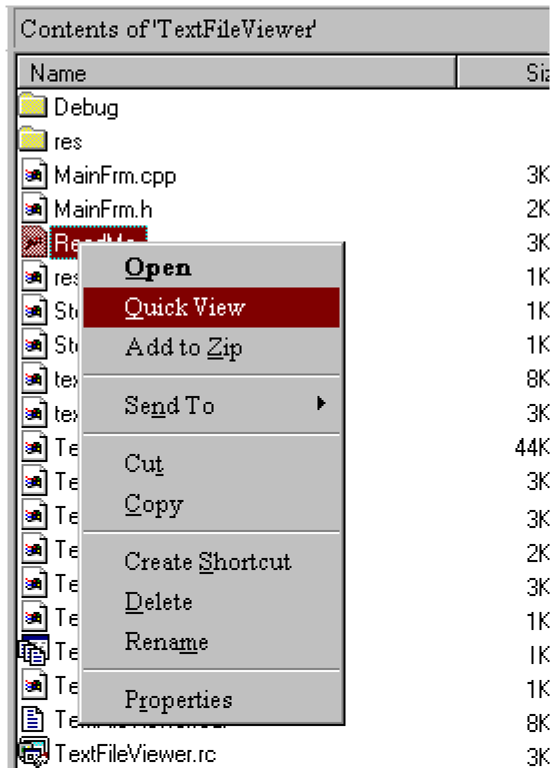
This code actually doesn't perform any magic. I simply respond to the notifications by performing the necessary code, as I specified above in my explanations of the notifications.

The buttons on the access bar control the edge that the access bar is docked against. I wrote MFC message map handlers for each button and simply reset the **`m_uEdge`** data member and call **`CalcPosition()`** to reposition the access bar.

That's the complete implementation of my access bar. You'll note that I didn't include code for the autohide feature, because this will vary between implementations. It's relatively simple to implement, but the shell doesn't manage this state for you; you must do it in your own code. The shell will send you a notification to alert you that the access bar's state has changed, but it's up to you to perform the actual showing or hiding of the bar. That's all there is to it. Nothing more, nothing less. With a few lines of code and one or two hours, you too can have an access bar up and running.

File Viewers

With Windows 95, Microsoft introduced a feature to allow you to quickly view files of any type. From the shell, you can right-click on a file to bring up a context menu on which you'll usually find an item, labeled Quick View. Selecting this item causes the shell to load a file viewer that is appropriate for the file and allows the viewer to display the file in its document window.

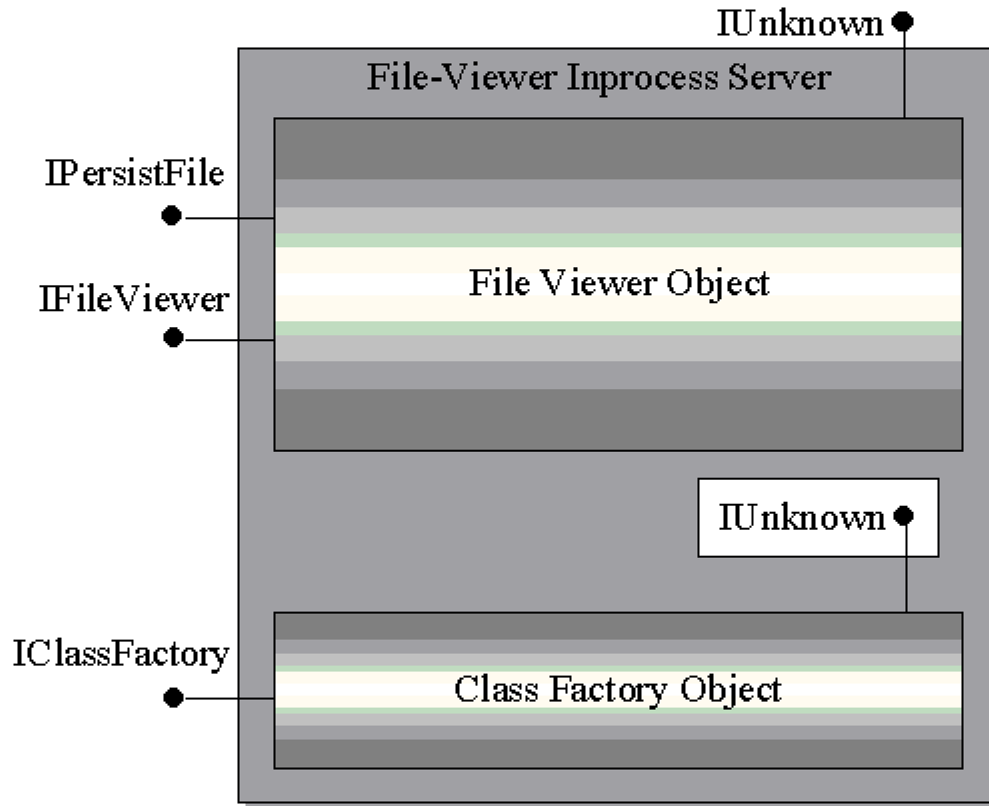


The greatest benefit that a file viewer offers is that the user doesn't have to load a fully-fledged application in order to see what a file contains. As a programmer, you can simply offer the drawing portion of the code in a file viewer and the users can take advantage of the reduced load times when they only need to view your files without editing them.

Keep in mind that the shell must know something about the file before it can find an appropriate viewer. This information is kept in the registry (we'll cover the necessary entries later in this section) and can either be based on a class ID (a way of identifying OLE objects) or a file extension. Later in the chapter, I'll demonstrate how to build a file viewer and associate it with a file extension.

OLE and File Viewers

OLE plays a major role when you're using file viewers, since file viewers are actually OLE component objects which expose several interfaces (as you can see from the diagram):



Since an in-process server is nothing more than an OLE-aware DLL, it needs an application with its own process space to host it. That's where the **`Quikview.exe`** application located in the **`\System\Viewers`** subdirectory of your Windows directory comes in. The shell actually loads this application and passes it several options, including the full path of the file the user selects for quick viewing (we'll discuss the Quick View application in more depth later in the chapter).

A file viewer must begin by exposing and registering the class factory object with OLE as soon as it's loaded. The class factory object will be told, via the **`IClassFactory`** interface, to create file viewer objects when they're needed .

The file viewer object exposes two interfaces, the first of which is used for passing the name of the document file and is called **`IPersistFile`**. The second interface is used for displaying the file in its document window and is called **`IFileViewer`**. We'll discuss both of these further in the next section.

You might be asking yourself, " Why did Microsoft implement this file viewer stuff as OLE objects?" One of the main reasons is extensibility. For example, if Microsoft wanted to provide new functionality in file viewer objects, all they'd have to do is provide a new common interface. You could implement and expose this interface and your file viewer would be in business with some new functionality. However, to support older file viewers, the system can use **`QueryInterface()`** to determine whether a file viewer supports the new interface. If it doesn't, it can simply use the old interfaces. That's the beauty of OLE; the power of extension at run time.

One more thing. Remember I mentioned that a file viewer can be associated with a class ID or a file extension? The shell actually first checks to see whether the document file is an OLE compound file. If it

is, it attempts to read the statistical information from the file (which is stored using the OLE compound file functions from the application that originally created the file; for more information on these functions, see the OLE reference or your online help) and uses the class ID stored with the statistical information to find and load the file viewer. If the file is not a compound file, it looks at the file's extension and attempts to find a file viewer associated with that extension. All of this information is kept in the registry. Later, we'll discuss the exact entries that you must provide to make all this file viewer stuff work.

IPersistFile

As you probably know, the **IPersistFile** interface usually provides an object with the ability for it to load and save itself to a disk file in any manner that the object chooses. **IPersistFile** has five methods (besides the usual **IUnknown** members, for which we'll get MFC to do most of the work for us), but a file viewer is actually only required to implement three of these. The other member functions can return **E_NOTIMPL** (which tells any callers that the function was not implemented). The three member functions that we must provide are **Load()**, **GetCurFile()**, and **GetClassID()**.

The **Load()** member function will be called and passed a path to the document file that the file viewer will need to display.

The **GetCurFile()** function is required to allocate space and return a copy of the file name in the allocated space.

The **GetClassID()** function must return the class ID of the file viewer. This is the same class ID that is associated with the class factory registered with OLE.

IFileViewer

The Quick View application calls the members of the **IFileViewer** interface to tell the File Viewer object when to display its user interface, display the file, or print the file. Besides the usual **IUnknown** members, **IFileViewer** contains three other functions: **ShowInitialize()**, **Show()** and **PrintTo()**.

The **ShowInitialize()** function is the first **IFileViewer** function to be called. The file viewer object should perform any creations, allocations, or loading (that includes loading the file to be viewed) . Don't be misled by the name. If you thought that the **IPersistFile::Load()** function is called when the document file needs to be loaded, you'd be wrong. The **IPersistFile::Load()** function should only store the file name; it shouldn't load the file. That's the responsibility of **ShowInitialize()**.

Don't place any initialization code that is likely to generate errors (for example, memory allocations) in the **IPersistFile::Load()** function. Rather, you should place it in the **ShowInitialize()** function. This function can return several predefined error values, all starting with **FV_E_**. Here you can see the full list:

File Viewer Errors

FV_E_BADFILE

FV_E_EMPTYFILE

FV_E_FILEOPENFAILED

FV_E_INVALIDID

FV_E_MISSINGFILES

FV_E_NOFILTER

FV_E_NONSUPPORTEDTYPE

FV_E_NOVIEWER

FV_E_OUTOFMEMORY

FV_E_PROTECTEDFILE

FV_E_UNEXPECTED

The **Show()** member function is called to initially display the main window and the content of the document file. This function receives a flag, indicating how the window should be displayed. The flag can be one of the standard **ShowWindow()** flags. (For a list of these flags, see the reference for the **ShowWindow()** API function.) The **Show()** function shouldn't return until the user has closed the main window. In other words, the function should enter a message loop and continue processing user input until it receives a **WM_QUIT** message. As you'll see a little later, handling the message loop in an MFC application requires a bit of thought, but there are no insurmountable problems.

PrintTo() is similar to **Show()** in that it mustn't return until printing has finished. The function receives a flag to determine whether or not the user interface of the file viewer should be displayed. This function will most likely get called instead of **Show()**, since **Show()** is used to display the file and **PrintTo()** is used to print the file.

In the example, I also implemented this interface inside my document class. Again, the source code and a detailed description follow later in this chapter.

Telling the System about a File Viewer

When you install your file viewer, there are several entries that must exist in the registry in order for the shell to locate it. You can enter these from your setup application, or you can provide a registration file for your user to merge with the existing registry. I chose to provide a registration file with my sample application.

First of all, the entries need to be in the **HKEY_CLASSES_ROOT** key of the system registry. You'll need to operate on two subkeys of the main **HKEY_CLASSES_ROOT** key. The first subkey is **\QuickView** and the other is **\CLSID**. This is the structure that you must implement:

```
HKEY_CLASSES_ROOT
   \QuickView
      \<extension> = <human-readable document type>
         \{<CLSID>} = <human-readable viewer name>
         \{<CLSID>} = <human-readable viewer name>
         \{<CLSID>} = <human-readable viewer name>

HKEY_CLASSES_ROOT
   \CLSID
      \{<CLSID>} = <human-readable viewer name>
         \InprocServer32 = <full path to file viewer DLL>
                         = ThreadingModel = "Apartment"
```

Entry	Description
HKEY_CLASSES_ROOT	One of several permanently open root keys in the system registry.
QuickView	The subkey where file extensions are related to their

	associated file viewers' CLSIDs.
CLSID	The subkey that contains all of the registered OLE component object class identifiers. These values are 128-bit values surrounded by curly braces. The associated file viewer's path is stored under these keys.
<*CLSID*>	The CLSIDs represent the viewer in-process server.
<*Human-readable document type*>	The text used for displaying a description of the CLSID or file extension to the user .
<*Human-readable viewer name*>	The text used for displaying to the user a description of the file viewer.
<*extension*>	The file extension prefixed with a period.

Note that the system allows for several file viewers for a particular extension.

I used a registration file to register my file viewer with the system. With a registration file, you either double-click on the file from Explorer or you run the **Regedit.exe** application and load the registration file into RegEdit. Here's the text that I included in my registration file:

```
REGEDIT4

[HKEY_CLASSES_ROOT\CLSID\{1F420CA7-3C7B-11CF-97E6-444553540000}]
@="Wrox File Viewer"
[HKEY_CLASSES_ROOT\CLSID\{1F420CA7-3C7B-11CF-97E6-444553540000}\InprocServer32]
@="c:\\msdev\\projects\\wroxfileviewer\\debug\\wroxfileviewer.dll"
"ThreadingModel" = "Apartment"

[HKEY_CLASSES_ROOT\QuickView\.wrx]
@="wrxfile"
[HKEY_CLASSES_ROOT\QuickView\.wrx\{1F420CA7-3C7B-11CF-97E6-444553540000}]
@="Wrox File Viewer"

[HKEY_CLASSES_ROOT\wrxfile\CLSID]
@="{1F420CA7-3C7B-11CF-97E6-444553540000}"
```

Let's start at the top. The **REGEDIT4** command tells the RegEdit application what version of commands to expect in this file. **REGEDIT4** is the latest set of registration file commands available. The text enclosed in brackets ([]) describes keys in the registry, while the strings beneath the keys describe values that appear under those keys in the registry. The name of the value is separated from its data by an equals sign. The at symbol (@) means that the data following the equals sign belongs to the default value of that key.

In the example, we've basically defined a file viewer for text files, but we've used the file extension **.wrx** to be associated with our viewer so that testing this example will have minimal effect on your system. If you want to test the example, you can simply merge the supplied **.reg** file with your registry, create a new text file and change its extension to **.wrx**, then select Quick View from the file's context menu. The viewer's path is part of the **.reg** file ,so you may need to change that to reflect the location of the file viewer on your own system.

The **InprocServer32** entry means that the module is a 32-bit server and the **ThreadingModel** key specifies which OLE threading model is used. An apartment model-aware application must have thread-safe entry points such as **DllGetClassObject()** or **DllCanUnloadNow()** (which we'll learn more of later on). The apartment model means that OLE blocking is made at the class factory level. Each object created

by a class factory falls within the same apartment. If two clients make calls to objects in the same apartment, one of the clients is blocked until the other client receives its return value.

In Windows 95 and NT 3.51, the threading model was basically always the apartment model, but later versions of Windows (NT 4.0 and later) will allow for different models, such as **free threading**, which will allow more flexible management of OLE objects. However, this is getting off the track a bit. The bottom line is that MFC handles this for us, so we really have nothing to worry about. Let's move on to writing code to interact with the Quick View application.

The Quick View Application

When the user selects a file to view, the system spawns an application called Quick View (`\Windows\System\Viewers\Quikview.exe`. Note that there's no `c` in the file name). The system passes the full path of the file to the Quick View application, along with some options to enable the application to determine whether it's performing a print operation or just viewing the file. The Quick View application simply acts as a stub for the in-process servers and has no message loop or interface of its own. If the Quick View application can't find an appropriate file viewer for the selected file, it will ask whether or not the user wants to load the file into the default file viewer, which treats the file as a hex dump (it looks fine if the file is made of plain ASCII text).

Since the Quick View application has no message loop of its own, it relies on the file viewer in-process servers to provide the message loop and the user interface.

The Quick View application takes the following steps in loading and interacting with the associated File Viewer:

- It finds the file viewer entries in the registry.
- After gathering the CLSID for the file viewer, it attempts to create a file viewer object, using that CLSID by calling `CoCreateInstance()` (probably asking for `IUnknown`).
- It retrieves pointers to the `IPersistFile` interface and the `IFileViewer` interface.
- It calls the `IPersistFile::Load()` function.
- It calls the `IFileViewer::ShowInitialize()` function.
- It calls the `IFileViewer::Show()` or `IFileViewer::PrintTo()` function to display or print the file.
- It waits until either `IFileViewer::Show()` or `IFileViewer::PrintTo()` return.
- When the Quick View application regains control, it releases two of the three interfaces immediately. A short time later it releases the final interface on the file viewer object.

The last interface is not released until a few minutes later for performance reasons. Think about it for a minute; if you load a file to perform a quick view on it, chances are that you're looking for a specific file. If the first file you load is not the one you want, you'll try to view another file, and so on down the line until you've exhausted all of the files or you find the one that you are looking for. In any event, it would be very expensive (performance-wise) to load and unload a module in memory each time you want to view a different file of the same type. This is why the same file viewer object is held on to for a few minutes.

When the user has stopped selecting files to view and a few minutes have gone by without any interaction, the Quick View application releases the file viewer object (letting go of the final interface). This causes the in-process server to decrement its object count to zero. When its `DllCanUnloadNow()` function is called, it returns `TRUE`, which causes the in-process server to be unloaded from memory.

You can find the sample application on our CD as `WroxFileViewer`. To run the sample you'll need to edit the `WroxFileViewer.reg` file so that the path to the viewer is the same as on your system. Then merge the file with your registry as we discussed earlier.

Creating a File Viewer

Before I set out to write my file viewer, I knew that I wanted to use MFC because it always offers me a fast path to having my application up and running, but I wanted to avoid the document/view architecture, since I also knew that performance was an issue.

When you write your own file viewers, keep in mind that the user wants to load and view the document as quickly as possible, so you shouldn't spend too much time in your load and initialization code. For this reason, things like writing several document viewers into a single server is a bad idea, since the server will take that much longer to load.

Generating the Skeleton Code

I started out by generating an OLE in-process MFC server (using MFC AppWizard (dll) with OLE Automation support selected). This provides all the necessary entry points and startup code. The automation support ensures that the correct header files are included and my module is initialized correctly for OLE.

Next, I added a **`CMainFrame`** class with toolbar and status bar and added some code to it to create an edit control to hold the text to display. Since a file viewer provides the user interface for the Quick View application, it's a good idea to follow the standards set by the other file viewers on your system. The standards for the interface are fully described in the documentation supplied with Visual C++.

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
   if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
      return -1;

   // Create the edit control.
   CRect rc;
   GetClientRect(rc);
   if (!m_wndEditCtrl.Create(WS_HSCROLL | WS_VSCROLL |
      ES_AUTOHSCROLL | ES_AUTOVSCROLL |
      ES_MULTILINE | ES_NOHIDESEL, rc, this, ID_EDITCTRL))
   {
      TRACE0("Failed to create edit control\n");
      return -1;      // fail to create
   }
   m_wndEditCtrl.ShowWindow(SW_NORMAL);

// ... Create toolbar and status bar here

   // Allow drag and drop from Explorer
   DragAcceptFiles();

   return 0;
}
```

I also added a class derived from **`CCmdTarget`** called **`CFileViewer`**. This will provide the implementation of the **`IFileViewer`** and **`IPersistFile`** interfaces.

Providing OLE's Entry Code

An in-process server must export two global functions: **`DllGetClassObject()`** and **`DllCanUnloadNow()`**. **`DllGetClassObject()`** is called sometime after the in-process server is loaded. Its job is to return a class factory object (actually the requested interface on a class factory object; most likely **`IClassFactory`**). Once the Quick View application grabs hold of the class factory, it can tell it to create a file viewer object for loading and displaying the document file.

Since our application is an MFC application, we can rely on MFC to do most of the work for us. My implementation of the **`DllGetClassObject()`** function is exactly the one provided by MFC AppWizard (dll):

```
STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID* ppv)
{
   AFX_MANAGE_STATE(AfxGetStaticModuleState());
   return AfxDllGetClassObject(rclsid, riid, ppv);
}
```

The other exported function is called whenever **`CoFreeUnusedLibraries()`** is called. This function calls the **`DllCanUnloadNow()`** function of all in-process servers in memory, and if they return **`TRUE`**, the server is unloaded from memory. **`DllCanUnloadNow()`** returns **`TRUE`** when it's no longer servicing any objects in memory, or if it's not locked in memory (via the class factory object). Again, I simply use the MFC implementation, which already handles the class factory, object counts and locking support. The function looks like this:

```
STDAPI DllCanUnloadNow(void)
{
   AFX_MANAGE_STATE(AfxGetStaticModuleState());
   return AfxDllCanUnloadNow();
}
```

Why We Need to Override CWinApp::Run()

You normally never have any need to override the **`CWinApp::Run()`** function, but I'm not a normal person (or so I've been told), so here's a situation where I had to override the function. Remember I mentioned that the **`Show()`** or **`PrintTo()`** functions had to provide a message loop and they shouldn't return until the user has chosen to close the application? How do we provide that functionality?

I knew that MFC implements the message loop in a function, called **`CWinApp::Run()`**, which is called after all the MFC window classes have been registered and the instance has been initialized (with a call to **`CWinApp::InitInstance()`**). To make sure that the message loop isn't executed before I want it to be (i.e. before **`IFileViewer::Show()`** has been called), I added some code to stop it from executing prematurely. This is purely defensive coding.

I did this by placing a **`static`** data member, **`m_bShowCalled`**, in the COM object class, **`CFileViewer`**, that I could use to determine whether **`IFileViewer::Show()`** has been called yet. When the framework calls the application's **`Run()`** function, it will be handled by my **`CWroxFileViewerApp::Run()`** function, which overrides **`CWinApp::Run()`**. I then check to see whether the **`IFileViewer::Show()`** function has been called yet. If it hasn't, I simply return from the function. Later, when **`IFileViewer::Show()`** is called, it will set **`m_bShowCalled`** to **`TRUE`** and will call **`CWinApp::Run()`** once again. At that point, I'll delegate to the **`CWinApp::Run()`** function, which begins the message loop.

```
int CWroxFileViewerApp::Run()
{
   // Don't do anything in here until the main window
```

```
   // and document have been created. IFileViewer::Show()
   // will call this function.

   if (CFileViewer::m_bShowCalled)
      return CWinApp::Run();

   return 0;
}
```

Implementing IPersistFile

Implementing the **IPersistFile** member functions was a cinch. I needed to provide the declarations in the header file (using the MFC macros as discussed in Tech Note 38 that can be found in the Visual C++ Books Online), delegate the **IUnknown** members to MFC and implement functionality for the **GetCurFile()**, **Load()** and **GetClassID()** members.

First thing's first. The declaration, which sits inside the declaration for my file viewer class, looks like this:

```
//IPersistFile interface
    BEGIN_INTERFACE_PART(PersistFile, IPersistFile)
      //IPersistFile methods
       STDMETHODIMP IsDirty();
       STDMETHODIMP Load(LPCOLESTR pszFileName,DWORD dwMode);
       STDMETHODIMP Save(LPCOLESTR pszFileName,BOOL fRemember);
       STDMETHODIMP SaveCompleted(LPCOLESTR pszFileName);
       STDMETHODIMP GetCurFile(LPOLESTR __RPC_FAR *ppszFileName);
       STDMETHODIMP GetClassID(LPCLSID pClsID);
    END_INTERFACE_PART(PersistFile)
```

Delegating to the MFC version of **IUnknown** is easy. I simply use the **METHOD_PROLOGUE** macro to get the pointer to the document object as **pThis**, then use it to call its **IUnknown** members:

```
STDMETHODIMP CFileViewer::XPersistFile::QueryInterface(REFIID riid,
   void** ppv)
{
   METHOD_PROLOGUE(CFileViewer, PersistFile);
   return pThis->ExternalQueryInterface(&riid, ppv);
}
```

I do the same thing for **AddRef()** and **Release()**. Next I provided the code for **GetCurFile()**, **Load()** and **GetClassID()**:

```
STDMETHODIMP CFileViewer::XPersistFile::GetCurFile(LPOLESTR __RPC_FAR
    *ppszFileName)
{
   USES_CONVERSION;
   ULONG cb;

   METHOD_PROLOGUE(CFileViewer, PersistFile);

   if (!pThis->m_bLoadCalled)
      return E_UNEXPECTED;

   if (ppszFileName == NULL)
      return E_INVALIDARG;

   // Allocate the memory for the string.
   cb = (pThis->m_strFilePath.GetLength() + 1) * sizeof(OLECHAR);
   *ppszFileName = (LPOLESTR)CoTaskMemAlloc(cb);
   if (*ppszFileName == NULL)
      return E_OUTOFMEMORY;
```

```
   memcpy(*ppszFileName, T2OLE(pThis->m_strFilePath), cb);

   return NOERROR;
}

STDMETHODIMP CFileViewer::XPersistFile::GetClassID(LPCLSID pClsID)
{
   METHOD_PROLOGUE(CFileViewer, PersistFile);

   *pClsID = clsid;
   return NOERROR;
}

STDMETHODIMP CFileViewer::XPersistFile::Load(LPCOLESTR pszFileName,
   DWORD dwMode)
{
   USES_CONVERSION;
   METHOD_PROLOGUE(CFileViewer, PersistFile);

   if (pszFileName == NULL)
      return E_INVALIDARG;

   pThis->m_strFilePath = OLE2T(pszFileName);

   // Remember that this function has been called.
   pThis->m_bLoadCalled = TRUE;

   return NOERROR;
}
```

These are relatively simple functions. As for the other members, I return **E_NOTIMPL**, which means that I didn't provide any implementation for the function. It's always a good idea to at least return this value, since someone might call the function and blow up if you don't provide it.

Implementing IFileViewer

The **IUnknown** members for **IFileViewer** delegate to the document's **IUnknown** functions like those in the **IPersistFile** interface, so the only members I needed to implement with some real working code were **ShowInitialize()**, **Show()** and **PrintTo()**. I simply return **E_NOTIMPL** from **PrintTo()**, so that brings us down to the two most important member functions of all.

Let's begin with **ShowInitialize()**. The function's one and only parameter is a pointer to an **IFileViewerSite** interface which the Quick View application sends it. I hold on to it and increment its reference count. Next, I create the frame window and load the file into my edit control by calling the **Serialize()** function in the file viewer class. This just delegates the call to **CMainFrame::Serialize()** which does the hard work of loading the file into the edit control.

```
STDMETHODIMP CFileViewer::XFileViewer::ShowInitialize(LPFILEVIEWERSITE
   lpfvs)
{
   TRACE0("Entered ShowInitialize()\n");

   METHOD_PROLOGUE(CFileViewer, FileViewer);

   HRESULT hr;

   // Be sure that you have the file viewer.
   if (pThis->m_lpfvs != lpfvs)
   {
      pThis->m_lpfvs = lpfvs;
      pThis->m_lpfvs->AddRef();
   }
```

```
   // Default error code
   hr = E_OUTOFMEMORY;

   // Load a file.
   if (pThis->m_strFilePath.IsEmpty())
      return E_UNEXPECTED;

   // Create the window.
   pThis->CreateFrameWindow();

   CFile file(pThis->m_strFilePath, CFile::modeRead);
   CArchive ar(&file, CArchive::load);
   pThis->Serialize(ar);

   //Tell IFileViewer::Show it's OK to call it.
   pThis->m_bShowInitCalled = TRUE;

   return NOERROR;
}
```

The **`IFileViewerSite`** interface offers two functions that can be called by our application: **`GetPinnedWindow()`** and **`SetPinnedWindow()`**. These are used to get and set the handle of the **pinned window**. Users can set a pinned window so that when they quick view a new file, the existing pinned window is 'reused' rather than a new window opening up.

In fact, the window isn't really reused; the new window simply uses **`GetPinnedWindow()`** to determine whether it needs to position itself in the same area as an existing pinned window. Once it's done this, the original window closes. Information about the position of any existing pinned window is passed to the file viewer by Quick View in the parameter to the **`IFileViewer::Show()`** function.

`IFileViewer::Show()` needs to show the user interface and begin the message loop. It receives a pointer to a **`FVSHOWINFO`** structure provided by the Quick View application which I hold onto until I'm done with it. This structure contains a lot of important information, such as the size that the main window should be, as well as its initial show state.

```
STDMETHODIMP CFileViewer::XFileViewer::Show(LPFVSHOWINFO pvsi)
{
   TRACE0("Entered Show()\n");
   METHOD_PROLOGUE(CFileViewer, FileViewer);

   if (!pThis->m_bShowInitCalled)
        return E_UNEXPECTED;

   pThis->m_pvsi = pvsi;
   CWnd* pWnd = pThis->GetFrameWindow();

   if ((pThis->m_pvsi->dwFlags & FVSIF_NEWFAILED) == 0)
   {
      if (pThis->m_pvsi->dwFlags & FVSIF_RECT)
         pWnd->MoveWindow(&pThis->m_pvsi->rect);
      pWnd->ShowWindow(pThis->m_pvsi->iShow);
      pWnd->SetWindowText(pThis->m_strFilePath);

      if (pThis->m_pvsi->iShow != SW_HIDE)
      {
         pWnd->SetForegroundWindow();
         pWnd->UpdateWindow();
      }

      // If an old window exists, destroy it now.
      if (pThis->m_pvsi->dwFlags & FVSIF_PINNED)
      {
         pThis->m_lpfvs->SetPinnedWindow(NULL);
         pThis->m_lpfvs->SetPinnedWindow(pWnd->GetSafeHwnd());
         ((CMainFrame*)pWnd)->m_bReplace = TRUE;
```

```
        }

        if (pThis->m_pvsi->punkRel != NULL)
        {
            pThis->m_pvsi->punkRel->Release();
            pThis->m_pvsi->punkRel = NULL;
        }
    }

    m_bShowCalled = TRUE;

    // Start the message loop
    AfxGetApp()->Run();

    return NOERROR;
}
```

Pinned Windows

The tricky bit with pinned windows is what happens when a new file is quick viewed when a pinned window is set. When this happens, Quick View sends a **WM_DROPFILES** message to the pinned window with the file name of the new file to be viewed. The existing viewer should check the file to see whether it can view it itself. If it can, it should replace the existing file with the new one. If it can't, it needs to fill in the **FVSHOWINFO** structure that it got a pointer to in the **IFileViewer::Show()** function to inform Quick View of the current situation. Then it needs to shut itself down. Here's the code that I provided to handle this situation:

```
void CMainFrame::OnDropFiles(HDROP hDropInfo)
{
    TRACE0("Entered OnDropFiles()\n");

    USES_CONVERSION;
    SetActiveWindow();      // activate us first !
    ::DragQueryFile(hDropInfo, (UINT)-1, NULL, 0);

    TCHAR szFileName[_MAX_PATH];
    ::DragQueryFile(hDropInfo, 0, szFileName, _MAX_PATH);

    if (strstr(_strupr(szFileName), _T(".WRX")))
    {
        // Delegate to default file handling.
        CFile file(szFileName, CFile::modeRead);
        CArchive ar(&file, CArchive::load);
        Serialize(ar);
        SetWindowText(szFileName);
        SetForegroundWindow();
    }
    else
    {
        HWND hWnd;
        m_pComObj->m_lpfvs->GetPinnedWindow(&hWnd);
        if (hWnd == GetSafeHwnd())
            m_pComObj->m_pvsi->dwFlags |= FVSIF_PINNED;
        m_pComObj->m_pvsi->dwFlags |= FVSIF_NEWFAILED;
        m_pComObj->m_pvsi->dwFlags |= FVSIF_NEWFILE;
        m_pComObj->m_pvsi->dwFlags |= FVSIF_RECT;
        GetWindowRect(&m_pComObj->m_pvsi->rect);
        wcscpy(m_pComObj->m_pvsi->strNewFile, T2OLE(szFileName));
        DestroyWindow();
        PostQuitMessage(0);
    }

    ::DragFinish(hDropInfo);
}
```

Since our window can also be pinned when it's closed down by the user, we need some code to keep Quick View updated when that happens. Here you can see that we just set the pinned window handle to **`NULL`** if the window that's closing was pinned:

```
void CMainFrame::OnClose()
{
   TRACE0("Entered OnClose()\n");

   HWND hWnd;
   if (m_pComObj->m_lpfvs)
   {
      m_pComObj->m_lpfvs->GetPinnedWindow(&hWnd);
      if (hWnd == GetSafeHwnd())
         m_pComObj->m_lpfvs->SetPinnedWindow(NULL);
      m_bReplace = FALSE;
   }
   CFrameWnd::OnClose();
}
```

And that's how to create a file viewer. I recommend that you look at the code on the CD to get the full picture. You should find it quite easy to adapt it to create file viewers for your own file types. The most important thing to remember about file viewers is that they should be fast.

Windows File Systems

Saving files under a system that only allows eight characters for the name and three characters for the extension has always been a problem for me. Furthermore, I have never quite understood why this limitation has hammered us for so long. However, I can now see the light at the end of the tunnel, thanks to the introduction and widespread use of Windows 95 and Windows NT, which both allow long file names.

Although the file system actually sits under the Windows shell that we'll be examining in the major part of this chapter, a quick look at the way Windows handles long file names is certainly in order, since it's only fairly recently that long file names have become a mainstream feature of Windows.

Depending on the configuration of a user's machine, your application may have to deal with any one of the following file systems:

File Allocation Table file system (FAT)
Protected-Mode FAT file system (virtual FAT)
New Technology file system (NTFS)
High-Performance file system (HPFS)

We'll look at the various features of the different file systems further in the following sections.

Over 500 Pounds - the FAT File System

For years, users have had to live with a file system that is limited to short file names. This file system uses what's called a **file allocation table***,* which contains entries for each file on the storage media. It also keeps what's called a **hierarchy directory structure**, which determines how the user will view the files. The user can then create directories within directories and build a logical hierarchy using the file system.

This file system has support for hard drives as well as floppies. It's main advantage is that it's supported by various operating systems (DOS, Windows, OS/2).

The FAT file system doesn't distinguish between uppercase and lowercase letters. A directory or file name consists of any combination of up to eight letters, digits, or the following special characters:

```
$ % ' - _ @ { } ~ ` ! # ( )
```

Of all the limitations of the FAT file system, the biggest one is that file names can only be up to eight characters long with an optional three character extension. This extension has been used to associate files with applications for many years under Windows.

When is a FAT not really a FAT? When it's a VFAT

When you install NT or Windows 95, the FAT file system is set up to support long file names. The new file system is called **protected-mode file allocation table** file system or **virtual FAT** (**VFAT**). The VFAT file system is very much like the FAT file system that we've just looked at in that it supports names with eight characters and three character extensions, but in addition, it also supports file names of up to 255 characters (including extensions). So, a string will need to be 260 characters to hold a fully qualified file name. This is enough room to fit your drive letter, colon, backslash, the directory path to the file, the file name and, of course, a terminating null. VFAT also supports a few characters not supported by FAT.

When you support long file names in your application, the API that creates the file name entry for you gives the entry two names. The first is the long file name and the other is an 8.3 format file name that can be viewed from applications that don't support long file names. You don't have to do anything special; the operating system will do it all for you. The file can be accessed from either name. If you know one of these names, you can ask the system for the other by using the **`GetShortPathName()`** or **`GetFullPathName()`** API functions. Keep in mind that the short file name might change, even if the long file name stays the same. This could happen if you copy the file from one directory to another, for example.

The algorithm used to determine the short file name is as follows:

First it checks if the file name is already in standard 8.3 format. If it is, it uses that. If not, it proceeds to Step 2.

It extracts all space characters from the file name and removes all except the final period.

It then retrieves the leading six characters and appends a tilde, plus a number.

Next it takes the first three characters following the period to use as the extension.

It then converts the 8.3 file name to capital letters.

If there are any characters that are illegal in the underlying file system, it replaces them with an underscore (_).

Finally, it checks for the existence of a file with the same created name. If it finds one, it increments the number following the tilde until there's no longer a collision.

Note that when you search through names in a directory that match the criteria using the Win32 API file functions, the match might occur with the long or the short file name. In other words, the operating system will look at both names for a match. You should interrogate the **`cFileName`** and the **`cAlternateFileName`** members of the **`WIN32_FIND_DATA`** structure to determine which one caused the match.

New Technology File System (NTFS)

The New Technology File System (NTFS) is supported by NT only (when I last checked). NTFS maintains files on a hard drive, but has no support for floppies. The file system creates an object-oriented view on the files by treating the files as objects with user- and system-defined attributes. NTFS provides all the capabilities of the FAT file system, but without its many limitations.

One of the strongest features of this file system is its performance over a FAT file system. It accesses files much faster is much more robust. The structure of the system is supposed to restore itself consistently to a disk after a CPU failure, system crash, or I/O error and without you having to use disk-checking utilities. However, NTFS provides these utilities in case recovery fails or corruption occurs outside the control of the file system. However, chances are, you'll never have to use **`chkdsk`** again.

The characteristics of NTFS file names are pretty much the same as in VFAT. They can be any practical length (up to 255 characters). NTFS also creates 8.3 format file names for every file just like VFAT. Unlike VFAT, the file names are Unicode-encoded to allow different character sets across the world. This feature is implemented internally by Windows NT.

High Performance File System

The High Performance File System (HPFS) only manages files on hard drives (there is no support for floppies). HPFS is a file system that supports extended attributes and long, mixed-case file names. It also improves operating system performance by implementing several levels of caching.

NT support for HPFS is only there for backward compatibility (for example, in cases of dual boots). Windows 95 can recognize HPFS formatted disks as drives, but, unlike NT, can't access the files. When you move or copy a file from NTFS to HPFS, be aware that the characters are converted to OEM and the name becomes case-insensitive. Also, all permissions are lost. HPFS is, however, fully supported by OS/2. If you're still using OS/2 for LAN server support, you might have to interact with an HPFS file system.

Long File Name Functions

All new Win32-based applications should use only the Win32 API file name functions for the best possible performance and usage. Most Windows 3.x functions that received a file name (such as `_lopen()` or `_lcreate()`) have been updated to support long file names. The `OpenFile()` function has not been updated and Microsoft warn that you should stay away from using this function. Instead, you should use the new `CreateFile()` API function. `OpenFile()` is still around, but it's only there for backward compatibility with older applications.

One thing to consider is that a single user might be using several file systems. For example, they might have FAT and NTFS on separate partitions on their local machine, or they could be talking to a machine across the LAN, which uses a totally different file system. There will probably come a time when you want to know which file system you are utilizing in order to maximize functionality specific to that file system. A Win32 API function, called `GetVolumeInformation()`, exists for just this purpose. You pass it several pointers to some variables you allocate and it fills those places with information.

Most developers allocate a static place in memory where they store their file names (usually they use `MAX_PATH` to determine the size of the file name). You can avoid wasted memory by calling `GetVolumeInformation()` and then using the `lpMaximumComponentLength` field to allocate the space dynamically before filling the variable with the file name. I won't go into all of the details on the `GetVolumeInformation()` function, since you can simply look up the function online and view its parameters.

Long File Name Gotchas

The following items are things that you should look out for when you write your applications. These are things that I have come across, but they're certainly things that you should also adhere to:

- An installation program that needs to enter information into configuration files, such as `Config.sys`, should make sure it uses paths that only consist of 8.3 file name components. This is because the long file names won't be visible at boot up time when startup files, such as `Config.sys` and `Autoexec.bat`, are processed.
- Try to use the long file name inside your application, because using the short file name can get you into trouble. Sometimes, certain operations change the short file name to something else. If your application is not prepared for this, it can blow up. If for some reason you need to determine whether a file name entered by the user is the same as a file that you currently have opened, you can check the `nFileIndexHigh` and `nFileIndexLow` members of the `BY_HANDLE_FILE_INFORMATION` structure, which is returned by the `GetFileInformationByHandle()` Win32 function.
- Beware when you temporarily rename a file, copy it and then rename it back to the original file name. The short file name might change to something different. Essentially, you shouldn't use the

short file name unless you're forced to by circumstances beyond your control. Windows treats the short file name as secondary to the long file name, and so should you.

Never assume that the extension contains only three characters. (This one has caught me a few times.) Also, keep in mind that the user might save the file with a different extension from the default one that your application provides.

Never allocate memory for a file name until you have gathered the maximum length of the file name from the **GetFileInformation()** function. This one will save you plenty of years in the loony bin. (I know that I get a little crazy whenever I can't find where I am overwriting memory.)

Keep in mind that users can type as many periods and spaces as their little hearts desire into a long file name.

Designing the UI to Support Long File Names

An application should support long file names and display them correctly. You can use the **SHGetFileInfo()** function in your application to retrieve the long file name for a file, as well as the file's icon, type name, attributes, and so on. If you include the File Open and Save As common dialog boxes in your application, you can use the **OFN_LONGNAMES** value to direct the dialog boxes to display and return long file names.

If you have used Explorer, you have already noticed that it doesn't display the extension of a file if it knows the type of that file. For example, instead of displaying My Letter To The IRS.doc, it would instead display My Letter to the IRS. Explorer will also display information about the type of document, for example Microsoft Word Document. If you want to eliminate the extension in your own applications and instead simply display type information for the file names, you can do this easily with the **SHGetFileInfo()** function.

Applications that must display a file name in the caption should display the long file name first, and if they display the name of the application, that should follow the file name separated by a dash character (-).

You should also support Universal Naming Convention (**UNC**) path names for files in your application. Using UNC names enables users to browse documents on the network directly and to open an application's files on remote machines without having to make an explicit network connection. Also, keep in mind that if your application usually parses through a file name to detect errors in the file name, as soon as you see **** in the beginning of the name, you should treat this name as a UNC name and parse it appropriately. An example of a UNC name is something like: **\\Myserver\Mypath\Myfile.ext**.

Finally, unless there is some special reason why you need to query the user with your own dialog boxes for file names, I suggest you use the common dialogs, because they already support all of the rules for file names (Unicode, UNC, etc.). When you use VC++ and MFC's classes for dialog boxes the work becomes even easier, since you don't have to set up those long structures (such as **OPENFILENAME**) for passing to the common dialog functions.

If you do have to write your own dialog boxes for querying the user for file names, make certain that the edit controls and list boxes can handle long file names. For list boxes, you must add the ability to scroll horizontally yourself.

Also, keep in mind that some of the things the user might want to traverse through are not necessarily file items from the system. They might be items from the shell, such as printers or other objects. Make sure you support these issues as well. For example, take a look at the common dialog box. Notice that it supports the concept of a My Computer object which leads to other objects (including files). The My

Computer object is a shell namespace item. We'll discuss the shell namespace next.

The Shell Namespace

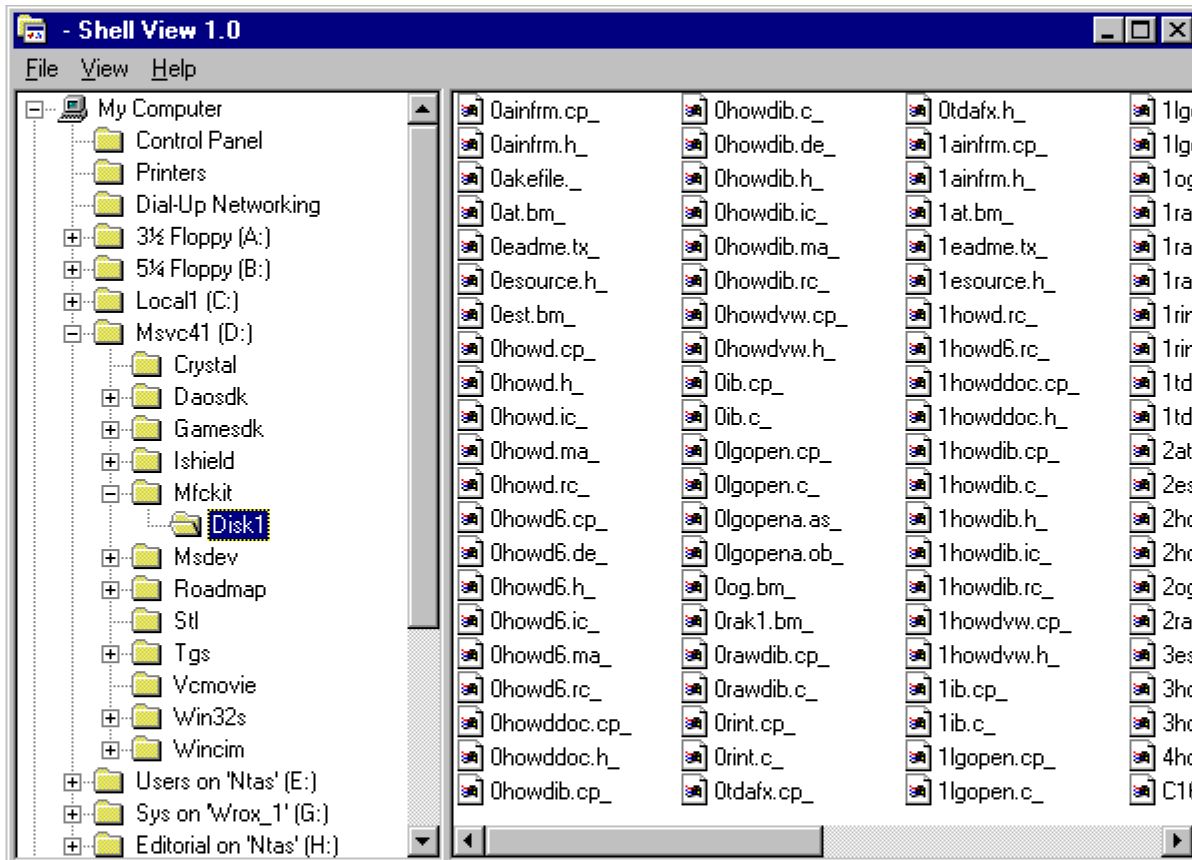
Windows 95 organizes information concerning the entire system in a logical hierarchy. At the top of this hierarchy is the **desktop**, which acts as the root (or parent) to all of the other items. Below the desktop are items such as drives, directories, files, and so on. There are also non-file system objects such as printers, control panel applications and program groups. All of this information is collectively known as the **shell's namespace**.

> **Remember, the shell's namespace is more than just the file system. It's a hierarchical collection of logical entities.**

The objects that make up the namespace can be divided into two groups: **file objects** and **folder objects**. File objects are at the end of the namespace hierarchy, since they don't contain other objects in the namespace, for example, files and printers. Folder objects, on the other hand, can be anywhere within the namespace hierarchy, since they can contain file or folder objects that are also part of the namespace.

In this section, we'll examine what it takes to build an application that utilizes the shell's namespace. I'll provide a pure MFC application that looks and acts very much like a junior version of Explorer. You can use the code as the foundation for your next Explorer-like killer application. My code takes total advantage of the MFC-provided classes for the framework, tree view, list view, and so on. I always try to use as few direct API calls as possible; instead I enjoy letting the MFC classes do much of the work for me so that I can spend more time playing video games and the like.

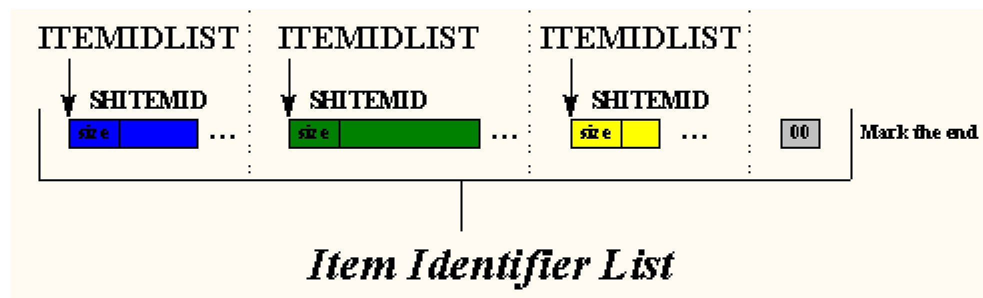
The application, which you can find on the CD, is called `ShellView`, and the main window is shown below. It acts very much like Explorer. If you click on the plus sign (+) on one of the folders in the tree list control, the child folders are displayed, and when one of the folders in the tree list control is selected, you immediately see the file objects that belong to the item in the list view control.



Keep in mind that when we say *file objects*, we don't necessarily mean an object in the file system. We're talking about the namespace. It can be a printer, a network connection, or any one of several other objects.

Every object in the shell namespace is assigned an item identifier that uniquely identifies the object within its parent's folder. This means that the parent of the object also has an item identifier that identifies the folder within its parent's folder. This continues until you reach the top object (which is the desktop object). To uniquely identify an object anywhere in the namespace, you would need to keep a list of the item identifiers for the object and all its ancestors. This list is known as an **item identifier list** or a **PIDL** (after **P**ointer to an item **ID**entifier **L**ist, which is how most shell functions accept or return an item identifier list; pronounced 'piddle').

There's a structure to help manipulate item identifier lists, called **`ITEMIDLIST`**. Its one and only member, **`mkid`**, is itself a structure, called **`SHITEMID`**. The first two bytes of the **`SHITEMID`** structure define the size of the rest of the structure which is undefined and contains a variable length array of bytes. An item identifier list (which consist of multiple item identifiers) consists of one or more **`ITEMIDLIST`** objects packed together followed by a 16-bit zero value to determine the end of the item identifier list. Here you can see how this looks in memory:



Users never see a PIDL. They usually see descriptive text that represents the PIDL (we'll see how we retrieve this text in a bit).

The most important thing to remember is that each folder object is, in fact, an OLE object which implements an interface, called `IShellFolder` (as well as `IUnknown` of course). When an application gets a pointer to an `IShellFolder` interface, it's called **binding to the object**. As a matter of fact, once you have a pointer to a folder's `IShellFolder`, you can retrieve a pointer to a child's `IShellFolder` by calling a member function, named `BindToObject()`, on the parent's `IShellFolder` interface, passing it the child folder's PIDL and you get back the child's `IShellFolder` interface.

In order to establish and display a hierarchy of objects (such as the one that Explorer and ShellView display) you'll need to start by getting a pointer to the topmost object's `IShellFolder`, which is the desktop. You do this by calling a function named `SHGetDesktopFolder()`. Every folder object must allow the caller to enumerate through all of its child objects. This is done with a call to `IShellFolder::EnumObjects()`. This function returns a pointer to an interface ,called `IEnumIDList`, which you can then call `Next()` on the interface (which returns a PIDL) to traverse through all of the folder's objects.

When the shell returns a PIDL to the calling application, it allocates them using OLE's task allocator (which your application can access by calling `SHGetMalloc()`). Also, if you need to create a PIDL and send it to the shell, make sure you get access to the task allocator and allocate the memory with the allocator. The task allocator implements an interface, called `IMalloc`. OLE applications that need to share data with each other must do so using the task memory allocator. Just remember to free any data that the shell sends you with the `IMalloc` interface, which you should also use to allocate and send to the shell any memory you need to.

I mentioned that there are some folders which keep objects that are not necessarily file system objects, such as the desktop folder or the network neighborhood folder. Program groups and the objects displayed on the desktop are kept in special directories on the file system and the location of these directories are kept in a registry key under:

```
HKEY_CURRENT_USER/Software/Microsoft/Windows/CurrentVersion/Explorer/Shell Folders
```

Other special folders which have no association to the file system at all, such as the control panel applications or the printers folder, are called **virtual folders**. The location of special folders can be retrieved with the `SHGetSpecialFolderLocation()` API function, which would retrieve a PIDL to the calling application. The function takes three parameters. The first is the handle of the owner window that should be used if the function needs to display a dialog box or an error message. The next is a value that indicates the folder who's location you want to retrieve. This parameter can be one of several flags that begin with the `CSIDL_` prefix, such as `CSIDL_BITBUCKET` or `CSIDL_CONTROLS` (for a description of all the

flags, look up **SHGetSpecialFolderLocation()** in your Win32 Programmer's Reference manual). The last parameter to the function is a **LPITEMIDLIST**, which is a PIDL specifying the folder's location relative to the root of the name space (the desktop).

Always free any PIDLs that the shell returns to you by calling **IMalloc::Free()** and any pointer to **IShellFolder**, **IEnumIDList**, or **IMalloc** by calling **Release()** on that interface.

The Functions and Interfaces

As we mentioned before, users don't see PIDLs and other types of identifier. Instead, they should see a friendly name that is more meaningful. The **IShellFolder** interface has several functions that you can use to query information about the folder or object.

For example, to ask for the friendly name, you would call **IShellFolder::GetDisplayNameOf()**. There are also a number of functions that take a PIDL as a parameter and return information. For example, **SHGetFileInfo()** returns information on the display icon of the object. Objects also have attributes associated with them. Things such as whether the object can be copied, the object has folder objects within itself, or the object is a drop target are just a few of the attributes that an object can support. These attributes can be queried by calling the **IShellFolder::GetAttributesOf()** member function of any **IShellFolder** object.

Using the Shell Namespace

Now let's start to examine some source code to see how we can build an application that lets us browse through the shell's namespace.

The code that I provided for this section (which you'll find on the CD in the **ShellView** directory) is a complete MFC application. If you look over the code, I hope you'll get a feel for what it takes to build an application like this with simplicity and efficiency, and most of all, using the MFC classes. You can build on this sample and extend it to do other things, such as provide drag-and-drop and activation of objects, just as Explorer does.

First of all, I generated the application using AppWizard as an MFC SDI application with no toolbar or status bar support. I also chose to have a splitter window (which is now a choice under the Advanced... section in AppWizard). My first view (which AppWizard allows you to create) was derived from **CTreeView** (a new class in MFC; a view window with a tree view control inside) and the other view is derived from **CListView** (a view window with a list view control inside). The splitter window is set up to create two static panes (1 row by 2 columns). I did all the work in the **CMainFrame::OnCreateClient()** member function:

```
BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT /*lpcs*/,
   CCreateContext* pContext)
{
   BOOL bRet = m_wndSplitter.CreateStatic(this, 1, 2);
   if (bRet == FALSE)
      return FALSE;

   CRect rc;
   GetClientRect(rc);

   bRet = m_wndSplitter.CreateView(0, 0,
      RUNTIME_CLASS(CShellViewTreeList), CSize(rc.right / 4, 0),
      pContext);
```

```
        if (bRet == FALSE)
            return FALSE;

        return m_wndSplitter.CreateView(0, 1,
            RUNTIME_CLASS(CShellViewListView), CSize(0, 0), pContext);
    }
```

To make things more logical, I created a class derived from **CObject** to handle the PIDLs. I called this class **CPidl** and it contains member functions to create PIDLs, copy them, concatenate them, and so on. If you walk through the code, you'll notice that I bombed it with comments all over the place to make it easy to understand. The most important thing to remember about this class is that most members use the task allocator to manage memory for the PIDLs. It stores two PIDLs within the class: one is the identifier that uniquely identifies the object within its parent folder and the other is the identifier list which leads you from the root object to the object a few levels down (in the hierarchy).

I should mention that a C++ object instantiated from this class can be sent to a shell function that requires a PIDL. This is done with a conversion operator which I created inside the **CPidl** class:

```
        operator LPITEMIDLIST() const
        {
            return m_pidl;
        }
```

When the tree view is first initialized (inside of **OnInitialUpdate()**), I call a function, named **PopulateTreeNode()**, which accepts a pointer to an **IShellFolder** interface, a pointer to an ID list (PIDL) and the node of the tree control to add the object. The **IShellFolder** pointer that is passed to the **PopulateTreeNode()** function is retrieved with a call to **SHGetDesktopFolder()**, which returns the desktop's folder.

After I create a few local variables in the **PopulateTreeNode()** function, I proceed to **EnumObjects()** on the passed in folder. This returns a pointer to the **IEnumIDList** associated with the folder. Once I've got the enumerator, I then start to traverse through each object within the folder and I create **CPidl** objects for each one.

The first folder is the desktop folder and as the user expands the plus (+) sign on the left of the folders in the tree control, I start to receive the different folders and fill their information. In other words, I call **PopulateTreeNode()** from **OnItemexpanding()**. I never fill anything until the user wants to see the underlying information. This means that the user must explicitly choose to expand a tree before I gather the items that belong to the particular folder. I use the **lParam** of the tree item to maintain the **CPidl** object for the item. The following is a code listing of **PopulateTreeNode()**:

```
    void CShellViewTreeList::PopulateTreeNode(LPSHELLFOLDER pFolder,
        LPITEMIDLIST pidlQ, HTREEITEM hParent)
    {
        HTREEITEM    hPrev = NULL;
        HRESULT      hr;
        LPENUMIDLIST pEnumIDList;
        LPITEMIDLIST pidlNew;
        ULONG        ulFetched;
        LPMALLOC     pMalloc;

        // Parameters needed for InsertItem function.
        UINT         nMask;
        CString      strItem;
        CPidl*       pidlObj;

        // Enumerate through the shell objects for the given folder.
        hr = pFolder->EnumObjects(GetSafeHwnd(), SHCONTF_FOLDERS |
                     SHCONTF_NONFOLDERS, &pEnumIDList);
        if (FAILED(hr))
```

```
            return;   // Might have clicked on network stuff
                      // where there is none.

         // Get the OLE task memory allocator object.
         hr = SHGetMalloc(&pMalloc);
         ASSERT(SUCCEEDED(hr));

         while (pEnumIDList->Next(1, &pidlNew, &ulFetched) == S_OK)
         {
            // What type of object do we have?
            ULONG ulAttrs = SFGAO_HASSUBFOLDER | SFGAO_FOLDER;
            pFolder->GetAttributesOf(1,(const ITEMIDLIST**)&pidlNew, &ulAttrs);

            // Is it a folder or does it have folders?
            if (ulAttrs & (SFGAO_HASSUBFOLDER | SFGAO_FOLDER))
            {
               // We don't want to add any non-folder objects
               // to the tree.
               if (ulAttrs & SFGAO_FOLDER)
               {
                  // Create a C++ CPidl object.
                  pidlObj = new CPidl(pidlNew, pFolder);
                  if (pidlQ != NULL)
                     pidlObj->SetQualifiedPidl(pidlQ);

                  // Setup the style of the node.
                  nMask = TVIF_TEXT | TVIF_IMAGE | TVIF_SELECTEDIMAGE |
                          TVIF_PARAM;
                  if (ulAttrs & SFGAO_HASSUBFOLDER)
                     nMask |= TVIF_CHILDREN;

                  // Get display name of object for showing
                  // in the tree ctrl.
                  pidlObj->GetDisplayName(strItem, SHGDN_NORMAL);

                  // Add the item to the tree ctrl.
                  hPrev = GetTreeCtrl().InsertItem(nMask, strItem,
                      pidlObj->GetNormalIcon(), pidlObj->GetSelectedIcon(),
                      0, 0, (LPARAM)pidlObj, hParent, hPrev);
               }
            }

            pMalloc->Free(pidlNew);
         }
         pEnumIDList->Release();
         pMalloc->Release();
      }
```

When the user selects to expand a folder, the `OnItemexpanding()` function is called, which then recalls `PopulateTreeNode()` on with the folder object:

```
         LPSHELLFOLDER pFolder;
         HRESULT hr = pidlObj->GetFolder()->BindToObject(*pidlObj, 0,
            IID_IShellFolder, (LPVOID *)&pFolder);

         if (SUCCEEDED(hr))
         {
            PopulateTreeNode(pFolder, pidlObj->GetQualifiedPidl(),
                             pNMTreeView->itemNew.hItem);
            pFolder->Release();
         }
```

As I mentioned before, it's very important to call `BindToObject()` in order to receive a PIDL for a selected folder object. This PIDL is then concatenated with its parent's PIDL to form its item identifier list (done inside `PopulateTreeNode()` with a call to `CPidl::SetQualifiedPidl()`).

The last question on your mind should be, "How does the list view get its information for the icons it

needs to display?" When a folder is selected in the tree view, the framework calls **OnSelchanged()**, which then sends a notification to the document class via **UpdateAllViews()** and passes it the folder that should be interrogated for its child objects:

```
LPSHELLFOLDER pFolder;
HRESULT hr = pidlObj->GetFolder()->BindToObject(*pidlObj, 0,
                                 IID_IShellFolder, (LPVOID *)&pFolder);

if (SUCCEEDED(hr))
{
   GetDocument()->UpdateAllViews(this, (LPARAM)pFolder, NULL);
   pFolder->Release();
}
```

When the list view's **OnUpdate()** member function is called, it begins by traversing through the objects and adding an item to the list view for each object in the passed in folder:

```
void CShellViewListView::OnUpdate(CView* pSender, LPARAM lHint,
   CObject* pHint)
{
   int           nItem = 0;
   HRESULT       hr;
   LPENUMIDLIST  pEnumIDList;
   LPITEMIDLIST  pidlNew;
   ULONG         ulFetched;
   LPSHELLFOLDER pFolder = (LPSHELLFOLDER)lHint;
   LPMALLOC      pMalloc;

   if (pFolder == NULL)   // Which it is, the first time.
      return;

   GetListCtrl().DeleteAllItems();

   // Get the OLE task memory allocator object.
   hr = SHGetMalloc(&pMalloc);
   ASSERT(SUCCEEDED(hr));

   // Parameters needed for InsertItem() function.
   CString   strItem;

   // Enumerate through the shell objects for the given folder.
   hr = pFolder->EnumObjects(GetSafeHwnd(), SHCONTF_FOLDERS |
        SHCONTF_NONFOLDERS, &pEnumIDList);
   if (FAILED(hr))
      return;    // Might have clicked on network stuff
                 // where there is none.

   while(pEnumIDList->Next(1, &pidlNew, &ulFetched) == S_OK)
   {
      CPidl pidlObj(pidlNew, pFolder);

      // Get display name of object for showing in the
      // tree ctrl.
      pidlObj.GetDisplayName(strItem, SHGDN_NORMAL);

      // Add the item to the tree ctrl.
      GetListCtrl().InsertItem(nItem++, strItem, pidlObj.GetNormalIcon());
      pMalloc->Free(pidlNew);
   }
   pEnumIDList->Release();
   pMalloc->Release();
}
```

Shortcuts and the Shell

I remember when I had to create a separate item for Program Manager in Windows 3.x in order to have an easy way of accessing my documents for further modifications. The path I entered was something like this:

```
C:\WINWORD\WINWORD.EXE   C:\DOCS\BOOK1.DOC
```

This would load MS-Word and my document into it. I'd have to create one for each document. This wasn't too bad, but the worst had to be when I created a separate program group, called My Desktop, and placed my most commonly used applications into it. Then it was necessary to bring up Program Manager, look for the My Desktop program group, and find the application before I could even get to the point of execution (by that point I wanted to execute myself).

Windows 95 has made this horror story completely go away with something they call **shortcuts** (a.k.a. **shell links**). The user has the ability to create links to documents, folders, applications, printers and other jewels stored in the shell's namespace. These links can be placed in any folder and appear in the form of an icon with an arrow by its side. (This arrow is actually known as the **system-defined link overlay icon**. That's quite a name! No one said that the Microsoft guys couldn't be verbose.) Here you can see the shortcut I have pointing to my Developer Studio.

The best part about shortcuts is that users don't have to remember where the path to the actual file or folder (or whatever it points to) is. And if the location of the object moves, the system tries to automatically update the shortcut next time the shortcut is activated.

Creating the shortcut is as simple as choosing Create Shortcut from an object's context menu or from Explorer's file menu. If you create a shortcut to another shortcut, the system copies the original shortcut without creating another shortcut. When you delete a shortcut item, the associated object is not affected. Activating the shortcut is just as easy; you simply double-click on it. Activating the shortcut doesn't necessarily mean opening up a document in an application. If the shortcut happens to point to a `.wav` file, double-clicking the shortcut might just play the sound file. You're executing the default behavior (or **verb**) of the object pointed to by the shortcut.

By now, you might be asking yourself, "That's great from a user's standpoint, but what does that have to do with me programming my applications?". Well, in the next section, we're going to look at creating shortcuts programmatically (this might be handy for placing certain files right on the desktop for easy access to the user) and resolving shortcuts in our own applications.

An application creates a shell link by using the `IShellLink` interface to create a shell link object and uses the `IPersistFile` or `IPersistStream` interface to store the object in a file or stream. The next section will use a sample application to demonstrate and explain how to call the different interfaces.

Link File Structure

The information for the link file is stored in a binary file with a `.lnk` extension. The following is a list of the items stored in that file:

- The location (path) of the object referenced by the shortcut.
- The working directory of the object.
- The list of arguments that the system passes to the object when the **`IContextMenu::InvokeCommand()`** member function is activated for the shortcut (see the context menu handlers section for more information).
- The show (**`SW_`**) command used to set the initial show state of the corresponding object.
- The location (path and index) of the shortcut's icon.
- The shortcut's description string which is displayed for it. For example, Shortcut to my document.
- The hot key for the shortcut.

The working directory is the directory that the object (application or whatever it is) will use to load working files from. For example, if a shortcut points to Developer Studio, you can tell it where the project file and source code files to use are located. Then, when you save new files, they will be saved to this same directory.

The command line arguments are the arguments that you want passed to the object when it's activated. Some applications take special symbols to achieve a specific goal.

Once the user has activated the object, it will be launched using the specified show-flag. The valid flags are the same as those used for the **`ShowWindow()`** API function. For a complete list, see the description of this function in the Win32 reference.

By default, shortcuts use the icon of the application they point to if the object is an application. If the object is a document, the shortcut uses the icon of the application registered in the registry for the document. It finds the application by looking at the extension of the document. When no registry entry has been made or the document has no extension, a default icon is assigned to the shortcut. The user is then asked to resolve the application that should be run, when the user attempts to activate the shortcut. You can change the icon of a shortcut programmatically if you wish.

Resolution

When a shortcut is loaded and the object it points to is loaded, this is called **resolution**. The system handles resolving the object in most cases (with the exception of shell links stored in a stream). The system proceeds with the following steps in order to resolve the object:

- The system searches the path associated with the link.
- If it doesn't find the object, it looks in the same directory for a file with the same creation date and attributes. This is done to resolve a link to a file whose name has been changed.
- Next it traverses recursively through any subdirectories of the current directory in the hopes of finding a file with the same name and/or time-stamp.
- If it doesn't find a match, it finally displays a dialog box with a message to the user, alerting them that the object was not found. You can have your application suppress this dialog box by specifying the **`SLR_NO_UI`** flag when you resolve the object programmatically.

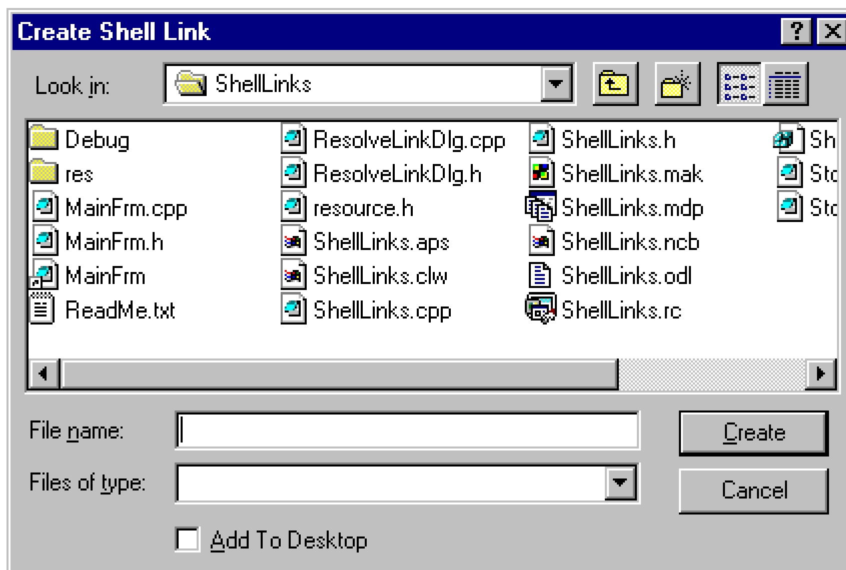
Creating Shell Links

I created a sample application to illustrate creating and using shortcut files, which you'll find on the CD as **ShellLinks**. I used MFC AppWizard (exe) to create the initial code for me with OLE Automation support. All other options were left with their default settings. Once the code was generated, I ripped out the document/view, since this application is one of those cases that doesn't need a document or a separate view. As a matter of fact, I didn't even need to write anything to the client area, since I handled everything via dialog boxes.

As ever, the initialization is handled in the **InitInstance()** function. This function calls **AfxOleInit()**, which must be called to initialize the OLE DLLs for further use. Later, we'll call other OLE functions, so we must make sure that the initialization is completed.

The application has two major choices on the File menu. One allows you to select a file to link to, and the other allows you to select a link file for viewing its information. Let's begin by creating the shell-link file (or shortcut).

Whenever I'm faced with a situation where I have to choose between doing some work or using code that does the work for me, I always choose to let the code do the work for me, which is exactly what I did to allow the user to choose a file. I knew that the Open common dialog already has support for traversing directories and displaying file names, but I needed to modify it a bit. That's where the **OPENFILENAME** structure and dialog hooks come in. You can see the final dialog box here:



Notice that my title reads Create Shell Link. The Open button has been changed to say Create and the Read Only check box now reads Add To Desktop. I made the change to the check box to allow me to determine whether the user wants to add the shortcut to the desktop or if the user wants to place the file in the current directory. All this work is done when the dialog box returns (if the user chose the Create button):

```
// If the user chose the "Create" button, continue.
if (dlg.DoModal() == IDOK)
{
   // Get the complete path + filename.
   CString strShortcut = dlg.GetPathName();
   CString strLink;

   // Did they select "Add To Desktop".
```

```
   if (dlg.GetReadOnlyPref() == TRUE)
   {
      // Synthesize the desktop path.
      TCHAR szPath[MAX_PATH];
      GetWindowsDirectory(szPath, MAX_PATH);
      strLink = szPath + CString("\\DESKTOP\\");
      strLink += dlg.GetFileName();
   }
   else   // Else store it in the current directory
   {
      // look for the extension (if any).
      TCHAR* pch = strchr(strShortcut, '.');
      if (pch != NULL)
         strLink = strShortcut.Left((int)(pch - strShortcut));
      else
         strLink = strShortcut;
   }

   strLink += ".LNK";

   CString strDesc = "Shortcut to " + strShortcut;

   if (!CreateNewLink(strShortcut, strLink, strDesc))
      AfxMessageBox("A problem occurred while trying to create the"
        " shell link.");
}
```

Note that I make an assumption about the location of the desktop folder. In your own applications, you'll want to retrieve this information from the system. You could, for example, call **SHGetDesktopFolder()** *and* **SHGetPathFromPidl()** *to retrieve the path.*

The real meat of the work is done in a function called **CreateNewLink()**. We begin by telling OLE and the shell server to create and return a shell link object that we can communicate with:

```
hResult = CoCreateInstance(CLSID_ShellLink, NULL, CLSCTX_INPROC_SERVER,
   IID_IShellLink, (LPVOID*)&psl);
```

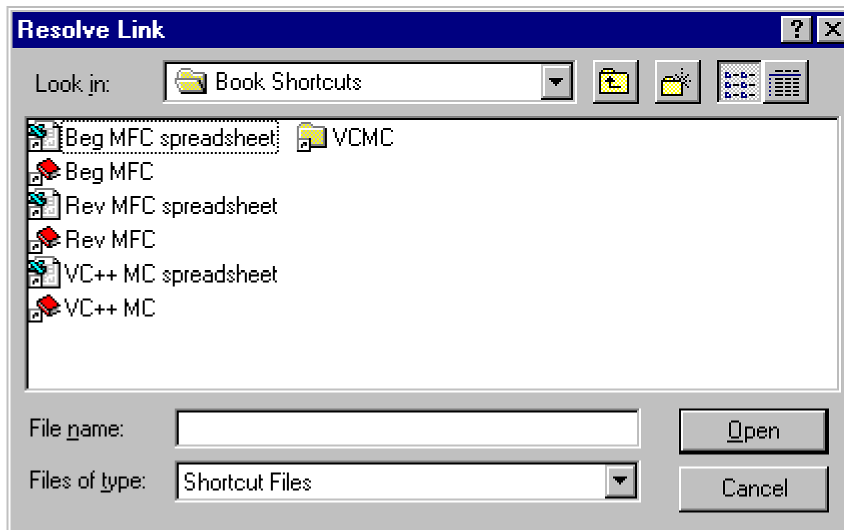
This object will also support an **IPersistFile** interface, which we'll use to tell the link file to save itself or load itself for us. The rest of the function retrieves the **IPersistFile** interface, sets the path and description of the object and then finally releases the object. Here's what the whole function looks like (I've removed some error checking here to save space and make it less confusing; the source code on the companion CD contains all of the error checking code):

```
BOOL CShellLinksApp::CreateNewLink(CString strShortcut, CString strLink,
   CString strDesc)
{
   IShellLink* psl;
   LPPERSISTFILE ppf;
   BOOL bRet = TRUE;
   OLECHAR wsz[MAX_PATH];   // buffer for Unicode string

   // Create the Shell-Link OLE object.
   CoCreateInstance(CLSID_ShellLink, NULL, CLSCTX_INPROC_SERVER,
      IID_IShellLink, (LPVOID*)&psl);

   // Get an interface pointer to the IPersistFile interface.
   psl->QueryInterface(IID_IPersistFile, (LPVOID*)&ppf);

   // Set the location for the object.
   psl->SetPath(strShortcut);

   // Set a description for the shortcut.
   psl->SetDescription(strDesc);

   // Create the link file.
```

```
    MultiByteToWideChar(CP_ACP, 0, strLink, -1, wsz, MAX_PATH);
    ppf->Save(wsz, TRUE);

    // Release both interfaces in order to free the object.
    ppf->Release();
    psl->Release();

    return bRet;
}
```

Displaying a Shortcut's Information

Selecting the Resolve Link menu item from the file menu brings up the dialog box shown:



Once the user has chosen a link file (and I do make sure that it's a *link file* within the code), I display the path information and description as shown:

Although I check the extension of the file involved to make sure that it's a link file, this isn't really necessary as you can assume that if the information is not appropriately returned when you try to resolve the link, something is wrong (the file is most likely not a shell link file).

Just like when we created a shell link file, we need to create an OLE shell link object, but this time we call the **Load()** member function of the **IPersistFile** interface to load the data instead of saving it. Before we call any other functions, we need to resolve the object, which means the object that the link file points to must be in the expected location or the path might need to be updated. The following source code illustrates displaying a link file's information. Again, I have removed the error handling to make everything clearer, but the full code is on the CD:

```
BOOL CShellLinksApp::ResolveLink(CString strLink)
{
    IShellLink* psl;
    LPPERSISTFILE ppf;
    BOOL bRet = TRUE;
    OLECHAR wsz[MAX_PATH];    // buffer for Unicode string

    // Create the OLE shell-link object.
    CoCreateInstance(CLSID_ShellLink, NULL, CLSCTX_INPROC_SERVER,
      IID_IShellLink, (LPVOID*)&psl);
```

```
    psl->QueryInterface(IID_IPersistFile, (LPVOID*)&ppf);

    // Load the link information.
    MultiByteToWideChar(CP_ACP, 0, strLink, -1, wsz, MAX_PATH);
    ppf->Load(wsz, STGM_READ);

    // Resolve the object.
    CResolveLinkDlg dlg;
    psl->Resolve(m_pMainWnd->m_hWnd, SLR_ANY_MATCH);\

    // Next, get the information we're looking for.
    TCHAR szPath[MAX_PATH];
    TCHAR szDesc[MAX_PATH];
    WIN32_FIND_DATA wfd;

    strcpy(szPath, strLink);
    psl->GetPath(szPath, MAX_PATH, (WIN32_FIND_DATA*)&wfd, SLGP_SHORTPATH);
    dlg.m_strPath = szPath;

    psl->GetDescription(szDesc, MAX_PATH);
    dlg.m_strDesc = szDesc;

    // Display the information.
    dlg.DoModal();

    // Release the OLE object.
    ppf->Release();
    psl->Release();

    return bRet;
}
```

When the **`Resolve()`** function is called, several parameters are passed. The first is the handle of the window to be used as the parent for display dialog boxes. The second is a combination of different values: **`SLR_ANY_MATCH`**, **`SLR_NO_UI`**, and **`SLR_UPDATE`**. We can combine these values to determine the action that should be taken by the system. **`SLR_ANY_MATCH`** resolves the link. If the user needs more information, they are prompted with a dialog box. However, if **`SLR_NO_UI`** is specified, which stops the system from displaying dialog boxes to the user, the system looks at the high-order word to determine the number of milliseconds to wait until it times out (the default is 3000). The last flag tells the system to update the path to the link and the list of identifiers if the link object has been changed.

Shortcuts to Non-file objects

We've seen how to create shortcuts to files programmatically, but we can also create links to other items within the shell name space. However, items such as printers don't rely on a file name to identify them within the namespace, so how can we create shortcuts to them? The answer, of course, is to use an **identification list**.

As we showed earlier, all items in the namespace (including files and directories) contain an identifier (ID) which uniquely identifies that item among all items contained within the same parent folder. Each parent folder also has its own ID. This being the case, each item can be identified by a list of identifiers, called an **ID list**. Remember that each item ID in an ID list is unique and meaningful only within the context of the parent folder that owns it. To create a shortcut to a non-file object, you would use the identifier list of the object and pass it to the **`IShellLink::SetIDList()`** member function.

IShellLink Functions

Once you get a pointer to an **IShellLink** interface, you can begin calling the functions of the interface (which include **AddRef()**, **Release()**, and **QueryInterface()**). The **IShellLink::GetArguments()** function is used to retrieve the command-line arguments associated with a shell link object. **IShellLink::GetDescription()** retrieves the description string for the associated shell link object. **IShellLink::GetHotKey()** returns the hot key associated with a shell link object.

The **IShellLink::GetIconLocation** function retrieves the location of the icon for a shell link object. **IShellLink::GetIDList** retrieves the list of item identifiers for the object. The **IShellLink::GetPath** retrieves the path and file name. The **IShellLink::GetShowCmd** function is called when you wish to display the shell link object and wish to know how to display it. The function returns the **SW_** flag associated with this entry. The **IShellLink::GetWorkingDirectory** returns the name of the working directory entry associated with the object.

Of course, you've already seen one of the most important functions, **IShellLink::Resolve**, which resolves the shell link and optionally searches for the object and updates the link's path and its list of identifiers. The other functions in the interface perform the opposite of the **Get*xxx*()** functions, allowing you to set the information for the link. These functions include **SetArguments()**, **SetDescription()**, **SetHotKey()**, and so on.

Shell Extensions

Have you ever wished that you could add menu options to a file's context menu, or assign a different icon to a file based on the content of the file instead of the same icon for all files of that type? If your answer to this question is yes, stand by because you're about to learn exactly how to do this, and more.

Windows 95 allows you to extend what they call the *shell* in many ways by providing **shell extensions**. Shell extensions allow you to provide a means for manipulating any type of shell object. Keep in mind that shell objects are not always files; they can be folders, drives, or even printer objects, as you saw when we considered the shell namespace. There are various actions for which you can supply **handlers** that will be called upon when that action takes place, such as when a file is being copied, when it's being dragged and dropped, or even when an object is dropped onto a particular file type. The following table lists all of the types of handlers that you can provide programmatically:

Handler Type	Description
Context menu handlers	Adds items to context menus for file objects.
Icon handlers	Provides an icon for instance-specific or class-specific file objects.
Data handlers	Provides additional formats for data being dragged or copied to the clipboard from the shell.
Drop handlers	Allows a file type to become a drop target for other objects.
Property sheet handlers	Adds pages to a property sheet for file objects.
Copy hook handlers	Determines whether a file object can be copied, moved, deleted, or renamed.
Drag-and-drop handlers	Provides a context menu when the user drags and drops an object to a new location.

In the next couple of paragraphs, we'll be discussing handlers, which brings in a few terms that you

should be aware of. First of all, a **shell object** is any object within the shell namespace. The term **file object** refers to any object within the shell namespace that doesn't contain any other namespace objects. **File class** (or **type**) refers to the set of files which are of the same type. Files are associated with a class or type, such as Word document types, or text file types. A handler is the extension code that you (or someone else) provides to work with a file class or object.

Just like file viewers and shell links, handlers also need to implement COM interfaces and depend heavily on the system registry to provide information to the shell about which handlers to load and when they need to be loaded.

I know you don't want me to simply go on writing without giving you a sample application to learn from, so I wrote one up to integrate most of the extensions into one in-process server. I created a server, called `ShellExts.dll`, that provides functionality for context menu, drag-and-drop, icon and property sheet handlers. The other types of handler are pretty easy and you'll be able to code them on your own once you've studied the code for the first couple of handlers. So let's get started with the nitty-gritty.

The Registry Entries

Before the shell can find any of your handlers, you need to provide some information in the system registry. I used a `.reg` file for my information, which you'll have to merge with the system registry before you try any of the handlers that I wrote. Also, keep in mind that if you place my handlers into different directories than the ones that I used, you'll have to modify the `.reg` file to provide the new location.

We'll take a look at the entries you'll need to provide for your handlers by examining the `.reg` file I provided with my example, `ShellExts.reg`. The first set of entries is the location of the handlers. These entries belong in the `HKEY_CLASSES_ROOT\CLSID` key. You'll need to provide the `InprocServer32` and the class ID entries. The `InprocServer32` should also contain the `ThreadingModel` entry. Since I used the same in-process server for all of my handlers, I got away with just one set of entries for the handlers. I chose to implement my handlers around the same Wrox (`.wrx`) file type as I used for my file viewer. Here you can see the CLSID entries:

```
[HKEY_CLASSES_ROOT\CLSID\{CC8DB1E8-4124-11CF-97E6-444553540000}]
@="Wrox File Extensions"
[HKEY_CLASSES_ROOT\CLSID\{CC8DB1E8-4124-11CF-97E6-444553540000}\InprocServer32]
@="c:\\msdev\\projects\\ShellExts\\debug\\ShellExts.DLL"
"ThreadingModel" = "Apartment"
```

In addition to the above, Windows NT requires that the handler's CLSID must be listed under a registry key that contains a list of approved handlers. By default, this key is protected from modification except by Administrators, so only they will be able to install shell extensions on NT. The key, shown below, has no effect on Windows 95:

```
[HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Shell Extensions\
Approved]
"{CC8DB1E8-4124-11CF-97E6-444553540000}" = "Wrox File Extensions"
```

The next set of entries are the file associate key and the application identifier key:

```
[HKEY_CLASSES_ROOT\.WRX]
@="wrxfile"
[HKEY_CLASSES_ROOT\wrxfile]
@="A Wrox File"
```

Once you've gone through the first steps, you're ready to start making entries for the different shell extensions. Keep in mind that just because I implemented my handlers all into the same server doesn't mean that I don't have to register each type of handler in the registry. To register the context menu handler for my server to be called each time a user right clicks on a Wrox file (**.wrx**), I used the following entries:

```
[HKEY_CLASSES_ROOT\wrxfile\shellex\ContextMenuHandlers]
@="WRXCM"
[HKEY_CLASSES_ROOT\wrxfile\shellex\ContextMenuHandlers\WRXCM]
@="{CC8DB1E8-4124-11CF-97E6-444553540000}"
```

I could provide more context menu servers to act on the same file class, but it just so happens that I only provided one in this case.

The next entry I provided in my **.reg** file is the icon handler. There can only be one icon handler, which is why I didn't provide an alias. I registered my icon handler with the following entries:

```
[HKEY_CLASSES_ROOT\wrxfile\shellex\IconHandler]
@="{CC8DB1E8-4124-11CF-97E6-444553540000}"
```

Drag-and-drop handlers are registered under the folder keys, such as the **Directory** key. This is a bit weird, but if you think about it for a second, you'll realize that it makes sense. What you're basically telling the system is that when you drag-and-drop any kind of object on this folder, it should use this handler. You can pretty much call your handler type anything you want, since the shell simply enumerates through all of the drag-and-drop handlers for that folder type and displays their menu items in the same context menu as all of the other handlers. In the example below, we're saying, when I drag-and-drop any object on any **Directory** folders, use the specified handler as well as the other handlers already registered for this folder:

```
[HKEY_CLASSES_ROOT\Directory\shellex\DragDropHandlers\wrxfile]
@="{CC8DB1E8-4124-11CF-97E6-444553540000}"
```

Remember that you could call **wrxfile** anything you want and it won't matter. The basic idea is to let anyone (including yourself) peeking around your system's registry know that this handler belongs to your set of applications. In my case, my **wrxfile** handler might have shipped with my Wrox application. If you only want to display the handler for a particular type of file, you'll have to perform this check from within your handler.

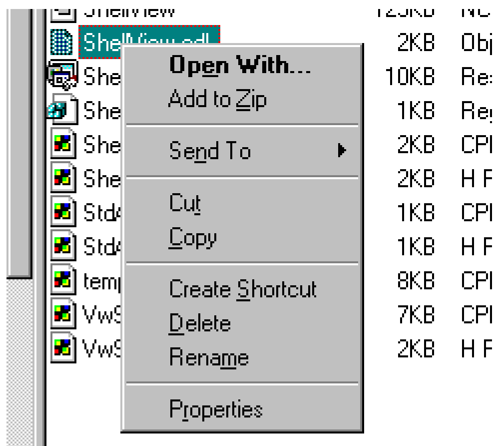
The following entries are for the property sheet handlers:

```
[HKEY_CLASSES_ROOT\wrxfile\shellex\PropertySheetHandlers]
@="WRXPP"
[HKEY_CLASSES_ROOT\wrxfile\shellex\PropertySheetHandlers\WRXPP]
@="{CC8DB1E8-4124-11CF-97E6-444553540000}"
```

You can also register a handler for all the file classes by using an asterisk (*) in the place of the application identifier, such as:

```
[HKEY_CLASSES_ROOT\*\shellex\ContextMenuHandlers\WinZip]
@="{e0d79300-84be-11ce-9641-444553540000}"
```

This entry says "Run the specified handler whenever a context menu is displayed regardless of the type of object selected."



Extensions without Code

On a different note, there are several registry keys that allow you to set a default icon for a file class or add commands to the File or New menus in Windows 95's Explorer without any programming at all. For example, you can set all files of the same type to have the same icon with an entry called **`DefaultIcon`** in the file association key for the file class. The entry should include the module and index of the icon. For example:

```
[HKEY_CLASSES_ROOT\.DOC\DefaultIcon]
@ = "C:\MSWORD\WINWORD.EXE, 0"
```

You can also add entries to the registry for context menus. The following is the type of entry in **`RegEdit 4`** format:

```
[HKEY_CLASSES_ROOT\<applicationID>\shell\<verb>]
@ = <menu text>
[HKEY_CLASSES_ROOT\<applicationID>\shell\<verb>\command]
@ = <command line>
```

An example would be something like this:

```
[HKEY_CLASSES_ROOT\wrxfile\shell\open]
@ = "Open an existing document"
[HKEY_CLASSES_ROOT\wrxfile\shell\open\command]
@ = "C:\MyProgs\MyEditor %1"
```

Although you or any user can do this, it's best to provide the information via shell extensions. The system then becomes cleaner and much more effective.

Creating Shell Extensions

When the shell needs to communicate with your shell extension handlers, it will first initialize the server via one of two interfaces. The first interface is **IShellExtInit**, which it will call when it needs to initialize an instance of a context menu, drag-and-drop, or property sheet handlers. You're already familiar with the other interface: **IPersistFile**. The shell will call this interface's **Load()** function for icon, data, copy hook, or drop handlers.

I used AppWizard (MFC AppWizard (dll)) to generate my sample application, ShellExts, with automation support (so that it would generate the appropriate OLE functionality for me). Later, I added a class called **CShellExt** derived from **CCmdTarget** which implements the interfaces for the shell extension handlers. I created the class with the Creatable by type ID option that appears in the New Class dialog box so that I wouldn't have to create a separate class factory for the objects. AppWizard adds the macros to do this automatically for me:

```
// Placed in the class declaration in the header file.
   DECLARE_OLECREATE(CShellExt)

// Placed in the implementation file.
IMPLEMENT_OLECREATE(CShellExt, "SHELLEXTS.SHELLEXT", 0xcc8db1e8, 0x4124, 0x11cf, 0x97,
0xe6, 0x44, 0x45, 0x53, 0x54, 0x0, 0x0)
```

Inside my **CWinApp::InitInstance()**, AppWizard adds a call to **COleObjectFactory::RegisterAll()** which is a **static** member function that registers all of the application's class factories with OLE.

Next, I added the **IShellExtInit** interface to my class in standard fashion (as described earlier in this book and in Technical Note 38 in the Visual C++ documentation). The interface declaration is placed inside my **CShellExt** class and it looks like this:

```
// Used in Menu, D&D, and Property sheet handlers.
BEGIN_INTERFACE_PART(ShellExtInit, IShellExtInit)
// IShellExtInit methods
STDMETHOD(Initialize) (LPCITEMIDLIST pidlFolder,
                       IDataObject* pdobj, HKEY hkeyProgID);
END_INTERFACE_PART(ShellExtInit)
```

There's only one member function (besides **IUnknown**'s members) for which I have to provide functionality: **Initialize()**.The shell will call this every time a handler needs to be initialized and it will receive all the information relevant to the handler.

The **LPCITEMIDLIST** parameter contains the folder that the user-selected object belongs to. The **IDataObject** pointer was the only member I needed to use. It contains the file names selected, and to retrieve them I must call the **GetData()** member function of the **IDataObject** interface to request data in the **CF_HDROP** format. This format is the same one used by the **DragQueryFile()** API function, so I can use that function to parse the file name(s). This is what my version of **Initialize()** looks like:

```
STDMETHODIMP CShellExt::XShellExtInit::Initialize(LPCITEMIDLIST
                        pidlFolder, IDataObject* pdobj, HKEY hkeyProgID)
{
   METHOD_PROLOGUE(CShellExt, ShellExtInit);

   HRESULT hr = E_FAIL;
   FORMATETC fe =
   {
      CF_HDROP,           // CF_HDROP format.
      NULL,               // No specific device to render for.
```

```
      DVASPECT_CONTENT,  // Complete content.
      -1,                // Must be -1 cause Microsoft says so.
      TYMED_HGLOBAL      // Expect it back as HGLOBAL memory.
   };

   STGMEDIUM med;

   // Check for object existance.
   if (pdobj == NULL)
   {
      TRACE("CShellExt::XShellExtInit::Initialize received invalid"
            " IDataObject.\n");
      return E_FAIL;
   }

   // Let's get the selected file name(s).
   // We provide the fe and it sends us back the HGLOBAL inside of med.
   hr = pdobj->GetData(&fe, &med);
   if (FAILED(hr))
   {
      TRACE("CShellExt::XShellExtInit::Initialize failed in call to"
            " GetData.\n");
      return E_FAIL;
   }

   LPTSTR lpStart = (LPTSTR)GlobalLock(med.hGlobal);
   pThis->m_strFileName = lpStart;
   GlobalUnlock(med.hGlobal);

   // We can use the Drag file functions with the HGLOBAL since the names
   // will be in the expected format.
   if (DragQueryFile((HDROP)med.hGlobal, (UINT)-1, NULL, 0)== 1)
   {
      LPTSTR lptstr = pThis->m_strFileName.GetBuffer(MAX_PATH);
      DragQueryFile((HDROP)med.hGlobal, 0, lptstr, MAX_PATH);
      pThis->m_strFileName.ReleaseBuffer();
      hr = S_OK;
   }
   else
      hr = E_FAIL;

   // Release the storage medium.
   ReleaseStgMedium(&med);

   return hr;
}
```

You'll remember that the shell extension also needs to implement **`IPersistFile::Load()`**. For this member function, you can look back at my example of the same function for the file viewer code. You don't have to provide the other members for **`IPersistFile`**. Simply return **`E_NOTIMPL`**. However, you have to provide the **`IUnknown`** members as I did (again, refer back to the previous sections for examples of implementing the **`IUnknown`** members).

Context Menu Handlers

To create a context menu handler you have to implement two interfaces in the shell extension object that will be exposed through the class factory. The two interfaces are **`IShellExtInit`** and **`IContextMenu`**. Besides the usual **`IUnknown`** members, **`IContextMenu`** contains three extra members: **`QueryContextMenu()`**, **`InvokeCommand()`**, and **`GetCommandString()`**.

`QueryContextMenu()` is called when an object's context menu is about to be displayed. It gives your handler a chance to include more menu items on the context menu. The function receives five parameters, **`hMenu`**, **`indexMenu`**, **`idCmdFirst`**, **`idCmdLast`**, and **`uFlags`**. You must call **`InsertMenu()`** to add your menu

choices using the provided menu handle. The **indexMenu** parameter specifies the position of your first menu option and you would need to increment it for any other options you add. The ID that you should use is specified by **idCmdFirst** and you would need to add one to all the other menu items you provide. When you add your menu choices, make sure that you specify the flags **MF_STRING** and **MF_BYPOSITION** to the **InsertMenu()** function. This is my implementation of the function

```
STDMETHODIMP CShellExt::XContextMenu::QueryContextMenu(HMENU hmenu,
   UINT indexMenu, UINT idCmdFirst, UINT idCmdLast,UINT uFlags)
{

    ::InsertMenu(hmenu, indexMenu++, MF_STRING | MF_BYPOSITION,
        idCmdFirst + ID_WHATSNEW, _T("&What's New?..."));

    return MAKE_SCODE(SEVERITY_SUCCESS, FACILITY_NULL, (USHORT)2);
}
```

There are several flags that the function can receive within the **uFlags** parameter (these flags might be passed in, combined). The first of the flags is the **CMF_DEFAULTONLY**, which tells any handler that responds to it that the user has double-clicked on the object and initiated the default action. **CMF_EXPLORER** is sent when the user right-clicks on an object in the left-most pane (the tree view) of Explorer. The next flag **CMF_NORMAL** is the one that you will normally respond to. The last one, **CMF_VERBSONLY** is sent to your handler when the object is a short cut item, but your handler will usually ignore it.

The **InvokeCommand()** function is called when the user selects a menu option from the context menu. You need to provide any code for the menu choices that you added. The one and only parameter to this function is a pointer to a **CMINVOKECOMMANDINFO** structure, which contains a member called **lpVerb**. This member provides the ID of the menu choice that the user selected in its low word. Keep in mind that the ID is the menu ID that you provided minus the **idCmdFirst** value:

```
STDMETHODIMP CShellExt::XContextMenu::InvokeCommand(LPCMINVOKECOMMANDINFO
   lpici)
{
   METHOD_PROLOGUE(CShellExt, ContextMenu);
   HRESULT hr = E_INVALIDARG;
   UINT idCmd;

   if (!HIWORD(lpici->lpVerb))
      idCmd = LOWORD(lpici->lpVerb);
   else
   {
      TRACE("CShellExt::XContextMenu::InvokeCommand contained invalid"
            " lpVerb.\n");
      return hr;
   }

   if (idCmd == ID_WHATSNEW)
   {
      CString str;

      wsprintf(str.GetBuffer(300), _T("There is nothing new today for"
               " file: %s"), pThis->m_strFileName);
      str.ReleaseBuffer();
      AfxMessageBox(str);
   }

    return NOERROR;
}
```

The last function is **GetCommandString()**, which is called to provide help text for the context menu choice that it provides you with. You simply check the ID and copy the help text into the **pszName** parameter that is passed to the function. The last parameter into the function, **cchMax**, determines the

maximum length of **lpszName**. Don't overwrite the string (or you know what happens after that). Once again, here's the code:

```
STDMETHODIMP CShellExt::XContextMenu::GetCommandString(UINT idCmd,
   UINT uType, UINT* pwReserved, LPSTR lpszName, UINT cchMax)
{
   if (idCmd > ID_WHATSNEW)
       return ResultFromScode(E_INVALIDARG);

   if (idCmd == ID_WHATSNEW)
       lstrcpy(lpszName, _T("Provide New Information"));

   return NOERROR;
}
```

That's all there is to context menu handlers. Don't forget to register the handler with the registry or you won't see anything happen when you right-click on the file objects for which you provided the menu handler.

Drag-and-drop Handlers

Once you understand how to implement code for a context menu handler, providing code for a drag-and-drop handler is the same thing. You use the same exact interfaces, except you use a different registry entry. That's it. Can't be much easier than that.

The drag-and-drop handler is activated when you drag and release a shell object from one place to another with the right mouse button. For information concerning the appropriate key to register in the system registry, see the relevant section in this chapter a few pages back.

Icon Handlers

Windows 95 allows you to control the icon displayed for a file or class of files by providing a shell extension called an icon handler. If the shell finds a registry entry for the class, it loads the server, creates an instance of the COM object and calls **IPersistFile::Load()** to pass the file name and initialize the COM object.

To retrieve the icon information from the handler, the shell communicates with an interface that must exist in the handler, called **IExtractIcon**. There are two member functions: **GetIconLocation()** and **Extract()**. **GetIconLocation()** is called first to determine the impact of the icon (whether it should span across the entire file class or just one file instance) and the location (filename plus index) of the icon.

In my example, I set the flag to **GIL_PERINSTANCE**, which acts on each file instance at a time, instead of the entire file class (it's better to use the procedure outlined in the registry section concerning the **DefaultIcon** entry if you want to set the icon for an entire file class) and I return the location and index of the icon I wish to use. You can retrieve this location from anywhere you'd like (the registry, **.ini** files, and so on.

A great use for icon handlers is when you want to display a different icon depending on the status of the file. In other words, if you read something from the file that tells you that the file is now in some different state, you might want to the display the icon for the file differently. The following is my version of the **GetIconLocation()** function:

```
STDMETHODIMP CShellExt::XExtractIcon::GetIconLocation(UINT uFlags,
```

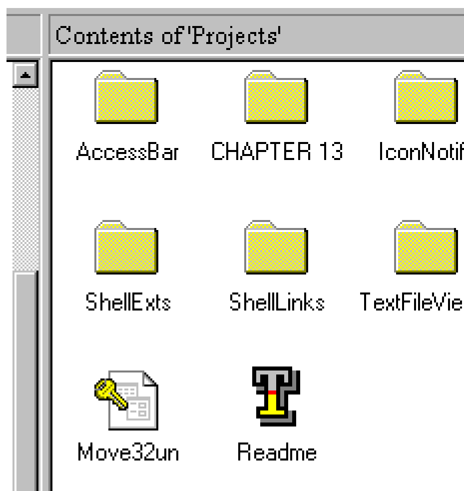
```
   LPSTR lpszIconFile, UINT cchMax, int* piIndex, UINT* pwFlags)
{
   METHOD_PROLOGUE(CShellExt, ContextMenu);

   // ICON_LOCATION defined at top of file
   lstrcpyn(lpszIconFile, ICON_LOCATION, cchMax);
   *piIndex = 0;
   *pwFlags |= GIL_PERINSTANCE;
   return S_OK;      // Return S_FALSE to use default icon.
}
```

There are two values that may appear in the **uFlags** parameter sent to the **GetIconLocation()** function. The first is **GIL_FORSHELL** and it's passed when a shell folder object is being worked on. The second, **GIL_OPENICON**, is used to determine whether the icon is being requested on behalf of an opened folder object.

The shell calls the **Extract()** function if the file returned by **GetIconLocation()** is not an **.exe** or **.dll** file, or if you specified **GIL_NOTFILENAME**. The handler must then return a handle to an icon. Note that you can't even return a **.ico** file. In my example, the file happens to be a dynamic-link library (**.dll**), so I just simply return **S_FALSE** from **Extract()** (the **Extract()** function won't be called anyway, since I return the file name from **GetIconLocation()**):

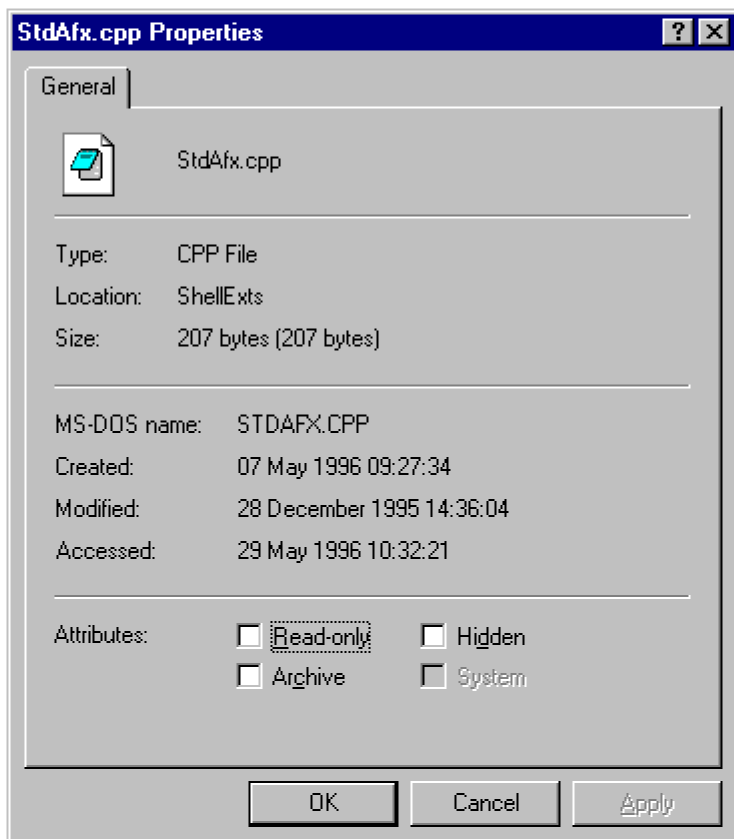
```
STDMETHODIMP CShellExt::XExtractIcon::Extract(LPCSTR lpszFile,
  UINT uIconIndex, HICON* phiconLarge, HICON* phiconSmall, UINT uIconSize)
{
   return S_FALSE;
}
```

Here you can see what the icon for a Wrox file (**.wrx**) looks like in Explorer, after my handler has been registered:



Property Sheet Handlers

When a user bring up the context menu (by right clicking) on a file object in Explorer and selects Properties or selects Properties from the File menu, the user is presented with a property sheet like that shown:



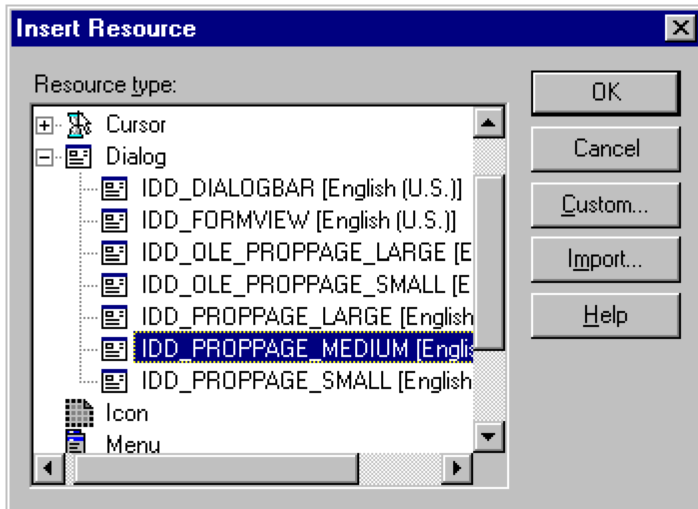
You can write a shell extension handler to add extra property pages for any particular type by registering your property sheet handler in the registry and providing the necessary COM interfaces.

The shell will initialize the handler by calling the **`IShellExtInit::Initialize()`** function first, followed by a call to a new interface, called **`IShellPropSheetExt`**. This interface contains two members: **`AddPages()`** and **`ReplacePages()`**.

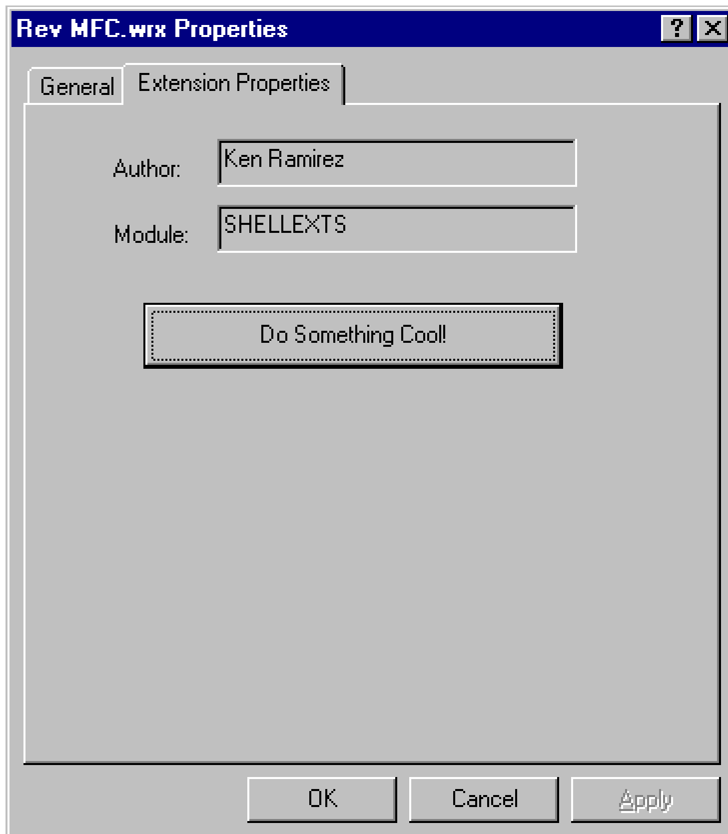
`AddPages()` is called so that a handler can add its pages to the property sheet dialog box. I used MFC to create the property page, since I always take the short route and never reinvent the wheel. This allowed me to use ClassWizard to add message handlers for any incoming messages to the property page. The **`AddPages()`** function is passed a pointer to a function that should be called to add the page to the dialog box once the page has been created. The function also receives an **`lParam`** to be passed to the function when the function pointer is executed.

I used the dialog editor to create my property page with the style set to Child, the border set to Thin and the Disabled flag turned on. The easiest way to create a property page with these styles is to use one of the predefined dialog resource templates provided with Visual C++. These are called **`IDD_PROPPAGE_LARGE`**,

IDD_PROPPAGE_MEDIUM and **IDD_PROPPAGE_SMALL**. They have the added benefit of being a standard size. I used **IDD_PROPPAGE_MEDIUM** as the basis for my property page.



I then created a class for the property page, called **CExtProps**, which is derived from the MFC **CPropertyPage** class. When it's run, the property page displays my name in the author box and the in-process server's name in the module box, as shown:



When you're using property pages, there are couple of very important things to consider. The most important of these is reference counting. To begin with I had the hardest time getting all this to work using MFC. The handler blew up on me several times, but it was consistent and always on the same line of code.

It turns out that the shell was creating a handler object, which would then create a property page as expected. The property page itself was handled by a **CPropertyPage** object that I created in my application. This, of course, contains an MFC dialog procedure which then calls any of the message handlers I added to my property page class. Unfortunately, once the handler had finished its work, the shell would release it, causing the server to be freed from memory, even though I still needed the server in memory for the **CPropertyPage** object.

I needed some way of letting the server know that it should stay in memory as long as the property page that the server created is being used. Luckily enough, the **PROPSHEETPAGE** structure that needs to be passed to the **CreatePropertySheetPage()** API function has a member for just such a situation, **pcRefParent**. (The **dwFlags** member of the same structure must contain **PSP_USEREFPARENT** for Windows to pay any attention to the value of **pcRefParent**.)

I set the **pcRefParent** member to the address of the **m_nObjectCount** member of the **AFX_MODULE_STATE** structure, of which one is created for every MFC application or DLL. This member is used by MFC to maintain its own reference count, so it should come as no surprise when I tell you that **m_nObjectCount** is the member that gets incremented when you call **AfxOleLockApp()** and decremented when you call **AfxOleUnlockApp()**. This member is also checked from MFC's default handling of **DllCanUnloadNow()**, which returns **TRUE** (which causes the DLL to be thrown out of memory) if the object count is less than one, or **FALSE** (which keeps the DLL in memory) if the object count is greater than zero. By sending the shell the address of **m_nObjectCount**, our handler will remain in memory until it's no longer needed, since the **m_nObjectCount** will be incremented and later released the correct number of times. I wish I could tell you more about these hidden gems inside MFC, but there's just not enough room in this book.

This is the code that I placed in the application's **InitInstance()** to handle this case:

```
AFX_MODULE_STATE* pState = AfxGetModuleState();

m_ppg = new CExtProps;

m_ppg->m_psp.dwFlags |= PSP_USEREFPARENT;
m_ppg->m_psp.pcRefParent = (UINT *)&pState->m_nObjectCount;
```

Notice that I made the property page a member of the application object. I did this for two reasons. The first is that the page object is created and initialized only once (as long as the DLL remains in memory). The second is that, if the extension object is released, the property page continues to live on (sort of like rock & roll). The code for my **AddPages()** function follows:

```
STDMETHODIMP CShellExt::XShellPropSheetExt::AddPages(LPFNADDPROPSHEETPAGE
   lpfnAddPage, LPARAM lParam)
{
   METHOD_PROLOGUE(CShellExt, ShellPropSheetExt);
   CShellExtsApp* app = (CShellExtsApp*)AfxGetApp();

   HPROPSHEETPAGE hPSP = CreatePropertySheetPage(&app->m_ppg->m_psp);

   if (hPSP == NULL)
      return E_OUTOFMEMORY;

   if (!lpfnAddPage(hPSP, lParam))
   {
```

```
            DestroyPropertySheetPage(hPSP);
        }

        return NOERROR;
    }
```

The other problem I ran into was that I needed to link my DLL with the static version of MFC, instead of the DLL version. The reason for this has to do with resource sharing. It couldn't find my resources because the code for the page's dialog procedure was in the MFC DLL, so it had no way of knowing my DLL's resources. To make a long story short, just link up to the static version of MFC and everything should work just fine.

The **ReplacePage()** function is called for Control Panel applications, when you provide a property sheet extension handler for any of those applications. To replace a page, a property sheet handler fills a **PROPSHEETPAGE** structure, calls **CreatePropertySheetPage()** and then calls the function pointed to by **lpfnReplacePage**. The function is allowed to replace any of the existing property pages with a new one. As for property sheets for non Control Panel applets, you can simply return **E_NOTIMPL** from this function.

Copy Hook Handlers

The system allows extensions called copy hook handlers which are called when a drive, folder or a printer object (which I will simply refer to as folder objects) is about to be copied, moved, deleted, or renamed. The purpose of the handler is to approve or disapprove of the operation by returning **IDYES** (to continue with the operation), **IDNO** (to prevent the operation on this folder, but allow others in a batch operation to continue) or **IDCANCEL** (to prevent all operations in the batch). The system calls all of the handlers registered for the appropriate folder type until all the handlers have been called or one of them returns **IDCANCEL**.

The copy hook handler is not initialized through **IShellExtInit** or **IPersistFile**. Instead, it's both initialized and activated with an interface called **ICopyHook**. The interface contains a single function, named **CopyCallback()**, which is called right before the operation is carried out. If the function returns **IDNO**, the operation is not carried out. If the function returns **IDYES**, the system proceeds with the operation (other handlers permitting).

The function receives a parameter, called **wFunc**, which contains a value indicating what is about to happen to the folder object. The values are: **FO_COPY**, **FO_MOVE**, **FO_RENAME** and **FO_DELETE**. It should be pretty obvious what these values mean. (There are also values relating to printers, but we won't go into those.)

Many times when we install our application, we assume that the user won't move the application's directory path to a new location (bad assumption). If the user *does* move the folder to a different place, wouldn't it be nice to know about it when it happens? If we register a copy hook handler for the directory which contains the application, we'll immediately know to update our information about the application's location. Of course, to make this happen, we need to work a little magic because copy hook handlers aren't alerted when the copy operation has completed. They are only used to allow or deny an operation from happening, so you would have to implement some other means of checking.

Drop Target Handlers

Drop target handlers exist so that files can become drop targets for other objects. For a handler to support this feature, it must implement two interfaces: the now famous **IPersistFile** and the **IDropTarget**

interface (see the OLE reference for a description of the `IDropTarget` interface and its members).

A drop target handler is activated when a user drags an object over one of the files that has a drop target handler registered for that file class. The `IDropTarget` interface is given a chance to respond when the object is dragged into the area of another object. Once the object is in the area of the drop target, the drop target can give visual feedback as the dragged object is moved over it, or when the object is moved away from the target. Finally, the handler can do its thing when the object is dropped on to the target object.

Keep in mind that this handler is only registered as *one per type*, meaning that each object type can only have one drop target handler registered for it.

The registry key used for registering a drop target handler is as follows:

```
[HKEY_CLASSES_ROOT\<file type>\shellex\DropHandler]
@ = <CLSID value>
```

Data Object Handlers

When the user drags a file from the shell to a new location or copies it to the clipboard, the system creates a COM object containing the `IDataObject` interface, which allows the file object to provide the data in several formats. You can register a data object handler for any file class, and it's called to provide additional data formats for the default `IDataObject` provided by the shell. In reality, what happens is that the default object delegates calls to the object provided by you. Unlike most handlers (but like the drop target handler), there can only be one data object handler per file type.

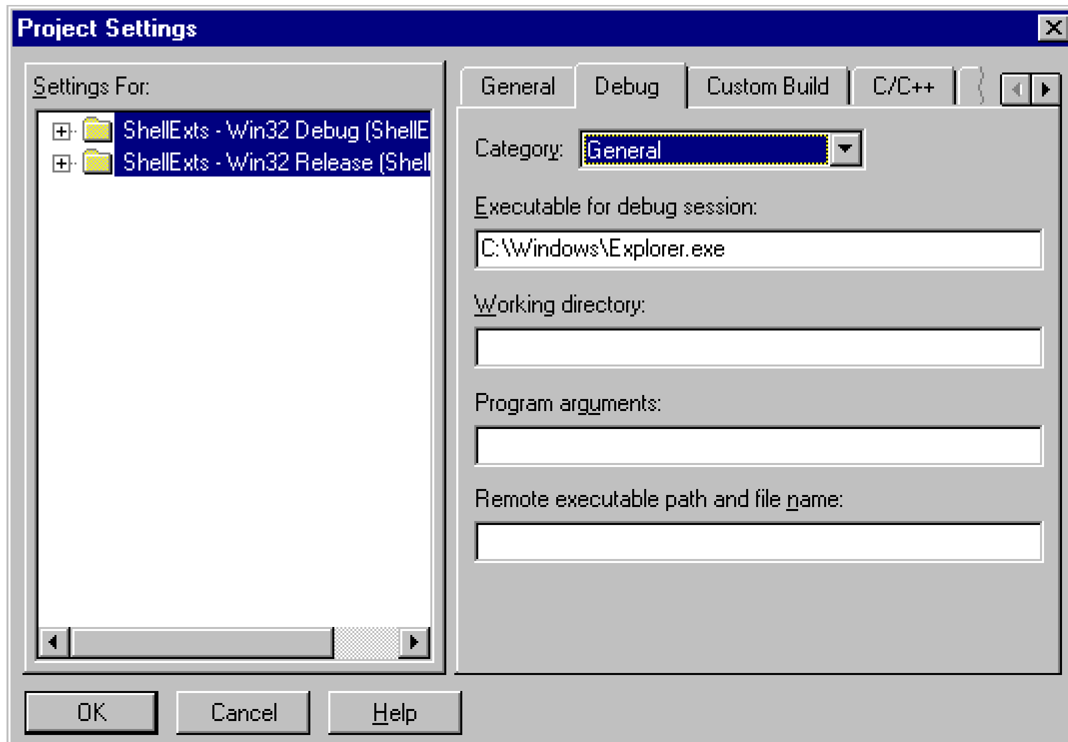
The shell calls the `IPersistFile` interface on your handler to initialize the extension handler and then calls the members of `IDataObject` for the rest of the work.

This is the registry key used for registering a data object handler:

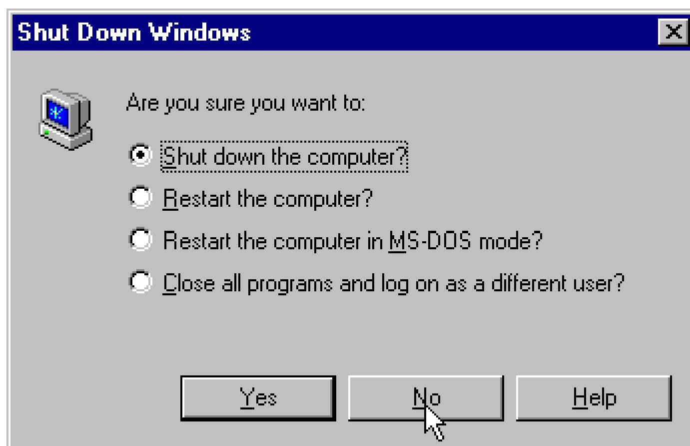
```
[HKEY_CLASSES_ROOT\<file type>\shellex\DataHandler]
@ = <CLSID value>
```

Debugging Shell Extensions

Since the shell extensions are in-process servers (or DLLs), they need to debugged within the context of an application. In the case of shell extensions, that application has to be the `Explorer.exe` application located in the Windows directory. To run this application for debugging the shell extensions, you need to tell Developer Studio that it should be run when ever you run the DLL. You do this by selecting the Settings... option from the Build menu. Once you see the Project Settings dialog, select the Debug tab and enter C:\Windows\Explorer.exe (or whatever the path is to your Windows directory) into the edit control for Executable for debug session.



However, you can't run the application as long as the shell is already up and running (which it is when you first load up Windows 95), so, before you run your application, you'll have to shut down the shell without shutting down Windows 95. You do this by selecting to shut down (which causes the Shut Down Windows dialog box to be displayed), then you hold the *Shift+Ctrl+Alt* buttons and click on the No button. This causes the shell (which is controlled by **`Explorer.exe`**) to be thrown out of memory.



You'll see a blank screen (such as the one you're used to seeing in Windows 3.x), but don't panic. You can still traverse through the applications, using the *Alt+Tab* combination, and you can double-click a blank area of your screen to bring up the task manager (which allows you to select currently running applications or fire off new applications).

Once you've gone through all of this, you can place break points on any line of your extension handlers and they'll be called when the shell calls down to your handler.

If you run into trouble, simply select Stop debugging from the Debug menu which will cause **`Explorer.exe`** to be thrown out of memory again and shut down your handler.

Summary

So now you've seen just some of the ways in which you can enhance and extend the Windows 95 shell. We've covered everything from access bars to shell extensions, so I hope there was something there that fired your imagination. Now it's up to you to get out there into the real world and start using this information to create something useful!

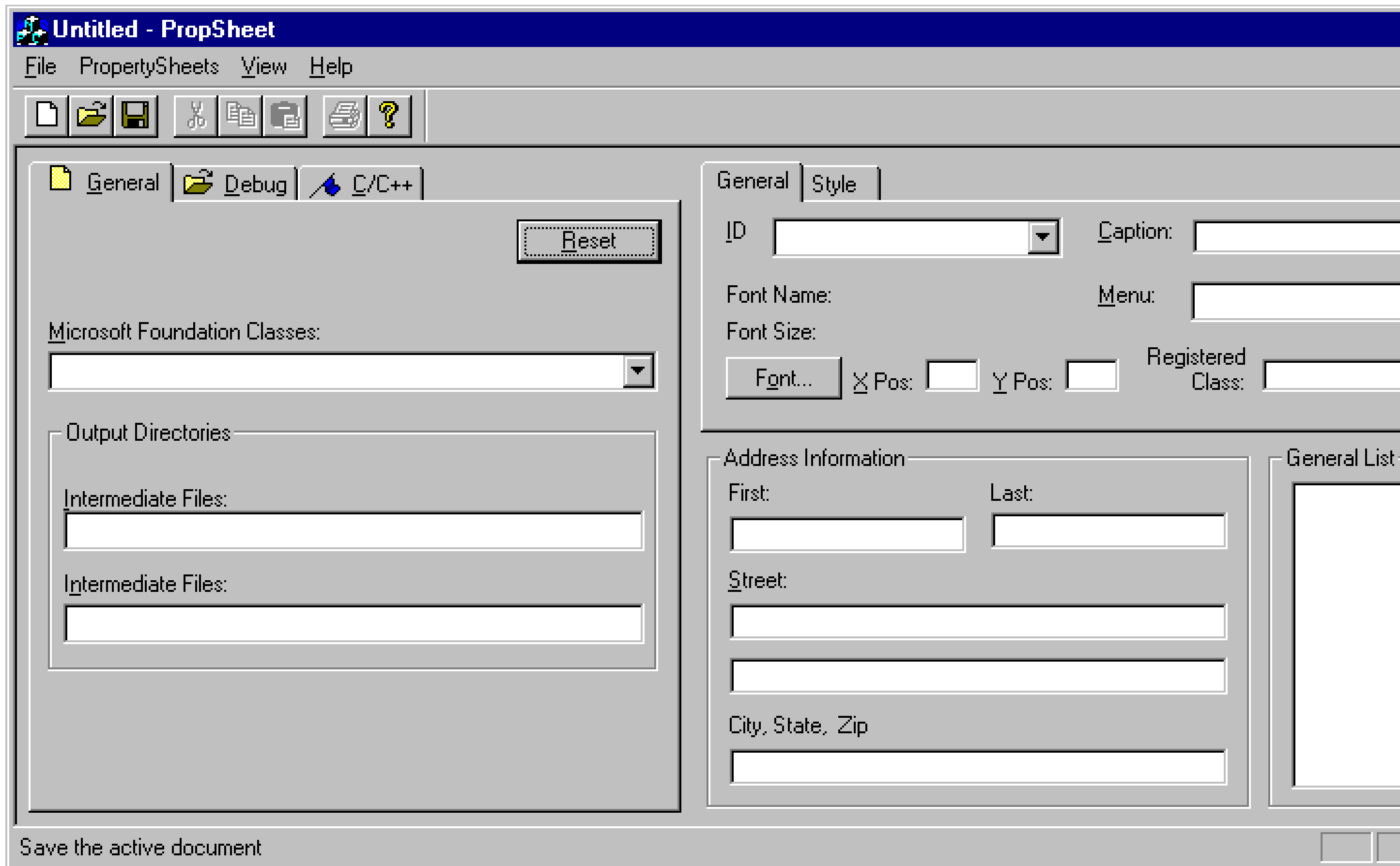
Advanced Dialogs and Property Sheets

In this chapter, we'll look in some depth at two related aspects of the Windows user interface—dialogs and property sheets—and at ways in which they can be customized and used together.

First, Dave Gillett will take a look at how dialogs and property sheets and pages work, and how they are used in Windows 95. This is followed by Saud Alshibani's practical tutorial on using, modifying and extending the MFC dialog and property sheet classes. As well as offering practical tips on using the stock controls, such as how to use bitmaps on the tabs of a tab control and how to embed property sheets in various types of window, we'll build a dynamic property sheet class which enables sheets to create the dialogs for their property pages from dialog templates at run time.

Here's just a sample of what the techniques and reusable classes presented in this chapter will allow you to do. The following figure shows a window which contains, among other things:

- A property sheet class with its associated property pages (which look, deliberately, like features in Developer Studio).
- A tab control (**`CTabCtrl`**) that encapsulates subdialogs that are created at run time.
- Additional subdialogs that are created at run time.
- A property sheet that has been resized at run time using the routines provided in the reusable base class **`CPropSheetExtended.`**
- Bitmap images on the tabs.
- A property sheet inside a **`CFormView`**.



Child Dialogs and Property Pages

Before we get into the nitty-gritty of coding, let's start the ball rolling with some thoughts on dialogs and property pages. Most Windows programmers are quite familiar the notion of dialogs, whether modal or modeless, as pop-up windows. A resource editor, such as the one included with Visual C++, is used to lay out a dialog resource, specifying various special kinds of windows as its children. At run time, the resource is loaded and the children created.

A dialog resource is a dual-purpose object. On the one hand, it defines a rectangular surface and the layout of various components upon it. On the other, it also specifies the style, caption, and so on, for a window to encapsulate that surface. The window normally has the **`WS_POPUP`** style and is usually owned by, but not clipped to, the application main window. It may be modal (the behavior of the application or even the entire system may be constrained while the window is visible), or it may be modeless, simply coexisting with the other windows on the screen.

Child Dialogs as 'Control Surfaces'

Pop-up windows are not the only possibility for dialogs, however. Application top-level windows are typically created with the **`WS_OVERLAPPED`** style, and one might create a dialog this way if it was to serve as an application main window. (This is not, however, how MFC's **`CFormView`** class works. It actually uses a borderless modeless dialog as the client of a frame window.)

The other alternative is the **WS_CHILD** style, which opens up some interesting possibilities. We know that a dialog contains child controls, but what if it also contained **WS_CHILD** dialogs? Nesting dialogs in the same way that you nest directories or folders, storages, or structures is a potent concept. There's a subtle but powerful distinction between the simple view of a dialog as a container for controls, and the notion of 'control surfaces' as a hierarchical way to compose user interface elements.

Of course, anyone who has tried to use a modeless dialog as a child of another dialog has discovered that this is difficult at best.

The Layout Problem

The first problem is layout. The Visual C++ dialog editor only allows you to edit a single dialog resource at a time, so somehow the child dialog, and the parent space which it's supposed to fit into, must be edited separately and aligned at run time.

There are basically three ways to do this. Perhaps the simplest method is to declare, early in the design, what the size and position of the child's client area will be. The chief drawbacks of this method are that it ignores visual editing tools like those in Visual C++, and casts precise details in concrete far too early in the development cycle.

A better approach is to have the child dialog size and position itself at run time to fit a space dictated by the parent. One simple method is for the parent dialog to include a static frame control, without the **WS_VISIBLE** flag, which gives a resizable frame on the parent at design time, and an invisible rectangle at run time for the child to position and scale itself to fit.

Perhaps the best approach, generally, is for the parent to scale itself to encompass its children. We can see how this works with existing MFC classes. The **CPropertySheet** dialog, for instance, sizes itself to encompass the largest **CPropertyPage** object that has been added to it, and the number of rows of tabs required. We could extend this idea in our own classes. For example, we could create a modal dialog whose constructor works out how many of each of three different sorts of child dialog will be needed, and as it calculates their tiled positions, it also calculates the overall size of the frame needed to hold them.

The Message Problem

A second problem is that a modeless dialog gets its messages dispatched from somebody else's message loop. It doesn't handle accelerators or participate in the tab order unless one of its children has the focus. In the attempt to fix this, Windows 95 has introduced two new window-style flags.

The first flag, **WS_EX_CONTROLPARENT**, tells Windows to include children of this window in the tab order. When Windows is scanning a group of sibling windows, looking for the next or previous window with the **WS_TABSTOP** style, it will scan 'down a level' and include children of any windows with this style. Unfortunately, this is an extended style (**WS_EX_***), so storing it in the template resource requires a resource of type **DIALOGEX**, rather than **DIALOG**.

The resource editor will create a **DIALOGEX** template resource under three circumstances:

- If there are any extended styles on the dialog.
- If you've specified weight, italic, or charset attributes on the dialog's font.
- If any control in the dialog template has a **DWORD** control ID, a help ID, or any extended style. (Although this last item is not documented by Microsoft at the time of writing.)

Unfortunately, the new common control classes, including the property page and property sheet controls, don't know how to handle **DIALOGEX** templates. If you try to create a **CPropertySheet** containing a **CPropertyPage** with this style set, you'll get an assertion failure from `Dlgcore.cpp`. (If you ignore the assertion failure, you'll find that the property page doesn't look right. The code that sizes the sheet to fit the largest page and the code that extracts the page caption to use as a tab label, can only work with **DIALOG** resources.)

Luckily, we have an alternative. A new dialog style, **DS_CONTROL**, duplicates the functionality of **WS_EX_CONTROLPARENT** for child dialogs and has an equivalent effect on the system code for dialogs, which translates accelerator keystrokes.

In addition to the special association of the *Enter* and *Escape* keys with the default (**IDOK**) and cancel (**IDCANCEL**) buttons, the system recognizes control captions containing the ampersand (**&**) character and implements accelerator keystrokes for the appropriate controls. Accelerators are not handled by MFC's **PreTranslateMessage()** method for modeless dialogs, but this style allows a modal parent to handle them. You should always enable this style for templates for property pages.

Additional Features of Property Pages

There's a bit more to a property page than just a dialog template with the **WS_CHILD** and **DS_CONTROL** styles set.

The usual way to set the text for the page's tab is to specify the **WS_CAPTION** style, so that the property sheet control interprets the caption strings of property pages as label text for the tabs it displays. To set **WS_CAPTION** and enter the label text, you must set the border of the dialog template to something other than None.

The Microsoft documentation specifies that the border should be Thin, but since the border won't actually be displayed in the property sheet, any setting but None will do, and this works fine if every page has its own template. A little later in this chapter, we'll show you another way to set the tab text, which allows multiple pages to share a template but have distinct tabs.

Unlike 'vanilla' modeless dialogs, property pages are not constructed by the **CreateDialog()**, **CreateDialogParam()**, **CreateDialogIndirect()** or **CreateDialogIndirectParam()** API calls. Instead, they use a new **CreatePropSheetPage()** API call, implemented in `Comctl32.dll`. This function expects a pointer to a **PROPSHEETPAGE** structure, so it's not surprising to find that MFC's **CPropertyPage** class has a public member, **m_psp**, which is an instance of just such a structure.

In theory, you could modify some aspects of how your property page behaves by manipulating this structure. In practice, if you're using MFC and dialog templates, MFC will initialize this structure appropriately, so there's very little reason to do this.

Here we find one of the areas where the implementation of a new control in `Comctl32.dll` differs from its implementation in an earlier version of MFC. The **PROPSHEETPAGE** structure contains a **dwFlags** member. One of the possible flag bits is **PSP_HASHELP**, used to tell the containing property sheet to enable its Help button when this page is displayed, and to send it a **PSN_HELP** notification message when that button is clicked (we'll discuss notification messages more thoroughly a bit later on). This lets a page provide its own help, which is necessary when the page is being inserted as an extension into a sheet provided by some other module.

MFC, however, intercepts any **PSN_HELP** notification sent to a **CPropertyPage** object and redirects it as a **WM_COMMAND|ID_HELP** message to the page window instead, and through the framework's command routing until a handler for it is found, in the usual way. You can provide page-specific help by handling this message at the page level.

```
BEGIN_MESSAGE_MAP(CMyPage, CPropertyPage)
   //{{AFX_MSG_MAP(CMyPage)
   //}}AFX_MSG_MAP
   ON_COMMAND(ID_HELP, OnHelp)   //   Add this
END_MESSAGE_MAP()
```

You might expect then, that MFC would automatically show the Help button on the property page, and enable or disable it via the usual **ON_UPDATE_COMMAND_UI()** mechanism. It doesn't. You'll still need to set the **dwFlags** member of the **PROPSHEETPAGE** structure appropriately. The button is shown only if some page has its **PSP_HASHELP** flag set (or if the property sheet has the **PSH_HASHELP** flag set in the **dwFlags** element of its **m_psh** member). It's enabled or disabled on a page-by-page basis, according to the flag for the active page.

```
MyPropPage.m_psp.dwFlags = MyPropPage.m_psp.dwFlags | PSP_HASHELP;
```

Two other interesting flags used with the **PROPSHEETPAGE** structure relate to the tab for the property sheet. If the **PSP_USETITLE** flag is set, the text for the tab is taken from the **pszTitle** member of the structure. This is useful if you want to have several pages in the same sheet which use the same dialog template, but obviously should have distinct tab labels. If you set this flag and member, your dialog template doesn't need a caption (although it can be handy as a default), and therefore may not need a border.

Another member of the structure holds a handle or resource ID to an icon to be used on the tab; it's a union, and there are flags to tell which type of value it is. The icon will be displayed at 16x16 pixel size, so it's a good idea to create it at this size. If you use the default 32x32 pixel size, you're unlikely to be pleased with what actually appears on the screen.

CPropertyPage Functions

When you create a **CPropertyPage**-derived class, you inherit ten overridable methods for interacting with a containing property sheet. These are handlers for the various **PSN_*** notification messages which can be sent to a page. This doesn't include **PSN_HELP** which, as we mentioned earlier, is handled separately.

Two more functions, **CancelToClose()** and **SetModified()**, are simple wrappers that send **PSM_*** messages to the page's parent window, presuming it to be a property sheet. Some other messages you might wish to send to the parent sheet are wrapped as methods of the **CPropertySheet** class, so if you want to use the framework, rather than the raw **SendMessage()** API for these, you'll need to obtain a **CPropertySheet*** pointer to the parent.

It would be nice if **CPropertySheet::AddPage()** stored such a pointer in a **CPropertyPage** data member, but it doesn't. You could save it there when you call **AddPage()**, but you'll have to add the member to your derived page class by hand. If you know that your page is only ever used as a child of a property sheet, you can use **GetParent()** to get a **CWnd*** and cast it to a **CPropertySheet***.

The lucky thirteenth inherited method has the useful-looking name **QuerySiblings()**, but the documentation for this function is terse beyond the point of usability. It turns out that this is a method whereby a page can pass a query to its sibling pages and get notified of the result. It's another

SendMessage() wrapper, this time sending **PSM_QUERYSIBLINGS** to the parent sheet. What happens there is rather unusual.

When a property sheet receives a **PSM_QUERYSIBLINGS** message, it starts sending it to its child pages, until one of them returns non-zero, or it runs out of pages. The non-zero result, if there is one, is returned to the caller. There's no indication of the order in which the children receive the message (although the page order is an obvious choice), and since there's no obvious way for the sheet to know who sent the message, it's prudent to assume that the page that sent the query message originally may also receive it.

For this reason, it makes sense for the page to pass its own window handle in **wParam**, and check for that in its message handler. That leaves **lParam** in which to pass whatever information a 'query' requires. This is big enough to hold a pointer to an instance of some arbitrary class, so a query structure may be as complex as necessary.

Note that although **CPropertyPage** provides the **QuerySiblings()** function to send this message, it doesn't provide a message handler for **PSM_QUERYSIBLINGS**. Since ClassWizard offers no help, you'll need to insert the **ON_MESSAGE()** macro by hand if you wish to handle this message.

Property Sheets

We've looked at property pages, so now let's turn our attention to property sheets. A property sheet is a special kind of dialog provided by `Comctl32.dll`, which can show any one of a number of subdialogs. Each subdialog, called a **property page**, is selected by the user clicking on one of the tabs on the sheet.

The size of the sheet doesn't come from a template, but is calculated based on the size of the first property page that it contains, the number of pages it contains (which determines how many tabs there will be), and the options selected. The MFC **CPropertyPage** class adds two useful properties to the basic control: it determines the size of the largest page contained in the sheet, and it doesn't actually create the dialog for any page until that page is displayed.

Property Sheets as Pop-up Dialogs

The most typical use of a property sheet is as a modal pop-up dialog. Create an instance of MFC's **CPropertySheet** class, use its **AddPage()** method to add some objects of classes you have derived from **CPropertyPage**, then call **DoModal()**.

If you want the property sheet to be modeless, the process is similar. You should create a **CPropertySheet** object dynamically with **new**, make it visible by calling **Create()** and destroy it with **delete** when it's no longer needed. One problem you'll encounter is that a modeless property sheet doesn't provide OK and Cancel buttons, so you need to provide some other way to determine when the property sheet should be destroyed. (In fact, you can cheat MFC into displaying these buttons even for modeless sheets, as you'll see later in the chapter.)

It's also important that the page objects which the sheet contains must continue to exist for the life of the sheet. This is a good reason to derive a class from **CPropertySheet** for any modeless property sheet in your application, and instantiate its page objects as members of that derived class, so that they get automatically constructed and destructed for you.

The Visual C++ Component Gallery includes a wizard to add a property sheet derivative and its pages to

your project. Its main advantage is that it can optionally generate code for a property sheet class which includes a preview window, to the right of the property pages, accessible as a member of the **CPropertySheet**-derived class. The intention is that you may override the preview window's **OnEraseBkgnd()** and **OnPaint()** methods to reflect the settings of controls on the property pages —the code to actually do this is left as an exercise! The generated code simply creates the window and resizes the sheet to include it in the visible area.

The page classes created by the wizard share a single header and a single source file, and by default are assigned to names and resource IDs in simple numerical order. You may find that this fits well with your approach to project design, but it requires that you already know how many pages you'll want and what they will be called. Personally, I find that I prefer to design my page template, use ClassWizard to create the page class, then construct the sheet object by adding pages at run time.

Property Sheet Notifications to Pages

Most of the code associated with a property page is there to handle notification messages. Some of these come from the controls on the page, but there are a number of them that come from the property sheet to its pages, most often, specifically to the page currently visible. These notifications arrive as **WM_NOTIFY** messages whose **lParam** value is a pointer to an **NMHDR** structure whose **code** member identifies the actual notification. Your code will not generally need to deal with the **NMHDR** structure directly, because the MFC framework dispatches notifications to handler functions which you can override.

It has been common for programmers to return message results from their dialog procedures by simply returning them as the function result, when in fact they should be stored in the dialog's extra bytes using the **SetWindowLong()** API call. 16-bit Windows included code to compensate invisibly for this, so most programmers had no idea that they were doing anything wrong.

The Win32 **WM_NOTIFY** message cannot be handled this way; the value to be returned to whoever called **SendMessage()** to issue the notification *must* be stored properly, and the function return from the dialog procedure is treated completely separately. Several recent books and magazine articles have labored this point, but this shouldn't worry MFC programmers; we return a function result from the handler, often the result of invoking the base class handler in addition to our own code, and MFC takes care of storing it correctly and returning a suitable dialog procedure result.

Activation and Deactivation Messages

When a page is about to be displayed, it will receive the **PSN_SETACTIVE** notification, which is handled by the **OnSetActive()** method. Note that while the page will only receive a single **WM_INITDIALOG** message when it is created, it can receive the **PSN_SETACTIVE** notification many times as the user navigates between the pages of a property sheet. Since the page's child dialog is just being hidden and shown, you shouldn't normally need to take any action on this notification, but it may be appropriate to set the focus to a particular control (if you don't want it to go to the first control), and there may be other states which you always want to reset on entry to the page.

Correspondingly, the page will receive the **PSN_KILLACTIVE** notification, handled by the **OnKillActive()** method, when the user has chosen to navigate away from the page. It's probably appropriate to call **UpdateData(TRUE)** here, to capture the state of the page's controls in member variables, and this is also a good point to validate the control contents. You can reject the request and retain the active page if some control is not currently valid by returning a result of **0**.

The Apply Notification

If the user clicks on the Apply button of the property sheet, the **`PSN_APPLY`** notification will be sent to every page that has been created, where it will be handled by the **`OnApply()`** method. Unless it's actually possible to undo the changes made by **`OnApply()`**, a page which actually changes anything is apparently expected to call **`CancelToClose()`** to disable the Cancel button and change the text on the OK button to Close.

This doesn't seem to work as expected, and stepping through code with the debugger, shows that the buttons change as they're supposed to, and then promptly change back to their default settings. It doesn't only seem to happen to user-written code either. Even system property pages are afflicted with this problem!

In practice, it appears that this particular wrinkle in the new Windows user interface hasn't been thoroughly thought out. It's true that once a user has clicked on the Apply button, it's unlikely that clicking on the Cancel button will undo those changes, and it would be a nice gesture to somehow provide feedback to that effect. But what if the user makes some changes, clicks on Apply, and then makes some more changes? Should Cancel undo everything, or just what has been changed since the last Apply?

This suggests that a properly designed property sheet dialog needs four buttons, not three. Just as the action of the OK button is like the Apply button, but dismisses the dialog, the action of the Cancel button should add dismissing the dialog to the action of an Undo button, which would roll back all changes since the previous Apply action. Just as the Apply button is only enabled when a change has been made since the last Apply action, the Undo button would be enabled only when a change has been made, or at least one reversible Apply action has been performed.

Anyway, when the user clicks on the sheet's OK button, the current page receives the **`PSN_KILLACTIVE`** notification, and can suppress the rest of the OK action by returning `0`. If it returns a non-zero result, the **`PSN_APPLY`** notification will be sent to every page that has been created, just as if the user had pressed the Apply button. Since the base class implementation of the **`OnApply()`** handler invokes **`OnOK()`**, that method will also be called, but it would be called if the Apply button were clicked. The MFC documentation seems to suggest that **`OnOK()`** can be invoked by actions which do not invoke **`OnApply()`**, and while this may have been true of some previous implementation, it isn't currently correct. Using the Common Controls, both cases arrive as **`PSN_APPLY`** notifications and are dispatched through **`OnApply()`** calling **`OnOK()`**.

The only distinction is that **`OnApply()`** may, like **`OnKillActive()`**, return **`FALSE`** to prevent the sheet from being dismissed. Control validation really should have been done in **`OnKillActive()`**, so the only time that a page should need to do this is if some setting on the page has been rendered inappropriate by a selection subsequently made on some other page. Since the page encountering this situation is not the current page, it should call **`GetParent()->SetActivePage(this)`** to activate itself so the user can correct the problem. (This could be a problem if multiple pages find themselves in this state, but it's very likely that the property sheet will stop sending **`PSN_APPLY`** notifications when one page indicates that it cannot comply.)

Similarly, when the user clicks on the Cancel button, each page is sent a **`PSN_QUERYCANCEL`** notification, handled by **`OnQueryCancel()`**. If no page returns **`FALSE`**, every page is sent a **`PSN_RESET`** notification. Again, the base class implementation of **`OnReset()`** calls **`OnCancel()`**, so both methods will be called and cleanup processing may be done in either place.

There are also some additional notifications that are specific to wizards, which we'll discuss later in this chapter.

Property Sheet Methods and Messages

We've already mentioned the **DoModal()**, **Create()** and **AddPage()** methods provided by **CPropertySheet** and these are sufficient for most purposes. **AddPage()**, though, is only one of a dozen or so wrapper functions provided by MFC, whose main functions are to send a message to the property page control implemented in `Comctl32.dll`.

SetTitle()

The **PSM_SETTITLE** message is sent to the sheet by **CPropertySheet::SetTitle()**. You should specify a title when you construct the **CPropertySheet**, but this message allows the title to be changed at any time.

CancelToClose()

The **PSM_CANCELTOCLOSE** message is sent to the sheet by **CPropertyPage::CancelToClose()**. As discussed in the previous section, the implementation of this message within `Comctl32.dll` appears to fall short of its documented purpose.

SetModified()

The **PSM_CHANGED** or **PSM_UNCHANGED** messages are sent to the sheet by **CPropertyPage::SetModified()**. As long as some page has sent **PSM_CHANGED** and has not rescinded it with **PSM_UNCHANGED**, the Apply button (if there is one) will be enabled. All created pages will receive the **PSN_APPLY** notification when the Apply button is pressed, even if they haven't sent this message.

GetTabControl()

The **PSM_GETTABCONTROL** message is sent by **CPropertySheet::GetTabControl()**, and returns a **CTabCtrl*** pointer to the tab control used internally by the property sheet. You could use this to add bitmaps or tooltips to the tabs. Cluttering your application's user interface is rarely a good idea, but if you decide you need to, you can. We'll discuss tab controls in more detail later in this chapter. Note that you don't create this tab control, so you can't replace it with your own class derived from **CTabCtrl**. If you need to customize it, you can't use a property sheet and must instead create your own dialog to contain your custom tab control and pages.

PressButton()

The **PSM_PRESSBUTTON** message is sent by **CPropertySheet::PressButton()** and can simulate a click on any of the property sheet buttons. It's probably not a good idea to try to simulate a click on a button that the current sheet does not have, but this may work as a way to invoke the functionality of the various buttons on a modeless property sheet. A related message, **PSM_APPLY**, is not wrapped by MFC but can be sent using **SendMessage()**. The separate message can occasionally be useful, because it returns a Boolean value indicating whether all of the pages in the sheet successfully applied their changes.

RemovePage()

There is a **PSM_REMOVEPAGE** message, wrapped by **CPropertySheet::RemovePage()**. While changing the number of pages in the sheet while it's visible is likely to confuse the user, there may be cases where it makes sense to add and remove pages rather than destroy and recreate entire sheets.

SetActivePage()

The **PSM_SETCURSEL** and **PSM_SETCURSELID** messages allow arbitrary navigation between pages in the sheet. Their message parameters are poorly documented, so use their wrapper method, **CPropertySheet::SetActivePage()**, which comes in two flavors; you can refer to the page you want either by index (zero-based) within the sheet, or by a pointer to a **CPropertyPage**.

PSM_RESTARTWINDOWS and PSM_REBOOTSYSTEM

Property pages are heavily used throughout Windows 95 to provide access to system settings, and a couple of special messages have been implemented in the property sheet to make this easier. If any page sends the **PSM_RESTARTWINDOWS** message to the sheet, the result returned by **DoModal()** will be **ID_PSRESTARTWINDOWS** instead of **IDOK**. Typically, a page might send this message when processing a **PSN_APPLY** notification.

If any page sends the **PSM_REBOOTSYSTEM** message to the sheet, it will return **ID_PSREBOOTSYSTEM** instead. In either case, it's up to the application which invoked the property sheet to recognize these special results and take some appropriate action. The usual approach has been to offer the user a message box, informing them that some of the changes just selected cannot take effect until Windows is restarted or the system is rebooted. The message box offers the user the opportunity to restart or reboot right now, or to continue to work without all of their selected changes in place.

Customizing Property Sheets

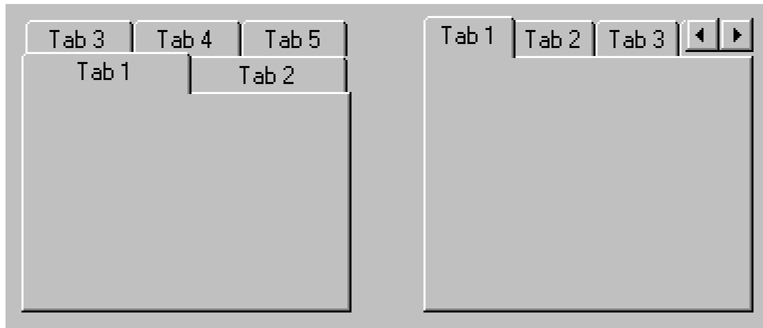
Just as **CPropertyPage** objects contain a **PROPSHEETPAGE** structure, **CPropertySheet** objects contain a **PROPSHEETHEADER** structure member, **m_psh**, and the behavior and appearance of the property sheet can be modified by manipulating the elements of this structure.

We said above that the **SetTitle()** method sent a **PSM_SETTITLE** message to the sheet. That's not quite the whole story, as it also sets the **pszCaption** member of the **PROPSHEETHEADER** structure. The **PSM_SETTITLE** message is only sent if the property sheet has been created and it has a valid window handle.

MFC appears to do a lot of work to preserve the interface exposed by previous implementations of **CPropertySheet**, while using the implementation provided by `Comctl32.dll`. The result is that directly modifying the flags in the **PROPSHEETHEADER** rarely works very well. In the most useful cases, the framework provides its own method which may include setting those flags, but updates other MFC structures as well.

For instance, there's a flag to control whether the sheet should show multiple rows of tabs, if there are too many pages for a single row, or whether a single row of tabs should be allowed to scroll left and right. Setting this flag directly doesn't work. **CPropertySheet** shadows this flag with a protected data member,

m_bStacked, and you can change its state from the default (**TRUE**) by calling **EnableStackedTabs(FALSE).**



As an aside, from a user interface design point of view, neither option is particularly desirable, because both of these behaviors break the visual metaphor of the tabs in some way. If at all possible, try to limit your tabs on any single property sheet to what will fit on a single row.

One flag which does work as expected is **PSH_NOAPPLYNOW**. If you set this flag, the property sheet will be displayed without an Apply button. You must decide whether this is suitable for your application.

The **PSH_PROPTITLE** flag is documented as providing the text Properties for in front of the supplied caption, but in fact it appends the word Properties to the *end* of the supplied caption. This feature can also be enabled if the flag value is supplied as a second argument to **SetTitle()**.

The **PSH_HASHELP** flag determines whether a Help button is visible on the property sheet; the **PSP_HASHELP** flags of the pages determine whether it's enabled or disabled. MFC will enable this flag if any page has its **PSP_HASHELP** flag set, so you should never need to set this flag directly unless you want a disabled help button at all times or if you plan to add sheets with **PSP_HASHELP** set at a later time.

MFC will set the **PSH_MODELESS** flag appropriately, depending on whether you call **Create()** or **DoModal()**. This, in turn, modifies the call to the underlying **PropertySheet()** API provided by `Comctl32.dll`.

Wizards

One of the flags in the **PROPSHEETHEADER** structure changes the behavior of the sheet radically, enabling several new messages and notifications. The **PSH_WIZARD** flag is set by the **SetWizardMode()** method of the sheet.

You've all seen wizards in action. A wizard guides the user through a series of modal steps in some process. Whereas the tabs of a normal property sheet allow the user to select pages at random, the wizard imposes a linear order, so that from any page, the user can only move forward to the next or back to the previous page.

To the programmer, the order of the pages is not quite so cast in stone. When the user clicks on the Back or Next button, the current page receives a notification: **PSN_WIZBACK** or **PSN_WIZNEXT**. MFC dispatches these notifications to the page's **OnWizardBack()** or **OnWizardNext()** method, and if this method returns **0**, the page will then receive a **PSN_KILLACTIVE** notification as the sheet navigates to the next page. The

page can return `-1` to either of these notifications to prevent the default page navigation.

Programmers frequently ask how they can override the default navigation, to select the next page to be shown based upon choices that the user has already made. The Microsoft documentation suggests that instead of `0` or `-1`, a page can return the 'identifier' (by which they mean the template resource ID) of some other page which is to be shown next. If the page returns anything but `-1`, and MFC cannot map the result to a known page template, the sheet navigates to the next or previous page, depending on which button was clicked.

Another approach, however, is to call **`CPropertySheet::SetActivePage()`** to tell the sheet to display the next page that you want the user to see, and then return `-1` to suppress the default navigation. **`SetActivePage()`** takes the ordinal number of a page within the sheet (starting from 0), which may be more convenient than using the template resource ID, especially as you may have several pages constructed from the same template. In either case, validation should be done in response to the wizard notification, before you attempt to navigate to the next page.

If you change the order of navigation dynamically like this, it's up to you to keep track of the path, so that the user's steps are properly retraced when they click the Back button. It's also up to you to ensure that the path doesn't loop back on itself, creating a bubble from which the Cancel button is the only escape.

When the last page in a wizard is shown, the Next button should be replaced by a Finish button, and because you can override the default page navigation, MFC doesn't do this for you automatically. **`CPropertySheet::SetWizardButtons()`** lets you hide or show the Back button and show either a Next button, a Finish button, or a disabled Finish button. Note that if you replace the Next button with a Finish button, but the user clicks on the Back button, the Finish button doesn't automatically revert to Next.

You might want to micro-manage the button state of the wizard, but a simpler approach is to conclude your wizard with a final page that offers no input controls. In this page's handler for **`PSN_SETACTIVE`**, call **`CPropertySheet::SetFinishText()`** to replace the Next button with an enabled Finish button, and then disable or hide the Back button. At this point, the user can no longer go back and change their mind; they can only click either on Cancel or Finish.

One expects the current page to receive a **`PSN_KILLACTIVE`** notification message when the user clicks on the Finish button, but it has been reported that this notification is not sent when using Win32s, which means that, by default, DDV/DDX processing of the page's controls won't be done. Again, you can code around this, or you can simply finesse the issue by supplying a final page with no input controls.

When the user clicks the Finish button, the current page receives a **`PSN_WIZFINISH`** notification and the framework calls the page's **`OnWizardFinish()`** method. This should always return **`TRUE`**, because the framework will take care of destroying the sheet and returning **`ID_WIZFINISH`** to the application. Of course, the **`PSM_RESTARTWINDOWS`** and **`PSM_REBOOTSYSTEM`** messages may be sent to wizards just as to ordinary property sheets, and will override this return value as appropriate. This is typically done by wizards that guide the user through system configuration or driver installation.

Tab Controls

A standard property sheet uses another of the new common controls— the tab control— to manage display of the tabs and navigation between the pages. You can build a similar effect on any dialog, including a **`CFormView`**, which, you'll recall, is a borderless modeless dialog that fills the client area of a frame window.

Standard Tab Control

The tab control provides a static rectangular frame with rows of tabs or buttons across the top. Each button is associated with an 'item' structure, which may contain label text, an index into an image list and a 32-bit data value which may be an arbitrary pointer. The tab control knows nothing about objects to which these item structures refer. It sends event notifications and owner-draw requests to its parent, which will normally be some class that you have derived from `CDialog`, and the parent is responsible for handling the notification itself or reflecting it to some child object.

In a property sheet, the sheet itself is the parent dialog and the pages are its children. Some of the notifications which the sheet sends to its pages reflect user interaction with the tab control, and others originate from additional buttons on the sheet.

The property page makes a good model class for child objects managed by a tab control. When the user clicks on a tab, the tab control sends its parent two notifications. The `TCN_SELCHANGING` notification identifies the current tab being switched away from, and corresponds to the `OnKillActive()` handler provided by `CPropertyPage` objects. Similarly, the `TCN_SELCHANGED` notification identifies the tab which is becoming active, and corresponds to the `OnSetActive()` handler.

You shouldn't be surprised to learn that the result returned to the `TCN_SELCHANGING` notification can suppress the change if some field fails validation. The return result, however, has the opposite sense of that specified for the result of `OnKillActive()`. In other words, `OnKillActive()` uses a return value of `0` (or `FALSE`) to prevent the active page from changing, whereas the return value for a `TCN_SELCHANGING` handler should use `TRUE` to prevent the selection from changing.

Using a tab control is a little more work than using `CPropertySheet`. MFC manages the creation of pages in a property sheet as they become necessary, and sizes the sheet to hold the largest page. The property page control extracts captions from dialog resources and constructs the tab control's image list to contain any icons specified in the `PROPSHEETPAGE` structures.

The child objects managed by the tab control certainly don't have to be property pages; they don't even need to be dialogs. It's normally convenient to use property pages, though, so that they are derived from a class which provides virtual `OnSetActive()` and `OnKillActive()` methods. It would be a bit more convenient if their base class exposed a virtual `Create()` method that accepted a pointer to the parent (the tab control) and supplied the template ID already defined in the class by ClassWizard.

This isn't too hard to add ourselves. We just need to create a class derived from `CPropertyPage` which adds the needed method. We can let ClassWizard generate our page classes as if they were derived directly from `CPropertyPage`, and then edit the source files to insert our new class into the inheritance hierarchy.

Property Sheets as Child Dialogs

A much easier way to get a tabbed subdialog effect managing a group of `CPropertyPage` objects is to use an embedded child `CPropertySheet`. We could use ClassWizard to derive a `CPropertySheetControl` class from `CPropertySheet`, then override the constructor (since this property sheet will not have a caption) and `OnInitDialog()`.

By default, ClassWizard will provide two constructors for this class: one which takes its caption in the form of an **LPCTSTR**, and one that takes the **UINT** identifier of a string resource. Since our derived class won't need a caption, you can remove the first of these completely and remove the first argument from the second constructor. Since the other two arguments are optional, you can use this as a default constructor for the class.

```
// PropertySheetControl.h : header file
//
// CChildPropertySheet

class CPropertySheetControl : public CPropertySheet
{
   DECLARE_DYNAMIC(CPropertySheetControl)

// Construction
public:
   CPropertySheetControl(UINT iSelectPage = 0);

// Overrides
   // ClassWizard generated virtual function overrides
   //{{AFX_VIRTUAL(CPropertySheetControl)
   public:
   virtual BOOL OnInitDialog();
   virtual int DoModal();
   //}}AFX_VIRTUAL
   virtual BOOL Create(CWnd* pParentWnd = NULL, DWORD dwStyle = (DWORD)-1,
                       DWORD dwExStyle = 0);

// Implementation
public:
   virtual ~CPropertySheetControl();

   // Generated message map functions
protected:
   //{{AFX_MSG(CPropertySheetControl)
      // NOTE - the ClassWizard will add and remove member functions here.
   //}}AFX_MSG
   DECLARE_MESSAGE_MAP()
};
```

In fact, you *could* remove the second argument as well. These child dialogs will always be invoked using the **Create()** method, and the parent window will be specified as a parameter at that point, so you can just pass the default values, an empty string for the caption and **NULL** for the parent window pointer, to the base class constructor. In the implementation we have provided, we have overloaded both the **DoModal()** and **Create()** methods, to ensure that they're not invoked inappropriately.

Note that the **Create()** overload provided by ClassWizard is an overload for **CWnd::Create()**, and not for **CPropertySheet::Create()**, which takes a different set of parameters and invokes a different base-class function. If you use ClassWizard to create the override function, and then edit the parameters and the code, remember to move the declaration outside of the **AFX_VIRTUAL** brackets so that ClassWizard will leave your changes alone.

```
int CPropertySheetControl::DoModal()
{
   ExceptXX(TRUE, "This class should not be invoked modally");
   return  IDCANCEL;
}

BOOL CPropertySheetControl::Create(CWnd* pParentWnd, DWORD dwStyle,
   DWORD dwExStyle)
{
   AssertXX(pParentWnd, "Parent window must be specified");
   AssertXX(dwStyle & WS_CHILD, "Control should have WS_CHILD style");
```

```
   if (!(pParentWnd && (dwStyle & WS_CHILD)))
      return  FALSE;
   return  CPropertySheet::Create(pParentWnd, dwStyle, dwExStyle);
}
```

Now that the sheet control class has a default constructor, instances of this class may be added to other dialog classes as data members. Typically, the individual pages derived from **CPropertyPage** will also be created as dialog class data members, and **AddPage()** will be used to add them to the sheet control. As data members, these will all get cleaned up when the dialog object is destroyed. The remaining step is to create and show the sheet control when the dialog is displayed.

The dialog editor doesn't know anything about the **CPropertySheetControl** class, so it has no way to let you create an instance of the class on a dialog template. What you can do, though, is create a rectangular control as a placeholder and use it to lay out the position which the sheet control will occupy at run time; a group box works well for this. Just pass in this control as the parent when you create the property sheet control.

However, don't make this proxy control visible, unless you want to use it to put an edge around the outside of the whole control. (Since the pages and tabs will be outlined by default, this is unlikely to look very professional in a release build, but it may come in handy during development, as you try to fine-tune the alignment of the sheet.)

You have two options when you're aligning the sheet control with the proxy control. The top left corner of the sheet should definitely be mapped to the top left corner of the proxy, but there is the question of whether the sheet should be clipped to the size of the proxy, or whether the sheet should remain unclipped and the size of the parent dialog be adjusted to accommodate it. In the example (**PSControl**), the sheet has been clipped to the proxy, but this complicates the problem of ensuring that the pages, and thus the sheet itself, will be properly visible.

In the overloaded sheet control class, the **OnInitDialog()** method has three basic responsibilities. It must call the base class **OnInitDialog()** method, of course. It must reparent the sheet dialog window created as a child of the invisible proxy to be a visible child of the dialog itself in the same position as the proxy, and it must fix up the sheet dialog's styles so that controls on its pages may participate in the dialog's tab order. .

You'll recall that the last of these depends on the **DS_CONTROL** and **WS_EX_CONTROLPARENT** styles. You might think that it would make sense to specify these styles in the call to **Create()**, but this doesn't work. The property sheet creation code gets caught in a loop, sending an infinite stream of **WM_GETDLGCODE** messages. This is probably a side effect of the fact that the parent and position of the sheet window have not yet been set correctly, so this work needs to be done in **OnInitDialog()**.

The tab control embedded within the property sheet is normally created, at least in a modeless sheet, with the **WS_TABSTOP** style, which allows the user to use the keyboard to navigate amongst the tabs. This is a useful feature. In the example, we have explicitly turned off this style, to demonstrate how you might customize its behavior.

```
BOOL CPropertySheetControl::OnInitDialog()
{
   CPropertySheet::OnInitDialog();
   CRect rr;
   CWnd* pOldParent = GetParent();
   CWnd* pNewParent = pOldParent->GetParent();

   pOldParent->GetWindowRect(&rr);
```

```
    pNewParent->ScreenToClient(&rr);
    SetParent(pNewParent);
    SetWindowPos(pNewParent,
       rr.left, rr.top, rr.Width(), rr.Height(),
       SWP_NOZORDER | SWP_SHOWWINDOW /* | SWP_NOSIZE */);
    ModifyStyle(0, DS_CONTROL, 0);
    ModifyStyleEx(0, WS_EX_CONTROLPARENT, 0);
    GetTabControl()->ModifyStyle(WS_TABSTOP, 0, 0);
    return  TRUE;
}
```

When Should You Use a Tab Control?

Given that a child property sheet control is a relatively simple thing, and so much easier to use than a raw tab control, when is it better to use a tab control? In fact, it turns out that there are several styles supported by the tab control which are not supported and exposed by the **CPropertySheet** class.

The **TCS_BUTTONS** style draws the tabs as complete rectangular buttons, rather than as folder tabs. A slight change to the drawing code produces a significantly different effect, but whether this effect is desirable or useful is debatable. (If the currently selected button stays depressed, the effect is reminiscent of the system taskbar.)

The **TCS_OWNERDRAWFIXED** style is a bit more promising. When this style is selected, the parent dialog will receive an initial **WM_MEASUREITEM** message to set the size of the tabs in this control, and thereafter will receive a **WM_DRAWITEM** message whenever a tab is being painted. Unfortunately, the **WM_DRAWITEM** message refers only to the tab 'face' and not to its outline, so this isn't sufficient to draw tabs along the side or bottom of the page window.

The **TCS_TOOLTIPS** style tells Windows that there is tooltip text associated with each control tab. The tips are actually stored in a tooltip control, associated with the tab control by sending it a **TCM_SETTOOLTIPS** message. This is particularly useful for providing a text label with tabs that would otherwise show only an icon or an owner-draw graphic.

Creating a Generic Dynamic Dialog Class

In this section we'll introduce a generic dynamic dialog class that will enable you to create, display and hide controls inside a window at run time.

You may have seen dynamic dialogs in action and wondered how it's done. An example of a dynamic dialog is Developer Studio's Project Settings dialog. When you select an item from the Category: combo box on the C/C++ page, a new set of controls appears on the screen and the old ones disappear. In this section, we'll show you how to do this through the use of general purpose classes using MFC, which will enable you to create controls anywhere, anytime at run time.

This section presents an MFC-based dynamic dialog class, **`CDynamicDialog`**, along with a test application which demonstrates the use of these classes.

Most of the techniques you see here have been covered by several authors in one way or another, but most tend to use a **hard-coded** approach, where you end up using that most beloved technique of programmers: copy and paste. What is presented here is a set of generic classes that you can use many times.

There are several ways to create controls at run time. One approach would be to do the following:

```
CMyDialog::OnInitDialog()
{
   m_ctlEdit.Create(WS_VISIBLE|WS_CHILD|WS_TABSTOP|WS_BORDER,
                              cltRect, this, IDC_EDITCTL);
   m_ctlStatic.Create("Name:", WS_VISIBLE|WS_CHILD|SS_LEFT,
                              ctlStaticRec, this, -1);
}
```

The problem here is that if you need to recreate the above controls in another dialog, you have to copy and paste. A more general approach would be to define a structure that defines the characteristics of the controls and use it in more than one place to create them. The problem with either approach is that you have to manually type in the coordinates of the controls, and if you add or remove one control, you have to recalculate the coordinates for each control.

It would be much more useful if we could use the dialog editor to define these controls and then display them at run time, so that's exactly what we'll do.

Most Windows developers use a resource editor to create a dialog that is automatically saved in the project's `.rc` file. Afterwards, a class derived from **`CDialog`** (for example **`CMyDialog`**), is created and the developer displays it as a modal dialog using the following statements:

```
CMyDialog dlg(this);
dlg.DoModal();
```

You usually don't care how the dialog is saved in the `.exe` file, but later on you'll see that knowing how the dialog is actually saved in the file and its data format will enable you to retrieve the characteristics of the controls, and create them dynamically.

Dynamic dialogs are nothing more than a collection of window controls whose characteristics are retrieved at run time, created dynamically and displayed in a predefined dialog.

Resource Storage

Dialogs are usually defined using a resource editor and saved in a resource file which includes other definitions (such as menus). Win32 binary resources are stored in Unicode form, so to manipulate string data you would usually use the **WCHAR** data type. Every dialog box in a resource file is of variable size, with a header, followed by one or more control data structures.

The dialog resource header has the following format:

Data Type	Name	Description
DWORD	dwStyle	Specifies the dialog window styles.
DWORD	dwExtendedStyle	Specifies the extended dialog window styles (used with **CreateWindowEx()** function).
WORD	NumberOfItems	Specifies the total number of controls in the dialog.
WORD	x	Specifies the position of the left side of the dialog window.
WORD	y	Specifies the position of the top of the dialog window.
WORD	cx	Specifies the width of the dialog box.
WORD	cy	Specifies the height of the dialog box.
Name or Ordinal	MenuName	Specifies the ordinal ID or menu name.
Name or Ordinal	ClassName	Specifies the class ID or name.
WCHAR	szCaption[]	Specifies the dialog caption.
WORD	wPointSize	Specifies the dialog's font point size (this is present only if the **FONT** statement was specified in the dialog definition).
WORD	szFontName	Specifies the dialog's font name (this is present only if the **FONT** statement was specified in the dialog definition).

The Win32 SDK defines a structure, called **DLGTEMPLATE**, which includes the first seven fields in the above table. In other words, **DLGTEMPLATE** is defined as follows:

```
typedef struct {
    DWORD style;
    DWORD dwExtendedStyle;
    WORD  cdit;
    short x;
    short y;
    short cx;
    short cy;
} DLGTEMPLATE;
```

We'll be using this in our **CDynamicClass**.

The data for each window control starts after the header. Please note that data for each control starts on a **DWORD** boundary which must be considered when moving from one control to the next. The control data format is very similar to that for the dialog header:

Data Type	Name	Description
DWORD	dwStyle	Specifies the control window styles.
DWORD	dwExtendedStyle	Specifies the extended control window styles (used with CreateWindowEx()).
WORD	x	Specifies the position of the left side of the control.
WORD	y	Specifies the position of the top of the control.
WORD	cx	Specifies the width of the control window.
WORD	cy	Specifies the height of the control window.
WORD	wId	Specifies the control window ID.
Name or Ordinal	ClassId	Specifies the ordinal ID or class name.
Name or Ordinal	Text	Specifies the string ID or text.
WORD	nExtraStuff	Extra padding.

There's also a structure call **DLGITEMTEMPLATE**, which contains the first eight fields in the control data block. In other words, **DLGITEMTEMPLATE** is defined as follows:

```
typedef struct {
    DWORD style;
    DWORD dwExtendedStyle;
    short x;
    short y;
    short cx;
    short cy;
    WORD  id;
} DLGITEMTEMPLATE;
```

We're going to use this structure to retrieve the control window data and create it at run time using **CDynamicDialog::CreateControl()**.

CDynamicDialog

CDynamicDialog allows you to create a subdialog, and add it to an existing dialog at run time (i.e. dynamically). You can use this generic class with any class derived from the MFC **CWnd** class, such as **CFormView** and **CPropertyPage**.

Here's what the class definition looks like; we'll go on to look at some of the more important functions in detail in subsequent sections.

```
class CDynamicDialog : public CObject
{
   DECLARE_DYNCREATE(CDynamicDialog)

public:
   virtual ~CDynamicDialog();
   CDynamicDialog();
   CDynamicDialog(CWnd* pWndParent, UINT uResourceID,
         UINT uPlacementWndID,
         LPCCLISTCTLTEXT arListCtlText = NULL);
   CDynamicDialog(CWnd* pWndParent, UINT uResourceID,
         int nXOffset = 0, int nYOffset = 0,
```

```
         LPCCLISTCTLTEXT arListCtlText = NULL);
   CDynamicDialog(const CDynamicDialog& a);
   CDynamicDialog&  operator=(CDynamicDialog& b);

// Attributes
public:
   void  Initialize(CWnd* pWndParent, UINT uResourceID,
         UINT uPlacementWndID = 0, int nXOffset = 0, int nYOffset = 0,
         LPCCLISTCTLTEXT arListCtlText = NULL);
   void  SkipOrdinalOrTextField(LPBYTE* pplDlgResource);
   BOOL  CreateControl(LPBYTE* ppDlgRes);
   BOOL  CreateDynamicDialog();
   void  DestroyDialog();
   BOOL  ShowDialog(BOOL bShowControls);
   void  ShowControls(BOOL bShowControls);
   void  GetXYOffset(int& xOffset, int& yOffset);
   void  AddStringList(LPCCTLINFO pCtlInfo) ;
   void  MapDialogRectEx(CRect& rectWndCtl);
   BOOL  IsDialogVisible();

// Implementation
protected:
   CWnd*       m_pWndParent;        // Ptr to the parent dialog window
   UINT        m_uResourceID;       // Dialog resource ID
   UINT        m_uNumOfCtls;        // Total num of controls in the dialog
   UINT        m_uPlacementWndID;   // Controls starting location
   CPtrList    m_ctlPtrList;        // List of controls information
   BOOL        m_bCreated;          // Created flag
   LPCCLISTCTLTEXT m_arListCtlText;
   int         m_nXOffset;          // X offset
   int         m_nYOffset;          // Y offset
   BOOL        m_bShowState;        // Has the dialog already been shown?

};
```

Construction and Initialization

There's nothing very special about the construction of this class, except that all the action taken during construction is handled by an initialization routine, which saves code being duplicated.

```
CDynamicDialog::CDynamicDialog(CWnd* pWndParent, UINT uResourceID,
      UINT uPlacementWndID, LPCCLISTCTLTEXT arListCtlText)
{
   Initialize(pWndParent, uResourceID, uPlacementWndID,
         0, 0, arListCtlText);
}
```

The initialization routine, in turn, just fills in the data members of the object from the arguments supplied to the constructor.

```
void CDynamicDialog::Initialize(CWnd* pWndParent, UINT uResourceID,
      UINT uPlacementWndID, int nXOffset, int nYOffset,
      LPCCLISTCTLTEXT arListCtlText)
{
   m_pWndParent      = pWndParent;
   m_uResourceID     = uResourceID;
   m_uPlacementWndID = uPlacementWndID;
   m_bCreated        = FALSE;
   m_arListCtlText   = arListCtlText;
   m_nXOffset        = nXOffset;
   m_nYOffset        = nYOffset;
   m_uNumOfCtls      = 0;
   m_bShowState      = FALSE;
```

```
    // Retrieve the X & Y offsets.
    GetXYOffset(m_nXOffset, m_nYOffset);
}

void CDynamicDialog::GetXYOffset(int& xOffset, int& yOffset)
{
    // See if a placement window ID has been specified
    if (m_uPlacementWndID)
    {
        CWnd* pWnd = m_pWndParent->GetDlgItem(m_uPlacementWndID);
        CRect rectWnd;
        pWnd->GetWindowRect(rectWnd);
        m_pWndParent->ScreenToClient(rectWnd);
        xOffset = rectWnd.left;
        yOffset = rectWnd.top;
    }
}
```

There are two parameters which deserve some explanation: **uPlacementWndID** (the 'placement window') and **arListCtlText** (the 'text list').

If a window ID is given as the 'placement window' parameter, the origin of this window will be used to set the position of the dialog. **GetXYOffset()** retrieves the coordinates of the top left of the placement window.

The dialog editor can be used to create the controls which populate the dialog, and can provide initial state information for some, but not all of them. For instance, you can specify the position and size of a list box in the dialog editor, but you can't specify its default contents. This is where the text list comes in; it provides a means of specifying a list of string resources for a given control.

```
typedef class CListCtlText
{
public:
    CListCtlText(UINT uCtlID = 0, UINT* arResStringIDs = (UINT*)NULL)
        : m_uCtlID(uCtlID), m_arResStringIDs(arResStringIDs)
    {
        // Blank
    }
    UINT  m_uCtlID;
    UINT* m_arResStringIDs;
} CLISTCTLTEXT, FAR* LPCCLISTCTLTEXT;
```

Each **CListCtlText** object contains a control ID and pointer to a list of string resource IDs. When the dialog is created, the list of **CListCtlText**s provided to the constructor is used to initialize the controls in the dialog.

ShowDialog()

You'll have noticed that the construction process does nothing to actually create or display the dialog. In common with many other MFC classes which wrap Windows objects, creating the MFC object doesn't automatically create and display the Windows object; in this class, that function is performed by **ShowDialog()**. As shown in the code below, **ShowDialog()** creates the dialog if it hasn't previously been created, and then shows the controls.

```
BOOL CDynamicDialog::ShowDialog(BOOL bShowControls)
{
// Create the controls, if they have not been created.
    if (!m_bCreated)
    {
        if (!CreateDynamicDialog())
```

```
        {
           TRACE(_T("Unable to create the dialog..."));
           return FALSE;
        }
        m_bCreated = TRUE;
     }
     ShowControls(bShowControls);
     return TRUE;
  }
```

CreateDynamicDialog()

`CreateDynamicDialog()` is the routine where it all happens, and it's also where we use our knowledge of how dialog resources are stored in the program's `.rc` file.

To explain exactly what's going on, we'll look at **`CreateDynamicDialog()`** in sections. Our first action is to use the resource ID to find the dialog resource in the program's resources:

```
BOOL CDynamicDialog::CreateDynamicDialog()
{
   // 1 - Get a handle to the "m_uResourceID" dialog box
   HRSRC    hResource;      // Handle to the resource
   hResource = ::FindResource(AfxGetResourceHandle(),
                              MAKEINTRESOURCE(m_uResourceID),
                              RT_DIALOG);
```

Once we have that, we can load the resource and get a pointer to the first byte of the resource data:

```
   // 2 - Load the "IDD_IDENTIFICATION" resource
   HGLOBAL hData;          // Data handle
   hData = ::LoadResource(AfxGetResourceHandle(), hResource);

   // 3 - Get a pointer to the first byte of the resource;
   LPBYTE  pDlgRes;
   LPBYTE  pDlgResFirstByte;
   pDlgRes = pDlgResFirstByte = (LPBYTE)::LockResource(hData);
```

We're pointing at raw, binary data, so, to make sense of it, we copy it into one of the **`DLGTEMPLATE`** structures that we discussed earlier. Once we've done that, we can use the fields in the structure to access the dialog data:

```
   // Copy the data into the dialog template
   DLGTEMPLATE DlgTemplate;
   memcpy(&DlgTemplate, pDlgRes, sizeof(DLGTEMPLATE));

   // Set the number of controls in this dialog
   m_uNumOfCtls = DlgTemplate.cdit;
```

Once we've copied the standard information into the **`DLGTEMPLATE`** structure, we can look through the rest of the data in the resource to extract the other information that we're interested in. However, there are some things in this data which we aren't interested in, so we use **`SkipOrdinalOrTextField()`** to step past them to get to the data we need.

```
   // Move past the "DLGTEMPLATE" structure to lookup the following:
   //     1 - char szMenuName[]
   //     2 - char szClassName[]
   //     3 - char szCaption[]
   //     4 - WORD wPointSize;          ONLY if "DS_SETFONT"
   //     5 - char szFaceName[]         ONLY if "DS_SETFONT"
```

```
    pDlgRes += sizeof(DLGTEMPLATE);

    // 1 - Check for MENU and move past it
    SkipOrdinalOrTextField(&pDlgRes);

    // 2 - Check for CLASS NAME and move past it
    SkipOrdinalOrTextField(&pDlgRes);

    // 3 - Check for the CAPTION and move past it
    SkipOrdinalOrTextField(&pDlgRes);
```

Since ordinals start with **0xFFFF**, **SkipOrdinalOrTextField()** can check for this value to determine what method it needs to use to skip the field. If the field's an ordinal, it advances the pointer by **sizeof(WORD)** and if it's text, it uses a loop to advance the pointer by **sizeof(WCHAR)** until it passes a null.

Once the **MENU**, **CLASS NAME** and **CAPTION** fields have been skipped, there may be a point size and **FACE NAME** field to skip. These will only be present if the **DS_SETFONT** style has been set in the dialog template, so we check for this before skipping the fields.

```
    // The following will be present ONLY if "DS_SETFONT" style has been
    // specified for the dialog box.
    // 4 - Check if a unique font is specified for this dialog.
    // 5 - Check if a FACE NAME is specified.

    if (DS_SETFONT & DlgTemplate.style)
    {
       pDlgRes += sizeof(WORD);   // Move past the point size field

       // "szFontName" contains "DOUBLE-NULL" terminated string. Se we
       // need to advance the pointer TWO UNICODE characters. Since
       // "SkipOrdinalOrText()" only advances past ONE null char, then
       // we need to advance it past the second NULL char ourselves.

       SkipOrdinalOrTextField(&pDlgRes);
       pDlgRes += sizeof(WCHAR);   // Move past the NULL char
    }
```

Once we've skipped past all that lot, we're into the control information, so we can loop through, creating each one in turn and adding any strings that we may have set up:

```
    // Now create the window controls one at a time
    for (UINT i = 0; i < m_uNumOfCtls; i++)
    {
       CreateControl(&pDlgRes);
       AddStringList((LPCCTLINFO)m_ctlPtrList.GetTail());
    }
    return TRUE;
 }
```

CreateControl()

CreateControl() is where the control is created from the information in the dialog template. It is quite long and complex, so, once again, we'll look at it in sections.

This routine works in a way similar to **CreateDialog()**, in that the first thing it does is to copy the information from the dialog template data into a structure, in this case a **DLGITEMTEMPLATE**. Once it has copied the data, it moves the pointer so that it points to the next control in the dialog:

```
BOOL CDynamicDialog::CreateControl(LPBYTE* ppDlgRes)
{
   DLGITEMTEMPLATE DlgItemTemplate;
   DLGITEMTEMPLATE* pDlgItemTemplate = &DlgItemTemplate;
   memcpy(pDlgItemTemplate, *ppDlgRes, sizeof(DLGITEMTEMPLATE));
   *ppDlgRes += sizeof(DLGITEMTEMPLATE);

   if (0xFFFF == *(WORD*)*ppDlgRes)
   {
      *ppDlgRes += sizeof(WORD);
   }
```

The adjustment is needed because the controls are aligned on **DWORD** boundaries, and we need to make sure that we're in the right position for the next control to be read.

Once we've done that, we can look to see what sort of control we're dealing with, and save the class name away ready for when we create the control:

```
   BYTE byClassType;
   CString strClassName;
   switch(byClassType = (BYTE)**ppDlgRes)
   {
      case  CTLTYPE_LISTVIEW : strClassName = _T("SysListView32") ; break;
      case  CTLTYPE_BUTTON   : strClassName = _T("button")    ; break;
      case  CTLTYPE_EDIT     : strClassName = _T("edit")      ; break;
      case  CTLTYPE_STATIC   : strClassName = _T("static")    ; break;
      case  CTLTYPE_LISTBOX  : strClassName = _T("listbox")   ; break;
      case  CTLTYPE_SCROLLBAR: strClassName = _T("scrollbar"); break;
      case  CTLTYPE_COMBOBOX : strClassName = _T("combobox") ; break;
      default                : strClassName = (WCHAR*)*ppDlgRes;
         TRACE("[Line - %d] - [default: ==>strClassname = [%s],
               Class Type = [%x].....\n", __LINE__,
               (LPCSTR)strClassName, byClassType);
         byClassType = 0;
         ASSERT(FALSE);
   }
```

If we have a control that hasn't been accounted for, we'll hit the **default** statement, which will output some error information and **ASSERT** in debug builds.

We skip past the class ID and look for the window text, saving it away if we find it:

```
   *ppDlgRes += sizeof(WORD);          // Move past the class ID

   // Check if the next byte is an ordinal number. If it's not,
   // then it's the control window text. Get it!

   CString szCtlText;
   if (0xFFFF != **ppDlgRes)
   {
      szCtlText = (WCHAR*)*ppDlgRes;
   }
   // Move past the control window text or ordinal number
   SkipOrdinalOrTextField(ppDlgRes);

   // Move past the extra byte
   *ppDlgRes += sizeof(WORD);
```

Once again, we align the pointer correctly before using the dimension data to set the position and size of the control. We also add on any offset which was given when the dialog object was originally created:

```
   // Move the dialog resource data pointer to the
   // DWORD alignment.
```

```
WORD nExtraOffsetBytes;
nExtraOffsetBytes = (WORD)(4 - (((WORD)(DWORD)*ppDlgRes) & 3)) % 4;
*ppDlgRes += nExtraOffsetBytes;

// Set the dimensions of the control window. Since the control
// windows units are DLUs, we need to convert them to screen
// pixels using CDialog::MapDialogRectEx().

CRect rectWndCtl(pDlgItemTemplate->x, pDlgItemTemplate->y,
                 pDlgItemTemplate->x+pDlgItemTemplate->cx,
                 pDlgItemTemplate->y+pDlgItemTemplate->cy);
MapDialogRectEx(rectWndCtl);

// Now add the offset pixels to control window dimensions before
// creating it. Remember that m_nXOffset and m_nYOffset were set in
// Initialize()

rectWndCtl.OffsetRect(m_nXOffset, m_nYOffset);
```

When we're using 3D dialogs, we need to set a special flag, **WS_EX_CLIENTEDGE**, in the extended style for edit controls and list boxes if they are to display properly:

```
// Note that to be able to display the "edit" and "listbox" in
// 3D you must add WS_EX_CLIENTEDGE to extended window style of
// the control

DWORD dwExtendedStyle = pDlgItemTemplate->dwExtendedStyle;
if (strClassName == "edit" || strClassName == "listbox")
   dwExtendedStyle |= WS_EX_CLIENTEDGE;
```

Now we get to the important part—actually creating the control. We use the **CreateWindowEx()** API and pass in as parameters all the information that we have gathered from the resource data:

```
// Create the control that is to be displayed as part of the dialog
HWND hWndCtl = ::CreateWindowEx(dwExtendedStyle,
                                (LPCTSTR)strClassName,
                                szCtlText, pDlgItemTemplate->style,
                                rectWndCtl.left, rectWndCtl.top,
                                rectWndCtl.Width(),
                                rectWndCtl.Height(),
                                m_pWndParent->m_hWnd,
                                (HMENU)pDlgItemTemplate->id,
                                AfxGetResourceHandle(), NULL);

ASSERT(hWndCtl);
m_ctlPtrList.AddTail((void*) new CCtlInfo(hWndCtl,
                           byClassType, szCtlText));
```

The final task is to set the control's font to the same as that used in the dialog:

```
   // Retrieve the dialog's font and use it to set the control's font
   CFont* pDlgFont = (CFont*)m_pWndParent->GetFont();
   CWnd* pWnd = (CWnd*)CWnd::FromHandle(hWndCtl);
   pWnd->SetFont(pDlgFont);

   return TRUE;
}
```

How to Use Dynamic Dialogs

Now that we've covered the structure and operation of the class (and before getting on to the more complex example in the next section), let's give a simple example of how one of these dialogs might be used.

Follow these steps to create the framework of the sample application:

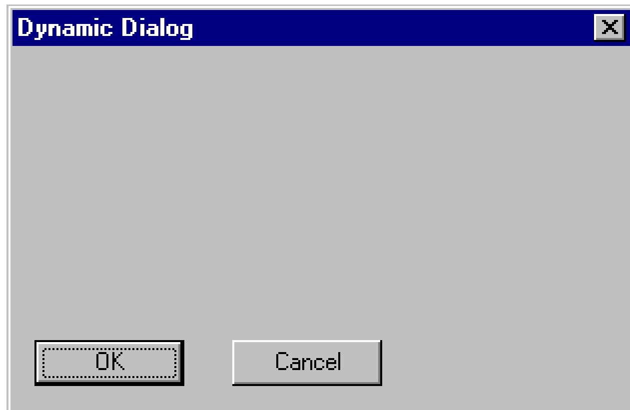
Use AppWizard to create an SDI program, with all the default options.

Create a dialog, using the resource editor, which has no controls other than the default buttons, and give it the ID **IDD_TESTDLG**.

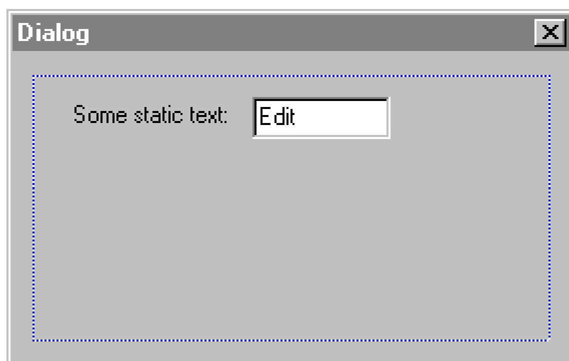
Use ClassWizard to add a class for the dialog called **CTestDlg**.

Add a menu item to display the dialog. You can add the handler to any class in your application you like.

If you compile and run the application at this stage, you'll get a blank dialog, like this:



We'll now use the dynamic dialog class to 'paste' some controls into this dialog at runtime. The first step is to use the resource editor to create the controls that we're going to include. Don't worry about the dialog caption or properties, because we aren't going to use them, but do remember to delete the OK and Cancel buttons, because we don't want them included twice. Give the dialog a suitable ID, such as **IDD_DYN1**. You can see from the picture below that the dialog contains two controls: a static text control and an edit control.



Now modify the **CTestDlg** class, adding a pointer to a **CDynamicDlg** and a destructor (which we'll use to delete the dynamic dialog object when we've finished):

```
class CTestDlg : public CDialog
{
// Construction
public:
```

```
    CTestDlg(CWnd* pParent = NULL);   // standard constructor
    ~CTestDlg();

// Dialog Data
    //{{AFX_DATA(CTestDlg)
    enum { IDD = IDD_TESTDLG };
        // NOTE: the ClassWizard will add data members here
    //}}AFX_DATA


// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CTestDlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:
    CDynamicDialog *pCD;

    // Generated message map functions
    //{{AFX_MSG(CTestDlg)
    virtual BOOL OnInitDialog();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

We need to initialize **pCD** in the constructor and arrange for its deletion in the destructor:

```
CTestDlg::CTestDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CTestDlg::IDD, pParent),
    pCD(NULL)
{
    //{{AFX_DATA_INIT(CTestDlg)
        // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
}

CTestDlg::~CTestDlg()
{
    delete pCD;
    pCD = NULL;
}
```

We want to place the controls into the dialog when it's created, so use ClassWizard to add a handler for **WM_INITDIALOG** to **CTestDlg**, and use this to create and show the dynamic dialog object, giving it the appropriate resource ID:

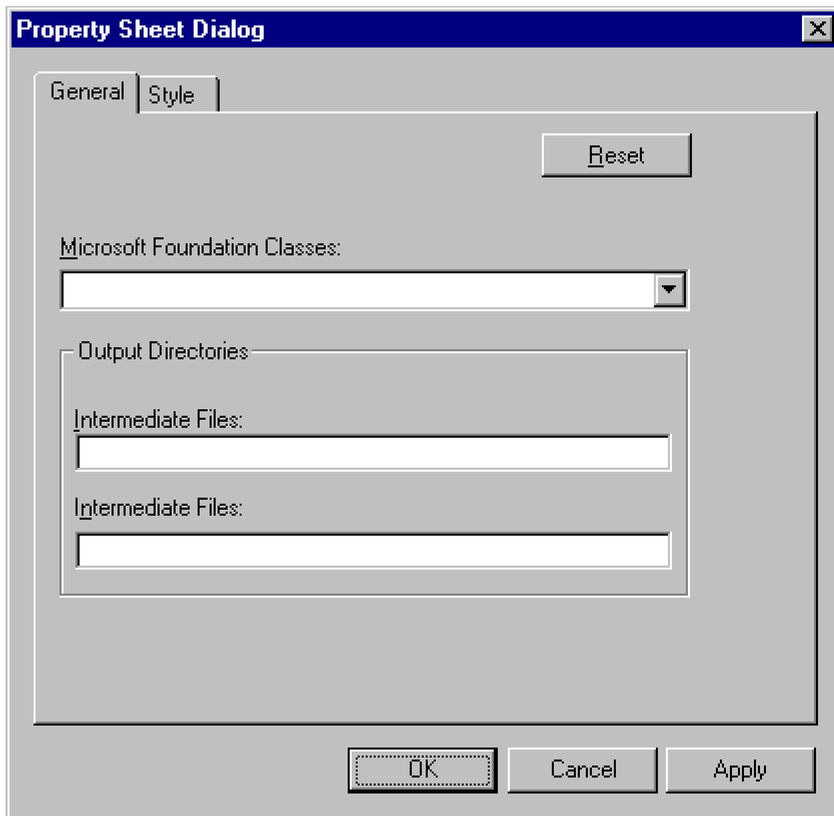
```
BOOL CTestDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // Add controls to the dialog
    pCD = new CDynamicDialog(this, IDD_DYN1);

    pCD->ShowDialog(TRUE);

    return TRUE;
}
```

That's it! When you build and run the program, you should see a composite dialog, built from **IDD_TESTDLG** and the controls in **IDD_DYN1**:

You can find the complete code on the CD in the `Dyn1` directory.

Constructing a Property Sheet from Scratch

Now let's turn our attention back to property sheets, to look at how they work in a different way. Instead of trying to disassemble the parts of a property sheet for you, we'll take a different approach, and show you how to assemble your own. This will give you a deeper insight into the workings of Microsoft's property sheets, and at the same time you'll see how the **`CDynamicDialog`** class can be used in a more substantial project.



In general, a property sheet (such as the one shown above) is a window that contains child control windows, such as tab and edit controls. To construct a property sheet from scratch, you need to perform the following steps:

Derive a class from **`CDialog`**, which will be the main window.

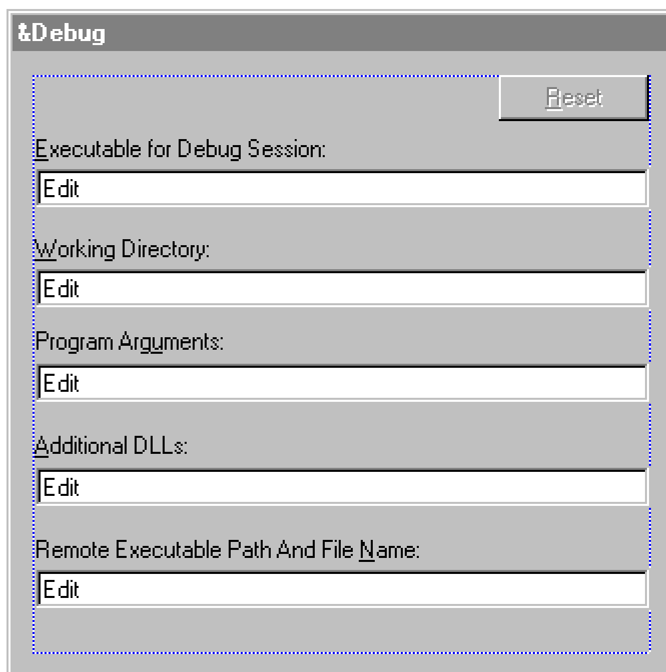
Create a new class derived from **`CTabCtrl`** and create an object of this class as a child of the main window.

Create one or more dialogs that will be displayed as the pages of the property sheet. The process is similar to creating dialogs for property pages, but in this instance we're going to display them dynamically at run time, with the help of the generic reusable class **`CDynamicDialog`**.

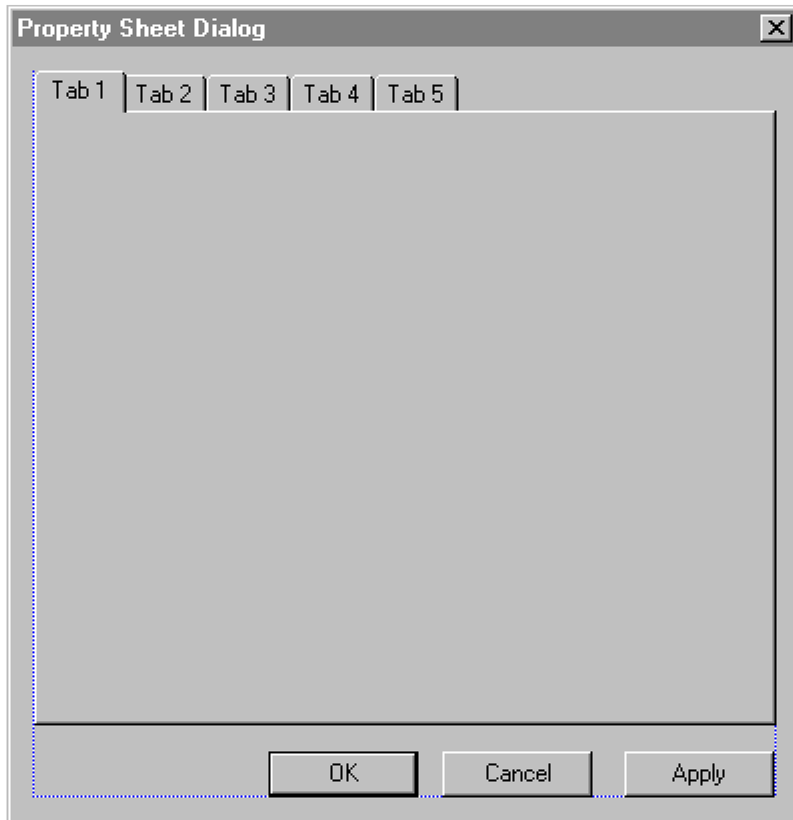
The rest of this section is going to show how to build a property sheet in this way, starting with creating the basic resource components.

Creating the Property Sheet Dialog and Subdialogs

The first task is to use the resource editor to create the **two** dialogs that will be displayed on the pages of the property sheet. Give them the IDs **`IDD_PAGE_GENERAL`** and **`IDD_PAGE_DEBUG`**. Note that these dialogs don't contain OK or Cancel buttons, as shown in the figure below:



Next, create the dialog box which will hold the property sheet and give it the ID **`IDD_PROPFROMSCRATCH`**. Add a button control to the property sheet dialog, with Apply as its caption, and an ID of **`IDC_PROPFROMSCRATCH_APPLY`**. Finally, add a tab control with an ID of **`IDC_PROPFROMSCRATCH_TABCTRL`**. You should end up with a dialog that looks like this. Note that the tab control has been made large enough to contain the dialogs which will be displayed on it at run time.



Creating the Property Sheet Class

Using ClassWizard, create a class called **CPropFromScratch** to represent the dialog, making sure that it's derived from **CDialog**. Provide a **WM_INITDIALOG** message handler for the class, **CPropFromScratch::OnInitDialog()** and add a **protected** member variable **CTabCtrlExtended m_TabCtrl** to represent the child tab control. We're going to use control subclassing and our own customized tab control class to handle the messages for the tab control.

```
class CPropFromScratch : public CDialog
{
// Construction
public:
   CPropFromScratch(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
   //{{AFX_DATA(CPropFromScratch)
   enum { IDD = IDD_PROPFROMSCRATCH };
   //}}AFX_DATA


// Overrides
   // ClassWizard generated virtual function overrides
   //{{AFX_VIRTUAL(CPropFromScratch)
   protected:
   virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
   //}}AFX_VIRTUAL

// Attributes
public:
   void  InitChildTabControl();
```

```
// Implementation
protected:
   CTabCtrlExtended m_TabCtrl;

   // Generated message map functions
   //{{AFX_MSG(CPropFromScratch)
   virtual BOOL OnInitDialog();
   //}}AFX_MSG
   DECLARE_MESSAGE_MAP()
};
```

We now need to modify **`OnInitDialog()`** and add **`InitChildTabControl()`**, to set up the tab control correctly.

```
BOOL CPropFromScratch::OnInitDialog()
{
   CDialog::OnInitDialog();

   // Use "SubclassDlgItem()" to subclass the tab control
   VERIFY(m_TabCtrl.SubclassDlgItem(IDC_PROPFROMSCRATCH_TABCTRL, this));

   // Initialize the tab control
   InitChildTabControl();

   // Add the sub-dialogs
   m_TabCtrl.AddDialog(0, IDD_PAGE_GENERAL);
   m_TabCtrl.AddDialog(1, IDD_PAGE_DEBUG);
   m_TabCtrl.ShowNewDialog(0);

   return TRUE;
}

void CPropFromScratch::InitChildTabControl()
{
   // Set the child tab control font. Note that you should
   // not call 'GetFont()' in the 'OnCreate()' routine.
   m_TabCtrl.SetFont(GetFont());

   // Now, set the tabs labels
   TC_ITEM tcItem;
   tcItem.mask = TCIF_TEXT;
   tcItem.pszText = _T("General");
   m_TabCtrl.InsertItem(0, &tcItem);
   tcItem.pszText = _T("Style");
   m_TabCtrl.InsertItem(1, &tcItem);
   m_TabCtrl.SetCurSel(0);
}
```

Deriving a Class from CTabCtrl

The next stage involves deriving a class from **`CTabCtrl`**, which will handle the dynamic display of dialogs on the property sheet pages.

First, using ClassWizard, derive a class from **`CTabCtrl`** and call it **`CTabCtrlExtended`**.
This class is going to be responsible for dynamically loading the dialogs when the user clicks on a tab, so it has to receive the notification message which would normally be sent to the parent window. This is done using a technique called **message reflection**; in ClassWizard, messages which can be reflected back to the control are marked with an equals sign (=). In this case, the message we want to reflect is **`TCN_SELCHANGE`**, so use ClassWizard to set up a handler for =TCN_SELCHANGE, and call it **`OnSelchange()`**.

The dynamic subdialogs will need to be stored in memory, so add two new **protected** member variables to the class, one to hold the list of dialog pointers and another to hold a pointer to the currently displayed dynamic dialog:

```
CMapWordToOb m_DlgsPtrList;
CDynamicDialog* m_pCurDynamicDialog;
```

Of course, these members will need to be initialized in the constructor and cleaned up in the destructor as shown:

```
CTabCtrlExtended::CTabCtrlExtended()
{
   m_pCurDynamicDialog = NULL;
}

CTabCtrlExtended::~CTabCtrlExtended()
{
   // Clean up dynamic dialog memory
   WORD wKey;
   CDynamicDialog* pDynamicDialog;

   POSITION pos = m_DlgsPtrList.GetStartPosition();
   while (pos)
   {
      m_DlgsPtrList.GetNextAssoc(pos, wKey, (CObject*&)pDynamicDialog);
      delete pDynamicDialog;
   }
   m_DlgsPtrList.RemoveAll();
}
```

The class also needs three helper functions, which will be used to manage the dynamic dialogs

```
AddDialog(),
ShowNewDialog()
ShowDialog()
```

AddDialog() adds a subdialog to the list of dialogs. This routine allocates memory for the new **CDynamicDialog** object and inserts it into the map table. **ShowNewDialog()** replaces the currently displayed dialog with a new dialog. **ShowDialog()** displays a dialog in the window. This function doesn't replace a subdialog that is currently displayed on the screen.

Here's what the header file should look like after it has been modified:

```
class CTabCtrlExtended : public CTabCtrl
{
// Construction
public:
   CTabCtrlExtended();

// Attributes
public:

// Operations
public:
   virtual void AddDialog(WORD wKey, UINT uDlgResourceID,
         UINT uPlacementWnd, LPCCLISTCTLTEXT arListCtlText = NULL);
   virtual void AddDialog(WORD wKey, UINT uDlgResourceID,
         LPCCLISTCTLTEXT arListCtlText = NULL);
   virtual void ShowNewDialog(WORD wKey);
   virtual void ShowDialog(WORD wKey, BOOL bShowDialog = TRUE);

// Overrides
   // ClassWizard generated virtual function overrides
   //{{AFX_VIRTUAL(CTabCtrlExtended)
```

```
   public:
   //}}AFX_VIRTUAL

// Implementation
public:
   virtual ~CTabCtrlExtended();
   CFont    m_Font;

protected:
   CMapWordToOb      m_DlgsPtrList;
   CDynamicDialog*   m_pCurDynamicDialog;

   // Generated message map functions
protected:
   //{{AFX_MSG(CTabCtrlExtended)
   afx_msg void OnSelchange(NMHDR* pNMHDR, LRESULT* pResult);
   //}}AFX_MSG

   DECLARE_MESSAGE_MAP()
};
```

Let's look at the most important member functions in more detail.

OnSelchange()

When the user clicks on one of the tabs of the tab control, it results in the tab selection being changed, in the same way that the selection in a list box changes. As we noted above, this results in a **`TCN_SELCHANGE`** message which will be reflected to our handler, **`OnSelchange()`**:

```
void CTabCtrlExtended::OnSelchange(NMHDR* pNMHDR, LRESULT* pResult)
{
   // Make sure that the user has added dynamic dialogs
   if (!m_DlgsPtrList.IsEmpty())
   {
      int nCurSel = GetCurSel();
      if (nCurSel < m_DlgsPtrList.GetCount())
         ShowNewDialog(nCurSel);
   }

   // No result code to return
   *pResult = 0;
}
```

The function checks whether the tab control has any dynamic dialogs defined, and if so, it calls **`ShowNewDialog()`** to display the one corresponding to the selected tab.

AddDialog()

`AddDialog()` is used to add a dynamic dialog object to the list maintained by the **`CTabCtrlExtended`** object. The tab class uses an MFC map to store its dynamic dialog objects, the key of the map entry corresponding to the index of the tab to which the dialog applies.

There are two slightly different flavors of the routine. One sizes the dialog to fit within the parent window, whereas the other takes a placement window which defines the dialog's position:

```
void CTabCtrlExtended::AddDialog(WORD wKey, UINT uDlgResourceID,
               UINT uPlacementWnd, LPCCLISTCTLTEXT arListCtlText)
{
   CDynamicDialog* pDynamicDialog = new CDynamicDialog(
                                       this,uDlgResourceID,
                                       uPlacementWnd, arListCtlText);
```

```
   m_DlgsPtrList.SetAt(wKey, pDynamicDialog);
}


void CTabCtrlExtended::AddDialog(WORD wKey, UINT uDlgResourceID,
               LPCCLISTCTLTEXT arListCtlText)
{
   CRect rectTabCtrl;
   GetWindowRect(rectTabCtrl);
   ScreenToClient(rectTabCtrl);

   CDynamicDialog* pDynamicDialog = new CDynamicDialog(
                        this, uDlgResourceID, rectTabCtrl.left + 7,
                        rectTabCtrl.top + 23, arListCtlText);
   m_DlgsPtrList.SetAt(wKey, pDynamicDialog);
}
```

ShowNewDialog()

This function displays a dynamic dialog on the screen, but only if it isn't already being displayed.

```
void CTabCtrlExtended::ShowNewDialog(WORD wKey)
{
   CDynamicDialog* pDynamicDialog = NULL;
   m_DlgsPtrList.Lookup(wKey, (CObject*&)pDynamicDialog);

   // Show the dialog only if it is different to the currently
   // displayed dialog...
   if (m_pCurDynamicDialog != pDynamicDialog)
   {
      // Remove the old dialog from the screen
      if (m_pCurDynamicDialog)
      {
         m_pCurDynamicDialog->ShowDialog(FALSE);
         InvalidateRect(NULL, TRUE);
         UpdateWindow();
      }
      pDynamicDialog->ShowDialog(TRUE);
      m_pCurDynamicDialog = pDynamicDialog;
   }
}
```

The function looks up the dialog in the map and checks it against the current one. If they are different, the old dialog is hidden and the new one displayed in its place.

ShowDialog()

`ShowDialog` is similar to **`ShowNewDialog()`**, but in this case the dialog being displayed doesn't replace the one on the screen, but adds to it. This gives the possibility of displaying more than one dialog at a time.

```
void CTabCtrlExtended::ShowDialog(WORD wKey, BOOL bShowDialog /*=TRUE*/)
{
   CDynamicDialog* pDynamicDialog = (CDynamicDialog*)NULL;
   m_DlgsPtrList.Lookup(wKey, (CObject*&)pDynamicDialog);

   // Make sure that it's not already displayed on the screen
   if (pDynamicDialog->IsDialogVisible() && bShowDialog)
      return;

   pDynamicDialog->ShowDialog(bShowDialog);
}
```

Making it Work

Now we've got all the routines in place, it is quite simple to set up the tab control to use the dynamic dialog. In fact, we've already seen the code in **`CPropFromScratch::OnInitDialog()`**:

```
// Add the subdialogs
m_TabCtrl.AddDialog(0, IDD_PAGE_GENERAL);
m_TabCtrl.AddDialog(1, IDD_PAGE_DEBUG);
m_TabCtrl.ShowNewDialog(0);
```

When the property sheet dialog is being set up, we add two dialogs, giving them the keys **`0`** and **`1`**, which correspond to tabs 0 and 1 on the tab control, and set the tab control to show dialog **`0`** initially. After this, all user interaction with the tabs will be handled by the reflected notification message. The only thing left is to declare an object of the new class and call its **`DoModal()`** function to display it. You can find the completed project on the CD in the **`FromScratch`** directory. It's all pretty simple really!

Advanced Property Sheets

This section will introduce a reusable property sheet base class, called **CPropSheetExtended**, derived from **CPropertySheet**, that will allow you to (amongst other things) do the following:

- Resize the property sheet and pages.
- Resize the tab control
- Move the tab control.
- Move the property pages
- Move the property sheet's standard buttons (OK, Cancel, and Apply).
- Hide the property sheet's standard buttons
- Display the standard buttons in a modeless property sheet
- Specify the default button.
- Change the caption of the tab control.
- Change the font of the tab control.
- Add images to the tab control
- Use accelerator keys to navigate between property pages.

We'll describe how these capabilities can be implemented in the following sections.

Resizing the Property Sheet

You can resize the property sheet at any time, using **CPropSheetExtended::InflateSize()** and **CPropSheetExtended::DeflateSize()**. These are pretty simple functions that make use of **CWnd::MoveWindow()**. It's as simple as this:

```
void CPropSheetExtended::InflateSize(int x, int y, const long lMoveFlag /*=MOVEXY_NONE*/)
{
   CRect rectProp;
   GetWindowRect(rectProp);
   ScreenToClient(rectProp);
   rectProp.InflateRect(x,y);
   MoveWindow(rectProp);
}
```

The **InflateSize()** function retrieves the current dimensions of the property sheet, uses **CRect::InflateRect()** to change the width and height of the **rect**, then **CWnd::MoveWindow()** to resize the property sheet.

Please note that if you use the above routine in **CPropertySheetExtended::OnInitDialog()**, you should only use it *after* the base class **OnInitDialog()**. Using **MoveWindow()** or **SetWindowPos()** before the base class's **OnInitDialog()** will *not* work, because **CWnd::GetWindowRect()** and **CWnd::GetClientRect()** don't return the correct dimensions until the base class **OnInitDialog()** is called.

Resizing the Tab Control

We saw in the previous section that the property sheet is actually a window which houses the tab control. So, to resize the tab control, all you need to do is retrieve a pointer to it and change its size, just as we did for the property sheet window itself. It's easy to get a pointer to the tab control using **CPropertySheet::GetTabControl()**.

```
void CPropSheetExtended::InflateTabControl(int x, int y)
```

```
{
   // Resize the Tab control
   CRect rectTabCtrl;
   CTabCtrl* pTabCtrl = GetTabControl();
   pTabCtrl->GetWindowRect(rectTabCtrl);
   ScreenToClient(rectTabCtrl);
   rectTabCtrl.InflateRect(x,y);
   pTabCtrl->MoveWindow(rectTabCtrl);
}
```

Note that the above code only changes the width and height of the tab control; it doesn't alter the coordinates of the top-left of the control. If you want to move the tab control, you'll need to use the technique presented in the next section.

Moving the Tab Control

Once we've gained a pointer to the control, moving the tab control to a new location inside the property sheet is simple:

```
void CPropSheetExtended::OffsetTabControl(int x, int y)
{
   // Move the Tab control
   CRect rectTabCtrl;
   CTabCtrl* pTabCtrl = GetTabControl();
   pTabCtrl->GetWindowRect(rectTabCtrl);
   ScreenToClient(rectTabCtrl);
   rectTabCtrl.OffsetRect(x,y);
   pTabCtrl->MoveWindow(rectTabCtrl);
}
```

Note that the last two techniques affect *only* the tab control. They don't affect the property pages, the buttons or the property sheet window itself. You'll see how you can affect each of those elements in the next few sections.

Moving and Resizing the Property Pages

Moving the property pages is not just a matter of using the **`CWnd::MoveWindow()`** routine, because the **`CPropertySheet`** class always saves the coordinates of the encapsulated property pages when they are first created. Since moving or resizing the property pages yourself will *not* update the coordinates stored in **`CPropertySheet`**, you'll need to resize the pages every time **`CPropertySheet`** tries to resize them using its internal coordinates. There are two occasions that this happens: when a new tab is selected and when the user presses the Apply button.

To get around this, store the new size of the property pages in a **`CRect`** data member and use this information to reposition the pages when necessary. You should intercept the **`TCN_SELCHANGE`** notification and the **`ID_APPLYNOW`** command and call a function in your class to resize the property pages. You need to do this after the base class implementation has been called, so, in our implementation, we post a user-defined message to ourselves whenever the pages need to be resized. This is handled by a function called **`OnResizePropPage()`**.

Here you can see the necessary code:

```
BEGIN_MESSAGE_MAP(CPropSheetExtended, CPropertySheet)
   //{{AFX_MSG_MAP(CPropSheetExtended)
   ON_WM_CREATE()
   ON_BN_CLICKED(IDOK, OnOK)
```

```
   ON_BN_CLICKED(IDCANCEL, OnCancel)
   //}}AFX_MSG_MAP

   ON_COMMAND(ID_APPLY_NOW, OnApplyNow)
   ON_MESSAGE(WM_USER_RESIZEPAGE, OnResizePropPage)
END_MESSAGE_MAP()

// You need to override this virtual function to intercept the
// TCN_SELCHANGE.
BOOL CPropSheetExtended::OnNotify(WPARAM wParam, LPARAM lParam, LRESULT* pResult)
{
   NMHDR* pNMHDR = (NMHDR*) lParam;

   if (TCN_SELCHANGE == pNMHDR->code)
   {
      PostMessage(WM_USER_RESIZEPAGE);
   }
   return CPropertySheet::OnNotify(wParam, lParam, pResult);
}

// ID_APPLY_NOW message handler
void CPropSheetExtended::OnApplyNow()
{
   PostMessage(WM_USER_RESIZEPAGE);
}

LONG CPropSheetExtended::OnResizePropPage(WPARAM, LPARAM)
{
   // Now, resize the property pages
   CPropertyPage* pPage = GetActivePage();
   ASSERT(pPage);
   pPage->MoveWindow(m_rectPage);
   return 0L;
}

void CPropSheetExtended::OffsetPropPage(int x, int y)
{
   // Now, offset the property pages
   CPropertyPage* pPage = GetActivePage();
   pPage->GetWindowRect(m_rectPage);
   ScreenToClient(m_rectPage);
   m_rectPage.OffsetRect(x,y);
   pPage->MoveWindow(m_rectPage);
}
```

Moving the Standard Buttons

Moving a window requires a pointer to the window to be moved, and the new coordinates to be used in **`CWnd::SetWindowPos()`** or **`CWnd::MoveWindow()`**, so all we need to do to move the standard buttons is to retrieve the window pointers to them and apply the move functions. The following code shows how to do it:

```
void CPropSheetExtended::MoveChildWindow(UINT nWndID, int x, int y)
{
   // Retrieve the window pointer and move it
   CRect rectBtn;
   CWnd* pWnd = GetDlgItem(nWndID);
   pWnd->GetWindowRect(rectBtn);
   ScreenToClient(rectBtn);
   rectBtn.OffsetRect(x,y);
   pWnd->MoveWindow(rectBtn);
   pWnd->Invalidate();
}

void CPropSheetExtended::OffsetStandardButtons(int x, int y)
{
   // Now Move the OK, CANCEL, and APPLY buttons
   MoveChildWindow(IDOK, x, y);
```

```
      MoveChildWindow(IDCANCEL, x, y);
      MoveChildWindow(ID_APPLY_NOW, x, y);
   }
```

Hiding the Standard Buttons

There are two techniques used to hide the standard buttons. One involves retrieving pointers to the button windows and using **CWnd::ShowWindow()**. The other is specific to the Apply button (whose ID is **ID_APPLY_NOW**).

Method 1

```
   void CPropSheetExtended::ShowCancel(BOOL bShow /*=TRUE*/)
   {
      CWnd* pWnd = GetDlgItem(IDCANCEL);
      pWnd->ShowWindow(bShow);
      pWnd->EnableWindow(bShow);
   }

   void CPropSheetExtended::ShowApply(BOOL bShow /*=TRUE*/)
   {
      CWnd* pWnd = GetDlgItem(ID_APPLY_NOW);
      pWnd->ShowWindow(bShow);
      pWnd->EnableWindow(bShow);
   }

   void CPropSheetExtended::ShowOk(BOOL bShow /*=TRUE*/)
   {
      CWnd* pWnd = GetDlgItem(IDOK);
      pWnd->ShowWindow(bShow);
      pWnd->EnableWindow(bShow);
   }
```

Method 2: Applies to ID_APPLY_NOW Button

Starting with Visual C++ 4.0, **CPropertySheet** now includes a **PROPSHEETHEADER** data member called **m_psh** which allows you to specify numerous things about the way a property sheet should be created. The **PROPSHEETHEADER** structure has a **dwFlags** member which can take a range of flags, including **PSH_NOAPPLYNOW**, which specifies that the property sheet should be created without an Apply button. So, you can specify this flag before displaying the property sheet on the screen like this:

```
   CPropertySheetDerived MyPropSheet(_T("Property Sheet"));
   MyPropSheet.m_psh.dwFlags |= PSH_NOAPPLYNOW;
   MyPropSheet.DoModal();
```

Displaying the Standard Buttons Inside a Modeless Property Sheet

To display the standard buttons inside a modeless property sheet, you need to understand something of the property sheet source code, because MFC, by default, displays the standard buttons only when it displays a modal property sheet.

Don't worry, though, because we've already explored the source code for you! What you need to do is to fool MFC when it creates a modeless property sheet so it thinks it's creating a modal sheet. You do this at a specific location in the code, before the base class **OnInitDialog()** is called. In the following code, the **CPropertySheet** data member **m_bModeless** is set to **FALSE** before the base class **OnInitDialog()** is

called, and reset to its original state afterwards. This sleight of hand doesn't produce any other side effects.

```
BOOL CPropSheetExtended::OnInitDialog()
{
   // Save the object state
   BOOL bModeless = m_bModeless;

   // We want to fool MFC if the sheet is modeless and we want to
   // display the standard buttons. OnCreate() will display the
   // standard buttons if the dialog is MODAL. So let him do it.

   if (bModeless && m_bDisplayStandardButtons)
   {
      m_bModeless = FALSE;
   }

   // Call the base class routine
   BOOL bRtnValue = CPropertySheet::OnInitDialog();

   // Restore the flag to its original state
   m_bModeless = bModeless;


   // WARNING - The following two statements MUST follow the call to
   //           the base class 'CPropertySheet::OnInitDialog()'
   //
   // Initialize the 'm_rectPage' member variable. This will make sure
   // that 'OnNotify()' TC_SELCHANGE behaves properly when the user
   // selects a tab from property sheet before making any modifications
   // to it.

   GetActivePage()->GetWindowRect(m_rectPage);
   ScreenToClient(m_rectPage);

   return bRtnValue;
}
```

Specifying the Property Sheet Default Button

The default push button that is enabled when property sheet is displayed is the OK button. To change this default setting, you would need to use the following member function that is provided by **`CPropSheetExtended`**:

```
void CPropSheetExtended::SetDefaultButton(UINT nID)
{
   SendMessage(DM_SETDEFID, nID);
}
```

For example, to make the Apply button the default push button, you would include a statement similar to that below after the base class **`CPropertySheet::OnInitDialog()`** has been executed:

```
m_pMyPropSheetExtened->SetDefaultButton(ID_APPLY_NOW);
```

Changing the Tab Control Caption

To change the tab control caption text, you would do the following:

```
void CPropSheetExtended::SetTabCaption(CString stCaption,
                  int nTabIndex /*=0*/)
```

```
{
   // Retrive a pointer to the tab control
   CTabCtrl* pTabCtrl = GetTabControl();

   // Now set the specified tab text
   TC_ITEM tcItem;
   tcItem.mask = TCIF_TEXT;
   tcItem.pszText = stCaption.GetBuffer(stCaption.GetLength());
   stCaption.ReleaseBuffer();
   pTabCtrl->SetItem(nTabIndex, &tcItem);
}
```

Changing the Tab Control Font

Changing the tab control font requires you to retrieve the pointer to the child tab control, change its font and resize it.

```
void CPropSheetExtended::ChangeTabControlFont(CFont* pFont)
{
   CRect rectTabCtrl;
   CTabCtrl* pTabCtrl = GetTabControl();

   // 1 - Get a pointer to the tab control device context
   // 2 - Save the old font already existing in the DC
   // 3 - Get the text metrics for the old font
   // 4 - Select the new font into the device context
   // 5 - Get the text metrics for the new font
   // 6 - restore the DC to its original context
   // 7 - Calculate the multiplication ratio
   // 8 - Adjust the size of the tab
   // 9 - Resize the tab

   // 1 - Get a pointer to the tab control device context
   CDC* pDC = pTabCtrl->GetDC();

   // 2 - Save the old font already existing in the DC
   CFont* pOldFont = pDC->SelectObject(pTabCtrl->GetFont());

   // 3 - Get the text metrics for the old font
   TEXTMETRIC tmOldFont;
   pDC->GetTextMetrics(&tmOldFont);

   // 4 - Select the new font into the device context
   TEXTMETRIC tmNewFont;
   pDC->SelectObject(pFont);

   // 5 - Get the text metrics for the new font
   pDC->GetTextMetrics(&tmNewFont);

   // 6 - restore the DC to its original context
   pDC->SelectObject(pOldFont);

   pTabCtrl->SetFont(pFont);

   // Get the rect of the tab control
   pTabCtrl->GetWindowRect(rectTabCtrl);
   ScreenToClient(rectTabCtrl);

   // 7 - Calculate the multiplication ratio
   long lOldHeight = tmOldFont.tmHeight + tmOldFont.tmExternalLeading;
   long lNewHeight = tmNewFont.tmHeight + tmNewFont.tmExternalLeading;

   // 8 - Adjust the size of the rect
   rectTabCtrl.left = rectTabCtrl.left *
                  tmNewFont.tmAveCharWidth / tmOldFont.tmAveCharWidth;
```

```
    rectTabCtrl.top = rectTabCtrl.top * lNewHeight / lOldHeight;
    rectTabCtrl.right = rectTabCtrl.right *
                        tmNewFont.tmAveCharWidth / tmOldFont.tmAveCharWidth;
    rectTabCtrl.bottom = rectTabCtrl.bottom * lNewHeight / lOldHeight;

    // 9 - Resize the tab
    pTabCtrl->MoveWindow(rectTabCtrl);
}
```

The following code shows how it can be used:

```
void CPropSheetView::OnPSTCChangeFont()
{
    // Create a new font
    CFont font;
    font.CreateFont(-11,0,0,0,400,FALSE,FALSE,0, ANSI_CHARSET,
                OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                DEFAULT_PITCH | FF_SWISS, "Arial");

    // Change the tab control's font
    m_pPSExample2->ChangeTabControlFont(&font);

    // Change the position and button positions on the page
    m_pPSExample2->OffsetPropPage(2,7);
    m_pPSExample2->OffsetStandardButtons(0, 25);
}
```

Adding Images to the Tab Control

Adding bitmap images to the tab buttons is a straightforward process:

Create a new class derived from **CImageList**.

Retrieve the property sheet's tab control and set its image list.

Declare a **TC_ITEM** object on the stack and set the **mask** data member equal to **TCIF_IMAGE**.

Iterate through the property sheet pages specifying the index of the bitmap for each tab.

This is the code that will perform these actions:

```
void CPropSheetExtended::SetImageList(UINT nBitmapID, int cx,
                                      int nGrow, COLORREF crMask)
{
    // Make sure that there are property pages defined
    m_pImageList = new CImageList();

    // Create a blank image list (note that you can adjust the size of the
    // bitmap to whatever you want (e.g. 32*32, 16*16, etc...)
    m_pImageList->Create(nBitmapID, cx, nGrow, crMask);

    // Retrieve a pointer to the tab control
    CTabCtrl* pTabCtrl = GetTabControl();
    pTabCtrl->SetImageList(m_pImageList);

    // Now set the image list for each tab in the property sheet
    TC_ITEM tcItem;
    tcItem.mask = TCIF_IMAGE;
    for (int i = 0; i < GetPageCount(); i++)
    {
        tcItem.iImage = i;
        pTabCtrl->SetItem(i, &tcItem);
    }
}
```

Using Accelerator Keys for Property Pages

The process is really quite simple under Visual C++ 4.0, and we'll look at two methods: firstly, using a hard coded technique to intercept certain characters and activate the corresponding page, and secondly, using a better generic approach that is used inside the **`CPropSheetExtended`** class.

Method 1

All you need to do is override the property sheet's **`PreTranslateMessage()`** function. This function is a member of **`CWnd`**, which allows access to messages before they are fed into MFC's messaging system; this is the place we need to tap into if we are to intercept the accelerator keystrokes in the raw message data.

Inside the function, you need to check if the message generated is **`WM_SYSKEYDOWN`** (assuming that the user will be using *Alt+<key>*). The reason for using **`WM_SYSKEYDOWN`** is that *Alt* (and *F10*) are considered system keys by Windows, and to receive messages generated by pressing and releasing these keys, you need to process **`WM_SYSKEYDOWN`** and **`WM_SYSKEYUP`**, rather than the more normal **`WM_KEYDOWN`** and **`WM_KEYUP`**. Of course, *Alt* acts as a modifier for other keys, so the **`wParam`** will contain the code for the modified key.

The following example assumes that you have a property sheet with three property pages with the captions General, Debug, and C/C++, respectively, so that they can be selected using the accelerator keys *Alt-G*, *Alt-D* and *Alt-C*:

```
BOOL CPropSheetGeneral::PreTranslateMessage(MSG* pMsg)
{

   if (WM_SYSKEYDOWN == pMsg->message)
   {
      switch(pMsg->wParam)
      {
         case 'G':
            SetActivePage(0);
            break;
         case 'D':
            SetActivePage(1);
            break;
         case 'C':
            SetActivePage(2);
            break;
         default:
            return CPropertySheet::PreTranslateMessage(pMsg);
      }
   }
   return TRUE;
}
```

Note that the above techniques won't work with Visual C++ versions prior to 4.0, because previous versions of MFC didn't use the Windows 95 tab controls. Instead, the tab control in previous versions of property sheets used a different technique, which involved drawing the tab controls manually. Since they used the `TextOut()` API call to display the tab text, accelerator keys were not supported.

Method 2

This method, which uses an accelerator table, is rather more useful than the previous one. The following

steps show how it can be done:
1 Define a new accelerator table using the resource editor.
2 Derive a class from **CPropertySheet**, for example **CPropSheetExtended**, and add a data member:

```
HACCEL m_hAccel;
```

3 Add a new member function to the class which loads the accelerator table.

```
void CPropSheetExtended::LoadAcceleratorTable(UINT nAccelTableID /*=0*/)
{
   if (nAccelTableID)
   {
      m_hAccel = ::LoadAccelerators(AfxGetInstanceHandle(),
          MAKEINTRESOURCE(nAccelTableID));
   }
}
```

4.Override **CPropertySheet::PreTranslateMessage()** and modify it as follows:

```
BOOL CPropSheetExtended::PreTranslateMessage(MSG* pMsg)
{
   // Check to see if the property sheet has an accelerator table
   // attached to it. If there is one, call it. Return TRUE if it has
   // been processed. Otherwise, pass it to the base class function.

   if (m_hAccel && ::TranslateAccelerator(m_hWnd, m_hAccel, pMsg))
      return TRUE;

   return CPropertySheet::PreTranslateMessage(pMsg);
}
```

Embedding Property Sheets

Now that we've seen how to extend the standard property sheet class, let's turn our attention to embedding the property sheet in various types of window. In the next few sections, you'll see how to embed a property sheet in the following windows:

- **CView**s
- **CMiniFrameWnd**s
- Dialogs
- Splitter windows

Embedding a Property Sheet in a CFormView Window

Displaying a property sheet inside a class that is derived from **CView** is no different from displaying a button. You override the **OnCreate()** function to create the property sheet, adjust the view's dimensions accordingly, then override **OnInitialUpdate()** to resize the main frame to fit the property sheet. Here's how it can be done:

```
int CPropSheetView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
   if (CFormView::OnCreate(lpCreateStruct) == -1)
      return -1;

   // Create the modeless property sheet. You should add the
   // property pages before creating the property sheet
   m_pPSExample1View->AddPage(&m_ViewGeneralPage);
   m_pPSExample1View->AddPage(&m_ViewDebugPage);
   m_pPSExample1View->AddPage(&m_ViewCplusPage);
   if (!m_pPSExample1View->Create(this, WS_CHILD | WS_VISIBLE, 0))
      return -1;

   m_pPSExample1View->SetImageList(IDB_TVBMPIMG, 16, 1, RGB(255,0,0));

   CRect rectSheet, rectWindow;
   m_pPSExample1View->GetWindowRect(rectSheet);
   rectWindow = rectSheet;
   CalcWindowRect(rectWindow);

   // Adjust the positions of the frame and property sheet
   SetWindowPos(NULL, rectWindow.left,rectWindow.top,
                  rectWindow.Width() + 10, rectWindow.Height(),
                  SWP_NOZORDER | SWP_NOACTIVATE);

   m_pPSExample1View->SetWindowPos(NULL, 0, 0 ,rectSheet.Width(),
             rectSheet.Height(), SWP_NOZORDER | SWP_NOACTIVATE);

   return 0;
}

void CPropSheetView::OnInitialUpdate()
{
   CFormView::OnInitialUpdate();
   GetParentFrame()->RecalcLayout();
   ResizeParentToFit(FALSE);
   ResizeParentToFit(TRUE);
}
```

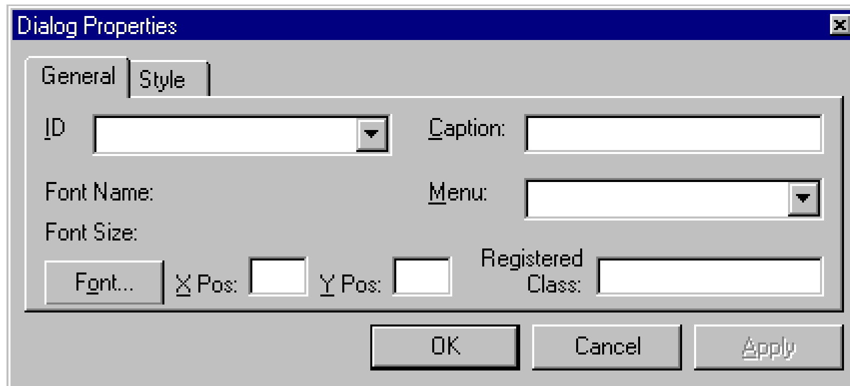
Note that you may add more than one property sheet to the view. All you then need to do is to create it

in the same way as the first one, and adjust the coordinates of the window accordingly.

Embedding a Property Sheet Inside a CMiniFrameWnd

This section will show you how to embed a property sheet inside a **CMiniFrameWnd**. In case you aren't familiar with this class, these are the half-height frame windows seen around floating toolbars. They behave just like normal frame windows, except that they don't have maximize and minimize buttons, and can be dismissed by a single click on the system menu.

The procedure consists of deriving a class, **CPSMiniFrameWnd**, from **CMiniFrameWnd** and creating the property sheet as a child window of the derived class. Does this sound familiar?



Creating the CPSMiniFrameWnd Class

Here's how to create the **CPSMiniFrameWnd** class. First, using ClassWizard, derive a class from **CMiniFrameWnd**; for example **CPSMiniFrameWnd**.

Then provide message handlers for **WM_ACTIVATE**, **WM_CREATE**, and **WM_SETFOCUS**. Windows sends **WM_ACTIVATE** to both the window that is being *deactivated* (first) and then to the window that is being *activated*. This message is sent only to top-level windows. **WM_CREATE** is sent to the window after it has already been created and before it has been made visible. You should limit your **OnCreate()** functionality to initialization issues (e.g., changing the size). Don't send messages from this function. **WM_SETFOCUS** is sent to the window after is has been activated (i.e. when it gains the keyboard and mouse focus).

You can override the base class **OnCreate()** function to create the property sheet as a child of the mini-frame window and add the member function **SetPropSheet()**. Also, add the member pointer object **m_pEmbeddedPropSheet** to the class declaration.

If you want to hide the standard buttons call the **CPropSheetExtended::SetDisplayStdButtons()** function, otherwise, they will be displayed automatically.

Here you can see the code for the `PSMiniFrameWnd.h` header file:

```
class CPSMiniFrameWnd : public CMiniFrameWnd
{
```

```
    DECLARE_DYNCREATE(CPSMiniFrameWnd)
public:
    // protected constructor used by dynamic creation
    CPSMiniFrameWnd(CPSMiniFrameWnd** ppSelf = NULL) ;

// Attributes
public:

// Operations
public:
    void  SetPropSheet(CPropSheetExtended* pPropSheet) ;

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CPSMiniFrameWnd)
    public:
    virtual BOOL Create(LPCTSTR lpszWindowName, CWnd* pParentWnd = NULL);
    //}}AFX_VIRTUAL

// Implementation
protected:
    CPropSheetExtended*  m_pEmbeddedPropSheet;
    CPSMiniFrameWnd**    m_ppSelf;

public:
    virtual ~CPSMiniFrameWnd();

    // Generated message map functions
    //{{AFX_MSG(CPSMiniFrameWnd)
    afx_msg void OnActivate(UINT nState, CWnd* pWndOther, BOOL bMinimized);
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnSetFocus(CWnd* pOldWnd);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
//////////////////////////////////////////////////////////////////
```

Let's look at some of the most important functions in this class (the entire code can be found on the CD). The order in which we consider them reflects the order in which they'll typically get used.

The construction code doesn't do much, except to initialize the pointer to the embedded property sheet to **NULL**.

SetPropSheet()

SetPropSheet() saves a pointer to the property sheet object which will be used with this window:

```
void CPSMiniFrameWnd::SetPropSheet(CPropSheetExtended* pPropSheet)
{
    m_pEmbeddedPropSheet = pPropSheet;
}
```

Create()

As with many MFC classes, instantiating the MFC object doesn't create and display the underlying Windows objects. That is done by another function; in this case, **Create()**. This function creates and shows the window, provided that the user has attached a property sheet to the object.

```
BOOL CPSMiniFrameWnd::Create(LPCTSTR lpszWindowName,
                             CWnd* pParentWnd /*=NULL*/)
```

```
{
   // Make the sure the user has assigned the pointer to the child
   // dialog.
   CRect rectMiniFrame(0,0,0,0);
   BOOL bRtnValue = CMiniFrameWnd::Create(NULL,lpszWindowName,
                        WS_POPUP | WS_CAPTION | WS_SYSMENU,
                        rectMiniFrame, pParentWnd);
   if (bRtnValue)
   {
      CenterWindow();
      ShowWindow(SW_SHOW);
   }
   return bRtnValue;
}
```

OnCreate()

The **ON_WM_CREATE** message is sent to the window after **Create()** has been called, but before the window is made visible. This routine creates the property sheet, and then resizes the frame window so that it fits around the property sheet, and finally puts them both into the correct positions:

```
int CPSMiniFrameWnd::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
   if (CMiniFrameWnd::OnCreate(lpCreateStruct) == -1)
      return -1;

   // Create the modeless property sheet
   if (!m_pEmbeddedPropSheet->Create(this, WS_SYSMENU |
                           WS_CHILD | WS_VISIBLE, 0))
      return -1;

   // Now we want to resize the miniframe window to fit the property
   // sheet inside it. In the following CMiniFrameWnd::CalcWindowRect()
   // adjusts the size of the mini frame window to encompass the property
   // sheet.

   CRect rectSheet, rectMiniFrame;
   m_pEmbeddedPropSheet->GetWindowRect(rectSheet);
   rectMiniFrame = rectSheet;
   CalcWindowRect(rectMiniFrame);

   // Adjust the positions of the miniframe and property sheet
   SetWindowPos(NULL, rectMiniFrame.left, rectMiniFrame.top,
            rectMiniFrame.Width(), rectMiniFrame.Height(),
            SWP_NOZORDER | SWP_NOACTIVATE);

   m_pEmbeddedPropSheet->SetWindowPos(NULL, 0, 0 ,rectSheet.Width(),
             rectSheet.Height(), SWP_NOZORDER | SWP_NOACTIVATE);

   return 0;
}
```

OnActivate()

The **WM_ACTIVATE** message, handled by the **OnActivate()** handler function, is sent when a window is being either activated or deactivated. In our example here, the mini-frame window sends the message on to the embedded property sheet, so that it can take any necessary action:

```
void CPSMiniFrameWnd::OnActivate(UINT nState, CWnd* pWndOther,
    BOOL bMinimized)
{
   CMiniFrameWnd::OnActivate(nState, pWndOther, bMinimized);

   // Get a pointer to the current message
```

```
    const MSG* pMsg = GetCurrentMessage();

    // Send the message to the child property Sheet
    m_pEmbeddedPropSheet->SendMessage(pMsg->message, pMsg->wParam,
                                     pMsg->lParam);
}
```

An Example

The following is an example of how it can be used:

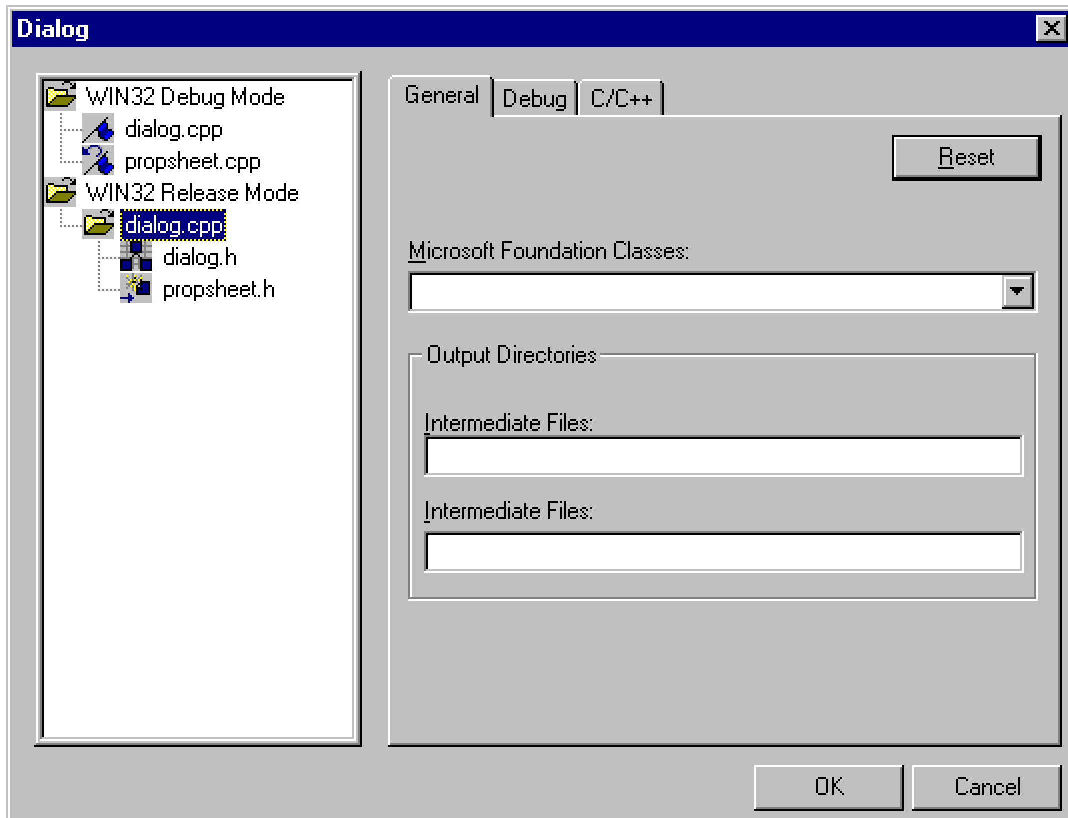
```
void CPropSheetView::OnPSMiniFrame()
{
    m_pPSMiniFrameWnd    = new CPSMiniFrameWnd(&m_pPSMiniFrameWnd);

    m_pEmbeddedPropSheet = new CPSExample2(AFX_IDS_APP_TITLE,
                                   this, 0, &m_pEmbeddedPropSheet);
    m_pEmbeddedPropSheet->AddPage(&m_MiniFramePageGeneral);
    m_pEmbeddedPropSheet->AddPage(&m_MiniFramePageStyle);

    m_pEmbeddedPropSheet->SetDisplayStdButtons();
    m_pPSMiniFrameWnd->SetPropSheet(m_pEmbeddedPropSheet);

    if (!m_pPSMiniFrameWnd->Create(_T("Dialog Properties"), this))
    {
        delete m_pEmbeddedPropSheet;
        delete m_pPSMiniFrameWnd;
         ASSERT(FALSE);
    }
}
```

Note that the above code uses `m_pEmbeddedPropSheet->SetDisplayStdButtons()` to instruct the property sheet not to display the standard buttons which would otherwise have been displayed.

Embedding a Property Sheet Inside a Dialog

Embedding a property sheet inside a dialog is the same as embedding it inside a **`CView`**-derived class, with one important addition, which we'll cover shortly.

The `PropSheet` example program shows how to construct a dialog with an embedded property sheet and a tree list, just like the Project Settings dialog from Developer Studio. We're only going to discuss how the property sheet is implemented.



The Dialog Object

The dialog object is a **CDialogSheet**; this class is derived from **CDialogExtension**, which adds various capabilities to the basic **CDialog** class, including the ability to handle accelerator keys.

When the dialog is created, we need to create the tree list and the property sheet:

```
BOOL CDialogSheet::OnInitDialog()
{
   CDialogExtension::OnInitDialog();
   InitTreeCtrl();
   OnDialogsheetDisplayprop();
   return TRUE;
}
```

The property sheet is set up as before, and three pages are added to it. The next step is to position it correctly in the dialog relative to the tree list:

```
void CDialogSheet::OnDialogsheetDisplayprop()
{
   // Create the modeless property sheet
   m_pPropSheetTest = new CPSExample3(_T("Property Sheet"), this,
                              0, &m_pPropSheetTest);
   m_pPropSheetTest->SetDisplayStdButtons();
   m_pPropSheetTest->AddPage(&m_GeneralPage);
   m_pPropSheetTest->AddPage(&m_DebugPage);
   m_pPropSheetTest->AddPage(&m_CplusPage);

   // Retrieve a ptr to the list ctrl window
```

```
   CWnd* pWnd = GetDlgItem(IDC_DIALOGSHEET_TREE);

   // Retrieve the location of the window
   CRect rectListWnd;
   pWnd->GetWindowRect(rectListWnd);
   ScreenToClient(rectListWnd);
   if (!m_pPropSheetTest->Create(this,
         WS_SYSMENU | WS_CHILD | WS_CLIPSIBLINGS | WS_VISIBLE, 0))
   {
      // Issue error message and return…
      return;
   }

   m_pPropSheetTest->SetWindowPos(NULL, rectListWnd.right+7,
    rectListWnd.top - 5,0, 0, SWP_NOSIZE | SWP_NOZORDER | SWP_NOACTIVATE);
}
```

The Property Sheet Class

Before using property sheets inside a dialog, you need to make one important addition to the property sheet class. Make sure that you set the **WS_EX_CONTROLPARENT** style for the sheet in its **OnInitDialog()** method, like this:

```
BOOL CPSExample3::OnInitDialog()
{
   ModifyStyleEx(0, WS_EX_CONTROLPARENT);
   return CPropSheetGeneral::OnInitDialog();
}
```

If you don't do this, your application will end up in an endless loop!

Embedding a Property Sheet Inside a Splitter Window

Using a property sheet inside a splitter window isn't hard either, because each of the panes in a splitter window is a view, so we can use the same techniques that we have used up to now to embed a property sheet in whichever pane we choose.

The sample program on the CD, called `Splitter`, was created as an SDI application using AppWizard, and choosing splitter window support from the Advanced... options. This sets up your application with the basic mechanics for using a splitter window, but you still have to add your application-specific code.

Creating the Property Sheet View Class

In our case, we need to create a new view class which will display the property sheet, and which will be displayed in the right-hand pane of the splitter window. As in previous examples, this class is derived from **CFormView** and contains data members representing the property sheet and the dynamic dialogs for the tabs.

```
class CFormViewDialog : public CFormView
{
protected:
   CFormViewDialog();           // protected constructor used by dynamic creation
   DECLARE_DYNCREATE(CFormViewDialog)

// Form Data
public:
```

```
    //{{AFX_DATA(CFormViewDialog)
    enum { IDD = IDD_FORMVIEW_DIALOG };
        // NOTE: the ClassWizard will add data members here
    //}}AFX_DATA

// Attributes
public:

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CFormViewDialog)
    public:
    virtual void OnInitialUpdate();
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:
    virtual ~CFormViewDialog();

    // Property sheet member objects
    CPSExample1*    m_pPSView;
    CDebugPage      m_ViewDebugPage;
    CCplusPage      m_ViewCplusPage;
    CGeneralPage    m_ViewGeneralPage;


#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

    // Generated message map functions
    //{{AFX_MSG(CFormViewDialog)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

Setting up the Splitter Window

`CMainFrame::OnCreateClient()` is the place where the splitter window is set up. We use **`CreateView()`** to attach a default view to the left-hand pane, and one of our new views to the right-hand one.

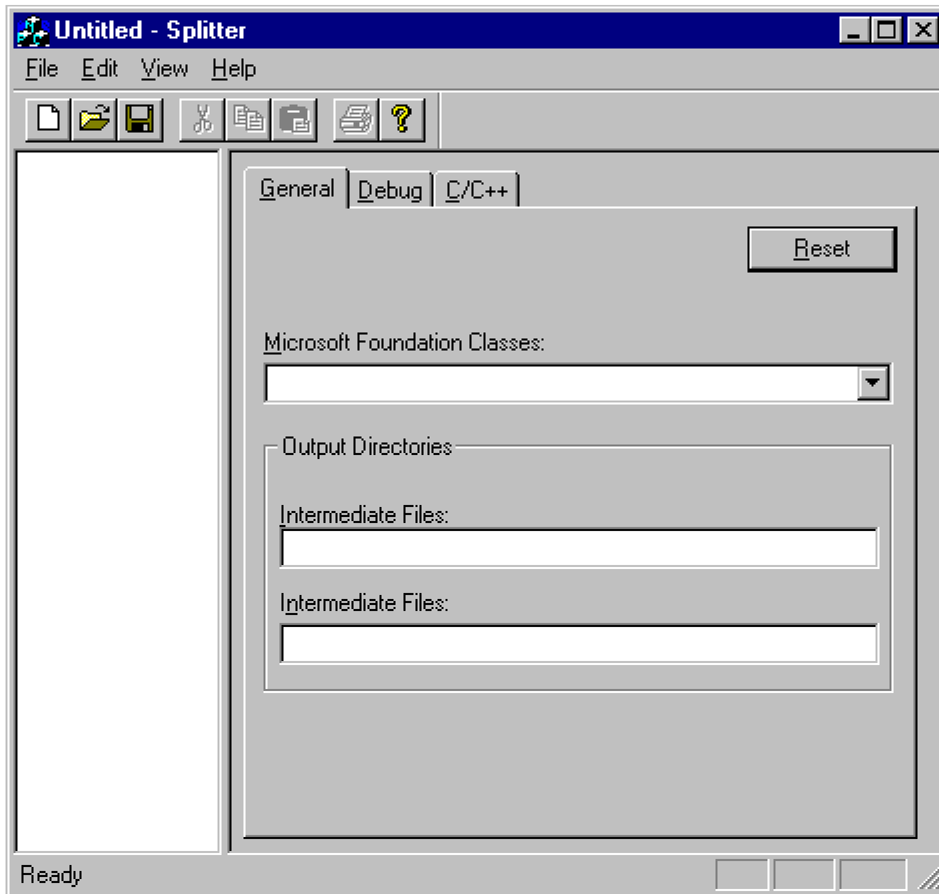
```
BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT /*lpcs*/,
    CCreateContext* pContext)
{
    m_wndSplitter.CreateStatic(this, 1, 2);
    m_wndSplitter.CreateView(0,0, RUNTIME_CLASS(CSplitterView),
                                    CSize(100,100), pContext);
    m_wndSplitter.CreateView(0,1, RUNTIME_CLASS(CFormViewDialog),
                                    CSize(100,100), pContext);
    return TRUE;
}
```

Building the Property Sheet

The process of building the property sheet is quite straightforward. When the view is created, we set up a new property sheet object (a **`CPSExample1`** in this case), and add the pages to it, after which we resize the window to fit the property sheet:

```
int CFormViewDialog::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
   if (CFormView::OnCreate(lpCreateStruct) == -1)
      return -1;

   // Allocate memory for the property sheet and make sure
   // that the standard buttons don't show up.
   m_pPSView = new CPSExample1(_T(""), this, 0, &m_pPSView);
   m_pPSView->SetDisplayStdButtons();

   m_pPSView->AddPage(&m_ViewGeneralPage);
   m_pPSView->AddPage(&m_ViewDebugPage);
   m_pPSView->AddPage(&m_ViewCplusPage);
   if (!m_pPSView->Create(this, WS_SYSMENU | WS_CHILD | WS_VISIBLE, 0))
      return -1;

   // Now we want to resize the window to fit the property sheet inside
   // it. CalcWindowRect() adjusts the size of the window to encompass the
   // property sheet.

   CRect rectSheet, rectWindow;
   m_pPSView->GetWindowRect(rectSheet);
   rectWindow = rectSheet;
   CalcWindowRect(rectWindow);

   // Adjust the positions of the window and property sheet
   SetWindowPos(NULL, rectWindow.left,rectWindow.top,
                   rectWindow.Width() + 5,
                rectWindow.Height(), SWP_NOZORDER | SWP_NOACTIVATE);

   m_pPSView->SetWindowPos(NULL, 0, 0 ,rectSheet.Width(),
                rectSheet.Height(), SWP_NOZORDER | SWP_NOACTIVATE);

   return 0;
}
```

And here you can see what you should end up with:



Summary

And there we have it. You can find all of the code we've discussed throughout the chapter, put to use in various projects on the CD. You'll also find some extra code and classes not discussed in this chapter, so it's definitely worth taking a look. Dialogs and property sheets are valuable members of the arsenal of controls that you have available to help you write Windows programs, and after reading this chapter, we hope that you'll be better equipped to use them effectively.

Windows Multimedia Services

In this chapter, we'll look at the basic multimedia services that are now standard features of the Win32 API. The core multimedia functions offered by the Windows environment are found in the Windows Multimedia System (**`Winmm.lib`**) library, which ships with Visual C++ 4.1.

The Windows Multimedia System first appeared as an add-on to Windows 3.0, probably as a response to the rise of the Apple Macintosh as the widely-recognized leader in multimedia. The add-on was shipped as a dynamic-link library called **`Mmsystem.dll`**, and the name MMSYSTEM is frequently applied to the features it contains. MMSYSTEM became a standard part of Windows with version 3.1 and is now called WINMM in the Win32 environments. MMSYSTEM provides a very versatile set of APIs for controlling audio and video devices, including CD-audio, AVI, MIDI and external devices like video-disc players. MMSYSTEM finally made Windows a serious contender in the multimedia market.

Overview

Let's take a high-level look at what's included in the Windows Multimedia System:

MCI

The Media Control Interface (MCI) provides a high-level, device-independent way to get basic multimedia services into your apps. If you just need playback or simple recording capabilities, MCI is the quickest path for you. However, if you need to do anything fancier with your media (like signal-processing or graphing an audio wave, for instance), MCI may not be able to handle it.

Low-level API

The Multimedia System includes low-level APIs for everything it does, so you can do just about anything you want. Unlike MCI, however, the low-level APIs require much more of a grasp of the underlying devices and media. The low-level APIs give you finer control over your apps' interaction with media devices.

Multimedia File (RIFF) I/O Services

When you're using the low-level APIs, for some special applications (like media editors), you'll need to crack open a Resource Interchange File Format (RIFF) media file and inspect or alter its contents. This isn't for the faint of heart, but the Multimedia System does provide RIFF file I/O services. A wave (**`.wav`**) file is an example of a RIFF file.

The Media Control Interface

The Media Control Interface (MCI) provides a very high-level, device-independent way to control media devices from Windows programs. MCI capitalizes on the fact that most media devices are similar in the interface they present to the user. CD players, video-tape and disc players, and MIDI sequencers all share some common **media transport controls**. (For instance, they all have some form of Play, Stop, Pause, and Rewind controls.) Following this model, MCI defines a set of transport commands that MCI devices can implement. An MCI device can be queried to determine which commands it supports (for instance, not all MCI devices support recording). Then, application programs can simply instruct an MCI device to play (or stop, etc.) without needing to know any of the operating details of the device or the media it supports. This approach makes MCI exceptionally easy to use and highly device-independent.

How MCI Works

To be accessible to MCI, a device needs to have an MCI device driver installed. The MCI driver provides the high-level interfaces to control the device. MCI device drivers can be controlled by two different methods; they can be sent commands in the form of text strings, or they can be sent command messages through theWINMM API.

The MCI Command-string Interface

The command-string interface is designed to facilitate device control by high-level, interpreted programming environments like Visual Basic. The programmer constructs device control commands as English-like strings (**"play fanfare.wav"**) and sends them to the device.

The MCI Command Message Interface

The command-message interface provides a more efficient means of controlling MCI devices for C programmers. Instead of constructing strings for commands, the commands are sent directly through API calls, like:

```
mciSendCommand(mci_device, MCI_PLAY, 0,(DWORD)(LPVOID)&play_struct);
```

To use the command-message interface, you must open the appropriate MCI device, construct and send the appropriate command messages, and close the device.

The MCIWnd Window Class

Perhaps the easiest way to incorporate MCI media playback into your Visual C++ app is to use the MCIWnd control from the Video for Windows library, which ships as a standard feature of Visual C++ 4.1. You can create one of these handy control windows in your app, and it will handle all of the MCI device communication for you, and manage the display and/or audio playback of the device's output.

Creating an MCIWnd

MCIWnd is a registered window class, like any of the standard control window classes. You can create an MCIWnd either as an independent window or as a child control on one of your app's windows.

MCIWndCreate()

You create an MCIWnd by calling the function **MCIWndCreate()**, which is defined as follows:

```
HWND MCIWndCreate(
   HWND hwndParent,       // HWND of parent (can be NULL)
   HINSTANCE hInstance,   // Instance handle of app
   DWORD dwStyle,         // Window style flags
   LPCSTR szFile          // File name of media file to load
   );
```

If **hwndParent** is **NULL** and **dwStyle** is **0**, the MCIWnd style defaults to **WS_VISIBLE|WS_OVERLAPPEDWINDOW**. If a parent **HWND** is given and **dwStyle** is **0**, the style defaults to **WS_VISIBLE|WS_CHILD|WS_BORDER**.

MCIWnd Styles

Aside from the regular window styles, MCIWnd also supports some special styles of its own that control its behavior and appearance. Here are some of them:

MCIWnd Style	Meaning
MCIWNDF_NOAUTOSIZEWINDOW	Don't resize the window to fit the movie.
MCIWNDF_NOPLAYBAR	Hide the transport controls.
MCIWNDF_NOAUTOSIZEMOVIE	Don't resize the movie to fit the window.
MCIWNDF_NOMENU	Don't post the menu when the right button is clicked on MCIWnd.
MCIWNDF_SHOWNAME	Show the file name in caption.
MCIWNDF_SHOWPOS	Show the position in caption.
MCIWNDF_SHOWMODE	Show the mode in caption.
MCIWNDF_SHOWALL	Show everything in caption (MCIWNDF_SHOWNAME| MCIWNDF_SHOWPOS| MCIWNDF_SHOWMODE).
MCIWNDF_RECORD	Allow recording and include a Record button on the playbar.
MCIWNDF_NOOPEN	Don't allow the user to open files from the MCIWnd menus.

Let's look at an example that you might put in the code for a view class in your own code:

```
HWND hwndMCI = MCIWndCreate(NULL, AfxGetInstanceHandle(),
   WS_VISIBLE|WS_OVERLAPPED|MCIWNDF_SHOWALL|MCIWNDF_NOOPEN,
   "intro.avi");
```

This code creates a free-floating MCIWnd showing the movie file **Intro.avi** with the media file name, play mode and playback position in the caption, and prevents users from opening other media files from the MCIWnd's menus.

Controlling an MCIWnd: MCIWnd Macros

Once you've created an MCIWnd, you can send it instructions through a set of macros defined in the Video for Windows header (**vfw.h**). These macros send messages to the MCIWnd to tell it what to do, and there are quite a few of them, so here are some of the most commonly-used ones:

Macro	What It Does
BOOL MCIWndCanPlay(HWND mciwnd)	Returns **TRUE** if the device is capable of playback.
BOOL MCIWndCanRecord(HWND mciwnd)	Returns **TRUE** if the device is capable of recording.
BOOL MCIWndCanEject(HWND mciwnd)	Returns **TRUE** if the device is capable of ejecting media from device.
BOOL MCIWndCanSave(HWND mciwnd)	Returns **TRUE** if the device is capable of saving media data to a file.
MCIWndPlay(HWND mciwnd)	Starts media playback.
MCIWndStop(HWND mciwnd)	Stops media playback.
MCIWndPause(HWND mciwnd)	Pauses media playback.
MCIWndResume(HWND mciwnd)	Resumes playback after a Pause.
MCIWndHome(HWND mciwnd)	Rewinds the media to the beginning.

`MCIWndEnd(HWND mciwnd)`	Advances the media transport to the end of the media.
`MCIWndSeek(HWND mciwnd, LONG lPos)`	Advances the media transport to the given position.
`MCIWndEject(HWND mciwnd)`	Ejects the media from the device.
`MCIWndRecord(HWND mciwnd)`	Starts the media recording.
`MCIWndSave(HWND mciwnd, LPCSTR szFile)`	Saves the media to a file.
`MCIWndDestroy(HWND mciwnd)`	Shuts down the MCIWnd.

Now let's see how to use MCI, through the MCIWnd, in an actual C++ app.

Project MCIPlayer - A Simple Media Player Using MCIWnd

The purpose of this project is to illustrate the use of the MCIWnd object provided in the Video for Windows library. You can use the techniques covered in this project to easily add basic multimedia playback capabilities to your own apps.

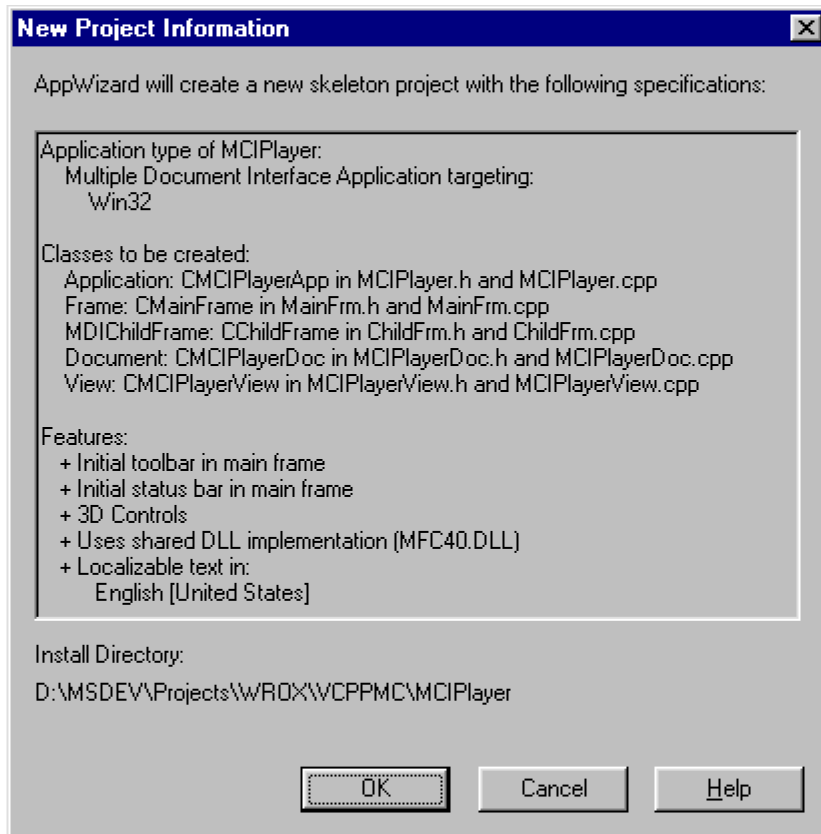
The MCIPlayer app is a very simple program. It allows the user to open multimedia files as documents and play them. Each open file gets its own MDI view, containing only an MCIWnd control for playing the file.

Step 1

Use AppWizard to create a new project entitled MCIPlayer. Create the project using the following AppWizard options:

AppWizard Step 1: use the default setting (multiple documents).
AppWizard Step 2: use the default setting (no database support).
AppWizard Step 3: use the default settings (no OLE support).
AppWizard Step 4: use the default settings, except turn off printing and print preview support (we won't be using it).
AppWizard Step 5: use the default settings.
AppWizard Step 6: use the default settings.

If everything worked all right, you should wind up with a New Project Information dialog that looks like this:


New Project Information
AppWizard will create a new skeleton project with the following specifications:
Application type of MCIPlayer:
Multiple Document Interface Application targeting:
Win32
Classes to be created:
Application: CMCIPlayerApp in MCIPlayer.h and MCIPlayer.cpp
Frame: CMainFrame in MainFrm.h and MainFrm.cpp
MDIChildFrame: CChildFrame in ChildFrm.h and ChildFrm.cpp
Document: CMCIPlayerDoc in MCIPlayerDoc.h and MCIPlayerDoc.cpp
View: CMCIPlayerView in MCIPlayerView.h and MCIPlayerView.cpp
Features:
+ Initial toolbar in main frame
+ Initial status bar in main frame
+ 3D Controls
+ Uses shared DLL implementation (MFC40.DLL)
+ Localizable text in:
English [United States]
Install Directory:
D:\MSDEV\Projects\WROX\VCPPMC\MCIPlayer
OK
Cancel
Help

Step 2

We need to add a member variable to the **CMCIPlayerView** class to hold the window handle of the MCIWnd control that we're going to create later. Go to the class definition for **CMCIPlayerView** and add:

```
private:
   HWND    m_hwndMCI;
```

Step 3

We'll actually create the MCIWnd control in the **OnInitialUpdate()** function for **CMCIPlayerView**. Use ClassWizard to add an **OnInitialUpdate()** function override to **CMCIPlayerView**, and put the following code in it:

```
void CMCIPlayerView::OnInitialUpdate()
{
   CView::OnInitialUpdate();

   // Get the filename of the opened MCI file from the document
   CString avifile = GetDocument()->GetPathName();

   // Create the MCI window as a child of the view
   m_hwndMCI = MCIWndCreate(this->GetSafeHwnd(), AfxGetInstanceHandle(),
      WS_CHILD|WS_VISIBLE|MCIWNDF_NOOPEN, avifile);

}
```

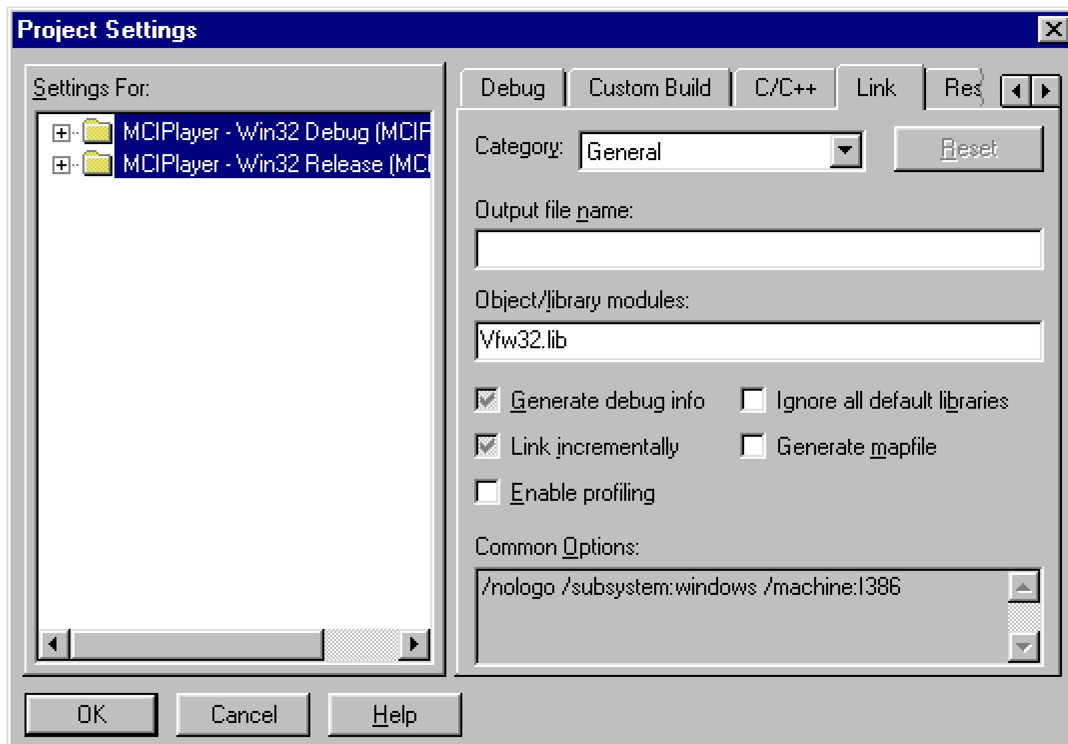
We're doing two things here. First, we're retrieving the name of the file for which this view was opened.

Then we're creating an MCIWnd control as a child of the view, telling it the file name of the media file for the document. We're going to use the **MCIWNDF_NOOPEN** style so that users must go through the app's file open mechanism to open new media files.

While we're in this module, we need to include the header for the Video for Windows library, since we're creating an MCIWnd here. Go to the top of this file and add,

```
#include <vfw.h>
```

after the other **#include**s you find up there. Also, we'll need to link with the Video for Windows library. Under the Developer Studio Build menu, choose Settings.... Go to the Link tab and add Vfw32.lib under the Object/library modules: heading:



Step 4

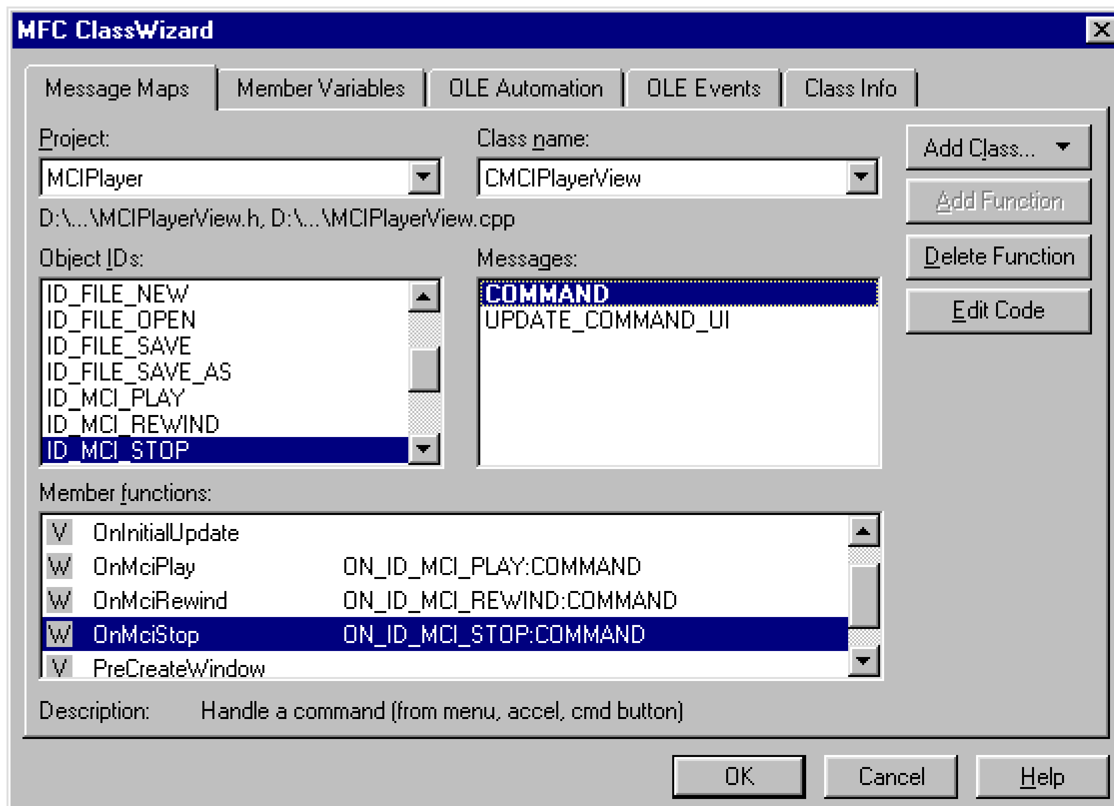
We need to block the creation of a new, blank document at start up in this app, since it's really just a file viewer. Go to the **InitInstance()** function in **CMCIPlayerApp** and add the following to the command-line processing code:

```
// Parse command line for standard shell commands, DDE, file open
CCommandLineInfo cmdInfo;
ParseCommandLine(cmdInfo);

// Dispatch commands specified on the command line
if (cmdInfo.m_nShellCommand != CCommandLineInfo::FileNew)
   if (!ProcessShellCommand(cmdInfo))
      return FALSE;
```

At this point, you can build and run the app so far. You should be able to open any **.wav**, **.mid**, or **.avi** file and use the MCIWnd control to play it.

Step 5

Let's enhance the app (and explore MCIWnd further) by adding some media transport controls. We'll add Play, Stop and Rewind functions to the app. First, add a new menu to the document menu bar (**IDR_MCIPLATYPE**). Delete the Edit menu from this menu bar and insert the new menu in its place. Call the new menu MCI. Add three items to this menu: Play, Stop, and Rewind. Using ClassWizard, add command handlers for each of these items to **CMCIPlayerView**.

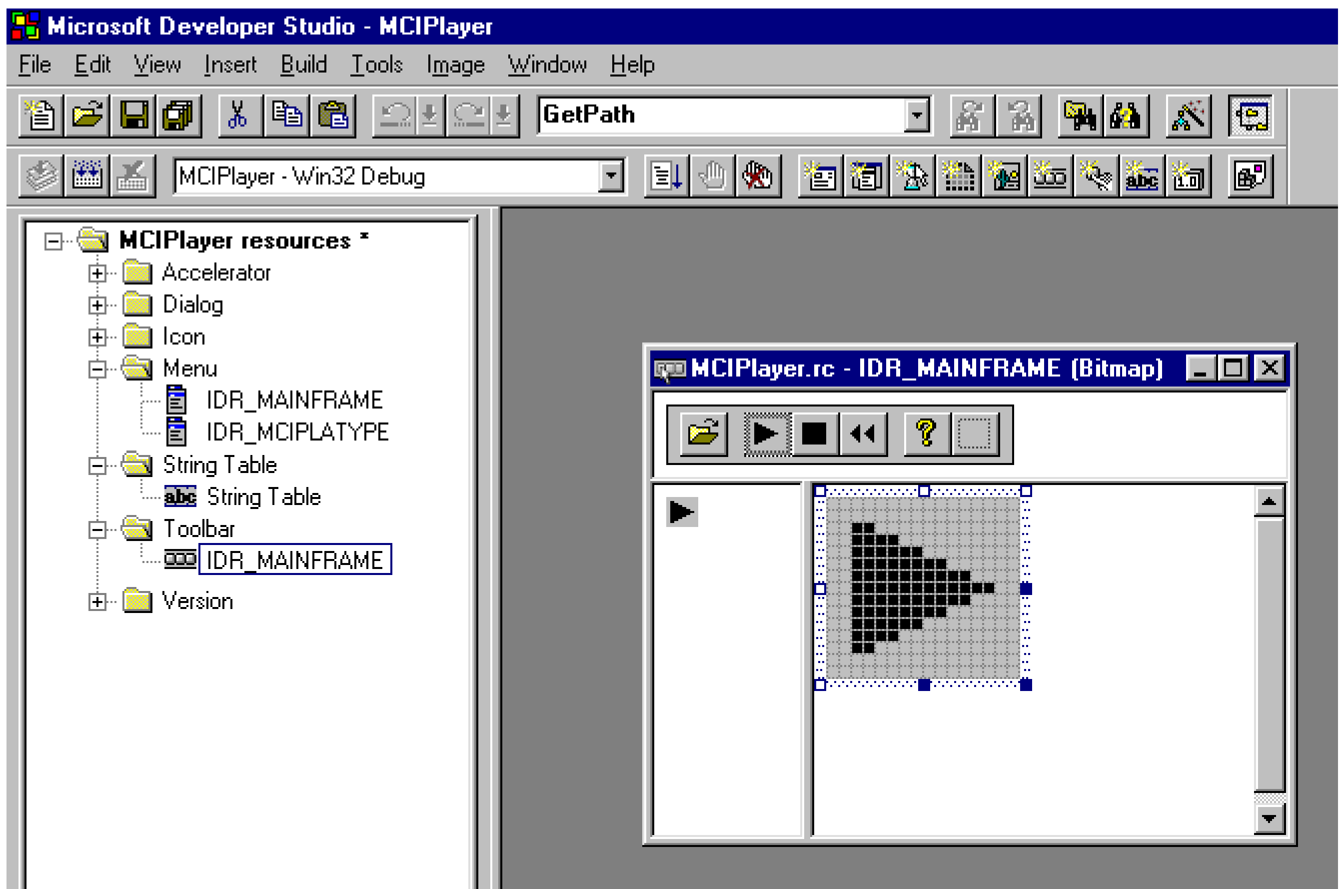


Invoke the appropriate MCIWnd control macro in each handler:

```
void CMCIPlayerView::OnMciPlay()
{
    MCIWndPlay(m_hwndMCI);
}

void CMCIPlayerView::OnMciRewind()
{
    MCIWndHome(m_hwndMCI);
}

void CMCIPlayerView::OnMciStop()
{
    MCIWndStop(m_hwndMCI);
}
```

Step 6

Finally, let's fix up the toolbar. Edit the app's tool bar resource and remove the File New, File Save, and Print buttons. Also remove the Cut, Copy and Paste buttons. In their place, add buttons for Play, Stop, and Rewind.



That's it! Here's what the finished app looks like:

MCIPlayer - Windmill.avi
File
MCI
View
Window
Help
Prvtjet.avi
Windmill.avi
Eagle.wav
Goodvib3.mid
Ready

The Low-level APIs

As we've just seen, MCI certainly is handy for getting simple multimedia services into your app quickly, but what if you need to do more with your multimedia data than MCI allows? What if you want to read and write **`.wav`** data? What if you want to interpret or generate individual MIDI events? What if you want to be able to inspect raw audio wave data so you can draw it, edit it, or synthesize it? The low-level Multimedia System APIs let you do all of these things and more. For our purposes here, we're calling everything in the Windows Multimedia System that's not part of MCI the **low-level APIs**.

The Organization of the Low-level APIs

Let's start off with a quick survey of what's available in the low-level APIs:

Feature Area	Function Names Start With
Various wave audio functions	**`wave`**
Low-level MIDI	**`midi`**
Multimedia file I/O	**`mmio`**
Joystick input functions	**`joy`**
High-resolution timers	**`time`**

To give you a feel for programming with the low-level Multimedia System APIs, we're going to devote the rest of this chapter to an in-depth look at one of the most popular uses of the Multimedia System: working with wave audio. In so doing, we'll hit just about every major concept in the low-level APIs.

Working with PCM Wave Audio

For the purposes of our discussion of wave audio, we'll focus on the basic PCM (Pulse-Code Modulated) wave audio format used by the Sound Recorder applet and all of the **`.wav`** files thatship with Windows 95 and the Plus! Pack. All popular **`.wav`** file editors also support this format. First, we'll look briefly at the **`PlaySound()`** function, then we'll look at the basics of wave audio devices and wave data structures. To get a real idea of what it's like to program with the low-level API, we're going build a class called **`CPCMWave`** that encapsulates PCM wave audio to simplify its use in a program. Finally, we'll build a Visual C++ project, WaveScope, using **`CPCMWave`**.

The PlaySound() Function

Before we delve into the more sophisticated mechanisms in the low-level wave audio API, it's worth taking a quick look at the **`PlaySound()`** function. Though it's technically a part of the low-level API, using **`PlaySound()`** is perhaps the easiest way (even easier than using MCI), to trigger wave audio playback in a C++ application.

`PlaySound()` is defined as follows:

```
BOOL PlaySound(LPCSTR pszSound, HMODULE hmod, DWORD fdwSound);
```

Parameter	Meaning

pszSound	The name of the sound to play. The meaning of this name varies with the setting of **fdwSound**. If **NULL**, any wave currently playing is stopped.
hmod	Only valid when **fdwSound** is set to **SND_RESOURCE**. Identifies the file handle of an executable file from which to load the sound resource named by **pszSound**.
fdwSound	Playback flags. Some combination of the flags in the following table.

The possible flags are:

Flag	Meaning
SND_APPLICATION	Use app-specific association.
SND_ALIAS	**pszSound** specifies a system sound from the registry. Mutually-exclusive with **SND_FILENAME** and **SND_RESOURCE**
SND_ALIAS_ID	**pszSound** contains a predefined sound ID.
SND_ASYNC	Play the wave asynchronously (don't wait for it to finish before returning).
SND_FILENAME	**pszSound** contains a **.wav** file name.
SND_LOOP	Used with **SND_ASYNC** to play a sound over and over again, until **PlaySound()** is called with **pszSound = NULL**.
SND_MEMORY	**pszSound** is a pointer to a memory image of a wave file.
SND_NODEFAULT	Fail if sound not found. Don't try to default.
SND_NOSTOP	Don't attempt to stop a sound that is already playing on the device.
SND_NOWAIT	Fail immediately if the driver is busy.
SND_PURGE	Abort all sounds playing for the calling task.
SND_RESOURCE	**pszSound** is the ID of a resource in the module **hmod**.
SND_SYNC	Play the wave synchronously (wait for it to finish before returning).

For example, to use **PlaySound()** to play the wave file **Tada.wav** asynchronously, we could put the following in our code:

```
PlaySound("tada.wav", 0, SND_FILENAME|SND_ASYNC);
```

Low-level Wave Audio Devices and WAVE_MAPPER

Before we jump into the details of wave audio data, let's look at the devices that will handle it. A Windows system has zero or more wave audio devices installed. Most new systems now come with a sound card, so usually there's at least one wave audio device available to Windows. Sometimes, however, there are multiple devices installed (or multiple drivers, like a manufacturer-supplied driver and a Microsoft Windows Sound System driver for the same board). Because not all devices support playback and recording, wave input and output functions are considered as separate 'devices', even if they reside on the same physical board.

When we're calling into the low-level wave audio API, we'll need to identify the wave audio input or

output device that we want to use. Audio devices are referenced by a device ID number, ranging from 0 to one less than the number of devices installed. So, if you have two wave output devices installed, they'll be numbered 0 and 1. The following snippet of code will tell you how many wave audio input and output devices you have:

```
#include <mmsystem.h>  // And link with winmm.lib

UINT nInDevs = waveInGetNumDevs();
UINT nOutDevs = waveOutGetNumDevs();
```

Next, you might want to know what capabilities your wave audio devices have. The following code fetches the device capabilities for wave in device 0 and wave out device 0:

```
WAVEINCAPS wic;
WAVEOUTCAPS woc;

waveInGetDevCaps(0 /* device 0 */, &wic, sizeof(WAVEINCAPS));
waveOutGetDevCaps(0 /* device 0 */, &woc, sizeof(WAVEOUTCAPS));
```

`WAVEINCAPS` and **`WAVEOUTCAPS`** are **`struct`**s that are defined as follows:

```
typedef struct {
    WORD      wMid;
    WORD      wPid;
    MMVERSION vDriverVersion;
    CHAR      szPname[MAXPNAMELEN];
    DWORD     dwFormats;
    WORD      wChannels;
    WORD      wReserved1; // padding
} WAVEINCAPS;

typedef struct {
    WORD      wMid;
    WORD      wPid;
    MMVERSION vDriverVersion;
    CHAR      szPname[MAXPNAMELEN];
    DWORD     dwFormats;
    WORD      wChannels;
    WORD      wReserved1;  // padding
    DWORD     dwSupport;
} WAVEOUTCAPS;
```

They are the same, except for the **`dwSupport`** member in **`WAVEOUTCAPS`**. Here's a table showing what the members mean:

Member	Meaning
`wMid`	The manufacturer ID number.
`wPid`	The product ID number.
`vDriverVersion`	The 16-bit version ID: high byte is major version number, low byte is minor version number.
`szPname`	A string containing the driver name.
`dwFormats`	A bit field containing flags for the standard formats supported by the driver. See the table below.
`wChannels`	The number of simultaneous channels the device supports (usually 1 (mono) or 2 (stereo)).
`wReserved1`	Not used.

dwSupport (**WAVEOUTCAPS** only)	A bit field containing flags for optional playback features supported by the driver. See the second table below.

The format flags are:

Flag	Meaning
WAVE_FORMAT_1M08	11.025 kHz, mono, 8-bit
WAVE_FORMAT_1M16	11.025 kHz, mono, 16-bit
WAVE_FORMAT_1S08	11.025 kHz, stereo, 8-bit
WAVE_FORMAT_1S16	11.025 kHz, stereo, 16-bit
WAVE_FORMAT_2M08	22.05 kHz, mono, 8-bit
WAVE_FORMAT_2M16	22.05 kHz, mono, 16-bit
WAVE_FORMAT_2S08	22.05 kHz, stereo, 8-bit
WAVE_FORMAT_2S16	22.05 kHz, stereo, 16-bit
WAVE_FORMAT_4M08	44.1 kHz, mono, 8-bit
WAVE_FORMAT_4M16	44.1 kHz, mono, 16-bit
WAVE_FORMAT_4S08	44.1 kHz, stereo, 8-bit
WAVE_FORMAT_4S16	44.1 kHz, stereo, 16-bit

The supported features flags are:

Flag	Meaning
WAVECAPS_LRVOLUME	The left and right volumes can be set separately.
WAVECAPS_PITCH	Allows playback pitch control.
WAVECAPS_PLAYBACKRATE	Allows playback rate control.
WAVECAPS_SYNC	Playback is synchronous only.
WAVECAPS_VOLUME	Allows playback volume control.
WAVECAPS_SAMPLEACCURATE	Playback position queries will be sample-accurate.

It's great to be able to get all of this information about the wave devices, and to be able to enumerate them, but it seems awfully cumbersome to have to do this every time you want to identify and use the main audio devices on a Windows system. To simplify things a bit, Microsoft provides a special device ID, **WAVE_MAPPER**, that you can use to grab the default or best-fit device for your purposes. You simply use the **WAVE_MAPPER** constant in the wave audio function calls that take a device ID, and you'll automatically get the right device.

The Structure of PCM Wave Data

To send wave audio data to a wave device, we have to make sure that it's in the right format. We'll use a **WAVEFORMATEX** structure to describe the data format of a wave. When we're attempting to open a wave device to use our wave data, we'll use the **WAVEFORMATEX struct** to tell the driver what it needs to be able to support to handle the data. The driver can then decide whether the wave data format will work and report back to us. The **WAVEFORMATEX struct** looks like this:

```
typedef struct {
    WORD  wFormatTag;
    WORD  nChannels;
    DWORD nSamplesPerSec;
    DWORD nAvgBytesPerSec;
    WORD  nBlockAlign;
    WORD  wBitsPerSample;
    WORD  cbSize;
} WAVEFORMATEX;
```

Member	Meaning
wFormatTag	The wave data format type (PCM, ADPCM, etc.). The **Mmreg.h** header file defines the latest wave formats registered with Microsoft. The most common one is **WAVE_FORMAT_PCM**, which we'll use in this chapter.
nChannels	The number of channels of sample data (usually 1 or 2).
nSamplesPerSec	The sample rate, in Hertz.
nAvgBytesPerSec	The average number of bytes to read per second of playback. Varies with the data format. For **WAVE_FORMAT_PCM**, it's just **nSamplesPerSec** x **nBlockAlign**.
nBlockAlign	The total bytes required to be read per sample period. For **WAVE_FORMAT_PCM**, it's (**nChannels** x **wBitsPerSample**)/8.
wBitsPerSample	The bitwidth of the sample data. For **WAVE_FORMAT_PCM**, this must be 8 or 16.
cbSize	The size of additional data after the end of this **struct** (not used by **WAVE_FORMAT_PCM**, but used by every compression format).

To set up for the playback of a 16-bit stereo 22.05kHz PCM wave, you would fill out a **WAVEFORMATEX** like this:

```
WAVEFORMATEX wfmt;

wfmt.wFormatTag       = WAVE_FORMAT_PCM;    // Vanilla PCM
wfmt.nChannels        = 2;                  // Stereo
wfmt.nSamplesPerSec   = 22050;              // 22.05kHz sample rate
wfmt.wBitsPerSample   = 16;                 // Bitwidth = 16 bits
wfmt.nBlockAlign      = 4;                  // (nChannels * wBitsPerSample) / 8
wfmt.nAvgBytesPerSec  = 88200;              // nSamplesPerSec * nBlockAlign
wfmt.cbSize           = 0;                  // No extra data for vanilla PCM
```

You send a block of wave data to a wave device (or receive it from one), via a header **struct**, called **WAVEHDR**. A **WAVEHDR** points to the block of raw sample data and gives a few extra flags to describe what's going on with the data. Before you use the **WAVEHDR**, you must be 'prepare' it by calling either **waveInPrepareHeader()** or **waveOutPrepareHeader()**, depending on whether you're going to use it for input or output (more on that later). The **WAVEHDR struct** looks like this:

```
typedef struct {
    LPSTR  lpData;
    DWORD  dwBufferLength;
    DWORD  dwBytesRecorded;
    DWORD  dwUser;
    DWORD  dwFlags;
    DWORD  dwLoops;
    struct wavehdr_tag far * lpNext;
    DWORD  reserved;
```

```
} WAVEHDR;
```

Member	Meaning
lpData	The address of wave data buffer.
dwBufferLength	The buffer size, in bytes.
dwBytesRecorded	The number of bytes recorded (input only).
dwUser	The 32-bit value for your use.
dwFlags	The flags to indicate what to do with data or current status. This can be any combination of the flags in the following table.
dwLoops	The number of times to loop (output only).
lpNext	Reserved. Must be 0.
reserved	Reserved. Must be 0.

The flags can be:

Flag	Meaning
WHDR_BEGINLOOP	Start looping at this block (output only).
WHDR_DONE	The wave device is done playing this block (output only).
WHDR_ENDLOOP	Stop looping at this block (output only).
WHDR_INQUEUE	The buffer is currently queued for playback (output only).
WHDR_PREPARED	The header has been prepared by **waveInPrepareHeader()** or **waveOutPrepareHeader()**.

The actual sample data is stored in a big block of memory on its own, as a string of bytes whose meaning depends on the audio format being used. For **WAVE_FORMAT_PCM**, there are four possible arrangements of the raw sample data. In each case, the data is made up of blocks of one or more bytes, with each block representing the data for all channels for a given sample period:

PCM Format	Data Block Layout in Memory
8-bit mono	1 byte per block; each byte is a separate sample.
8-bit stereo	2 bytes per block; byte 1 is left channel sample, byte 2 is right channel sample.
16-bit mono	2 bytes per block; byte 1 is low-order byte of sample, byte 2 is high-order byte of sample.
16-bit stereo	4 bytes per block; byte 1 is low-order byte of left channel sample, byte 2 is high-order byte of left channel sample, byte 3 is low-order byte of right channel sample, byte 4 is high-order byte of right channel sample.

8-bit Mono

Sample 1	Sample 2	Sample 3	Sample 4	Sample 5	Sam
Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	By

8-bit Stereo

Sample 1 Left	Sample 1 Right	Sample 2 Left	Sample 2 Right	Sample 3 Left	Sam Ri
Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	By

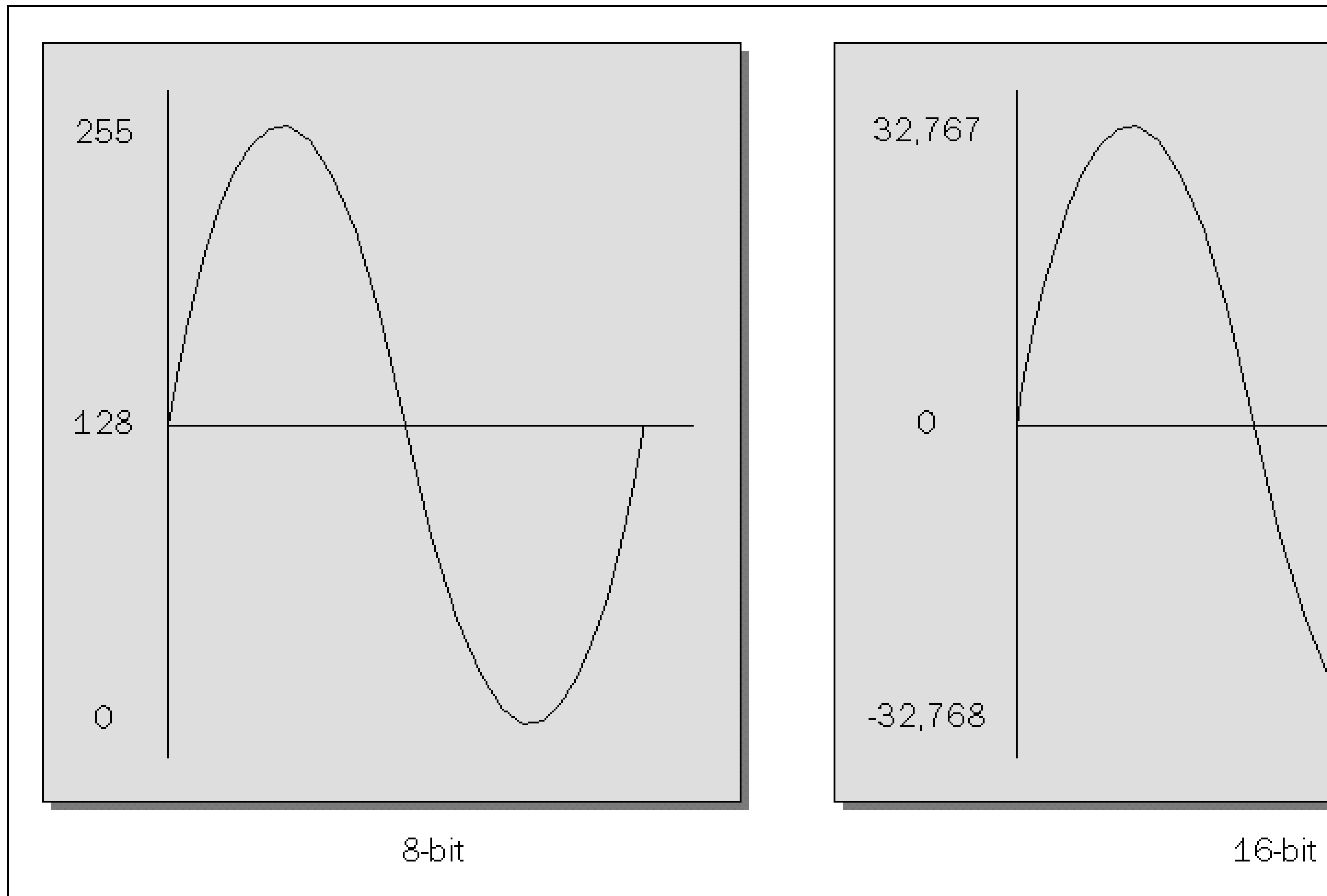
16-bit Mono

Sample 1		Sample 2		Sample 3	
Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	By

16-bit Stereo

Sample 1 - Left		Sample 1 - Right		Sample 2 - Left	
Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	By

Before you can do anything with the raw data once you've got access to it, you need to know what it actually means. Basically, the individual channel wave samples represent the relative linear amplitude (height) of the audio wave at the instant in time when the sample was recorded. If you were to plot the sample values on a graph and connect them together, you'd get back the shape of the original waveform. The problem is that waves are usually bipolar. In other words, they flip-flop between positive-going and negative-going values, centered on zero. 8-bit data ranges from 0-255 (no negative numbers) and 16-bit data ranges from 0-65535. To overcome this, 16-bit wave data is represented in memory as a C++ signed `short int`. For 8-bit data, it's a little different. Instead of using 2's complement signed binary arithmetic, the values simply range from 0-255, with 128 representing zero. The graph below should help make this clearer:



An important note about memory allocation for **WAVEHDR**s and raw sample data is that both of these need to be allocated using **GlobalAlloc()** with the **GMEM_MOVEABLE** and **GMEM_SHARE** (yes, still under Win32!) flags, since they will actually be handed off to the wave device to manage during playback or recording.

OK, now you know a few things about PCM wave audio. You use a **WAVEFORMATEX struct** to describe the format and general playback requirements of a wave. You use a **WAVEHDR** to actually pass the wave data to a wave audio device. You place the wave data in a big block of bytes, following the block format and amplitude value guidelines we've just laid out. Now let's look at how to actually play it back on a wave output device.

Using a Wave Device for Playback

There are basically seven steps to using the low-level audio API to play a sound:

1. Set up a **WAVEFORMATEX**.
2. Open the wave output device.
3. Prepare a **WAVEHDR**.
4. Write the sample data and **WAVEHDR** to the device.
5. Wait until device completes playback (either synchronously, or via **MM_WOM_DONE**

window message or via function callback).

6 Unprepare the **WAVEHDR**.

7 Close the wave output device.

The following code snippet shows how all of this goes together:

```
// Our block of raw wave data lives in here:
HPSTR   hpRawDataBuffer;
long    bufferLength;   // number of bytes in buffer

// (Some code to fill the buffer with wave samples, or block could have
// been read from file earlier)

// 1. set up a WAVEFORMATEX
WAVEFORMATEX wfmt;

wfmt.wFormatTag       = WAVE_FORMAT_PCM;  // Vanilla PCM
wfmt.nChannels        = 2;                // Stereo
wfmt.nSamplesPerSec   = 22050;            // 22.05kHz sample rate
wfmt.wBitsPerSample   = 16;               // Bitwidth = 16 bits
wfmt.nBlockAlign      = 4;                // (nChannels * wBitsPerSample) / 8
wfmt.nAvgBytesPerSec  = 88200;            // nSamplesPerSec * nBlockAlign
wfmt.cbSize           = 0;                // No extra data for vanilla PCM

// 2. open the wave output device and tell it the format we want
HWAVEOUT hWaveOut;      // Handle to receive the wave device handle

waveOutOpen(&hWaveOut,   // Device handle to fill
   WAVE_MAPPER,          // ID of device to open
   &pcmWF,               // WAVEFORMATEX describing wave
   0,                    // Callback target - not used here
   0,                    // Callback user data - not used here
   0 );                  // Open flags - 0 means no callback

// 3. prepare a WAVEHDR

// 3a. Globally-allocate the wave header
HANDLE hWaveHdr = GlobalAlloc(GMEM_MOVEABLE|GMEM_SHARE,
   (DWORD)sizeof(WAVEHDR));

// 3b. Lock it down
LPWAVEHDR lpWaveHdr = (LPWAVEHDR) GlobalLock(hWaveHdr);

// 3c. Set the required fields for playback
lpWaveHdr->lpData = hpRawDataBuffer;
lpWaveHdr->dwBufferLength = bufferLength;
lpWaveHdr->dwFlags = 0;

// 3d. Prepare the header
waveOutPrepareHeader(hWaveOut, lpWaveHdr, sizeof(WAVEHDR));

// 4. write the sample data and WAVEHDR to the device
waveOutWrite(hWaveOut, lpWaveHdr, sizeof(WAVEHDR));

// 5. wait until device completes playback
// This is a simple way to approximate synchronous playback - usually,
// you would specify a callback function or window when opening the
// device and handle the rest of this when it receives "done" notification
// from the device.
while (!(lpWaveHdr & WHDR_DONE));

// 6. unprepare the WAVEHDR
waveOutUnPrepareHeader(hWaveOut, lpWaveHdt, sizeof(WAVEHDR));

// 7. close the wave output device
waveOutClose(hWaveOut);

// Clean up header
```

```
GlobalUnlock(hWaveHdr);
GlobalFree(hWaveHdr);
```

Note that this code assumes the raw sample data is already present. You could fill it out yourself (through synthesis, etc.), or you could have previously read it from a file (more on that in the next section). Note also that we've skipped error-handling to keep the code readable. The **`CPCMWave`** class, which we'll see soon, will show the correct error-handling.

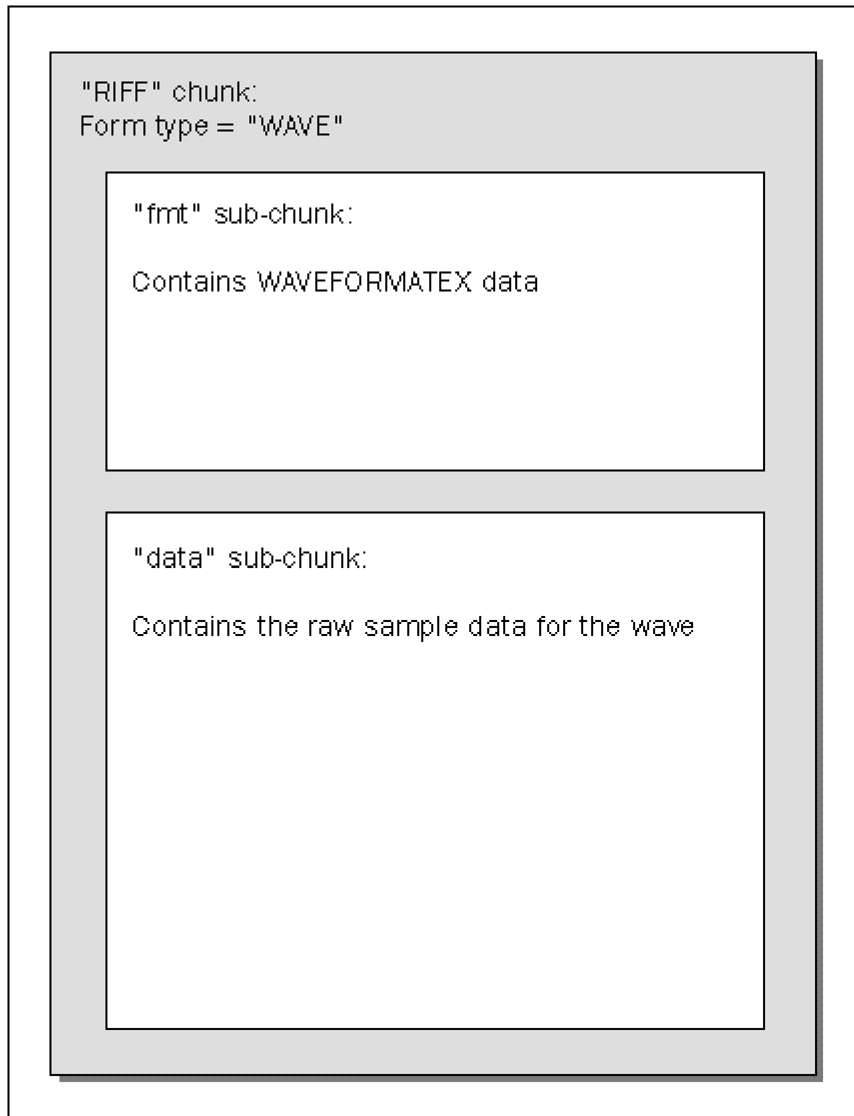
RIFF Files

Usually, when you're writing apps with the low-level APIs, you'll want to work with multimedia data stored in standard-format files that can be read and written by other multimedia apps. Some of the common media files that you see all the time, like **`.wav`** and **`.avi`** files, are written in a standard format called **Resource Interchange File Format** (RIFF).

RIFF files are organized into data blocks, called **chunks**. Chunks are identified by four-character tags, called **FOURCCs**. The FOURCC identifies the type of data stored in the chunk. The low-level Multimedia System API provides functions (whose names start with **`mmio`**) for reading and writing RIFF chunks.

The .wav File Format

A **`.wav`** file is a RIFF file used to store wave audio data. It consists of one big chunk, whose type is **`RIFF`**, that contains at minimum two subchunks: **`fmt`**, which specifies the wave's format, and **`data`**, which holds the raw sample data for the wave. There are other chunks that can show up in a **`.wav`** file (like the **`fact`** subchunk, which is required for all new wave file formats), but they are not required for our purposes here. One of the nice features of the mmio functions is that they automatically locate requested chunks and gracefully ignore any others, so we needn't worry about chunks we're not using. The following diagram shows the basic chunk structure of a basic PCM **`.wav`** file:



Reading and Writing .wav Files

To work with a PCM wave in memory, we'll need to read its format into a **`WAVEFORMATEX struct`** and its sample data into a big buffer of bytes. That means we need to open the main **`RIFF`** chunk and read those two subchunks. Likewise, to save a wave out to a **`.wav`** file, we'll need to first write a main **`RIFF`** chunk, then write the format to a **`fmt`** subchunk and the sample data to a **`data`** subchunk. Here are the steps to read a PCM wave out of a **`.wav`** file:

1. Open the **`.wav`** file with **`mmioOpen()`**.
2. 'Descend' into the main **`RIFF`** chunk with **`mmioDescend()`**.
3. 'Descend' into the **`fmt`** subchunk with **`mmioDescend()`**.
4. Read the format data into a **`WAVEFORMATEX`** (or **`PCMWAVEFORMAT`**) **`struct`**.
5. 'Ascend' out of the **`fmt`** subchunk with **`mmioAscend()`**.
6. 'Descend' into the **`data`** subchunk with **`mmioDescend()`**.

7 Read the raw sample data into your sample buffer.

8 Close the `.wav` file with `mmioClose()`.

Writing a `.wav` works similarly, except `mmioCreateChunk()` is called instead of `mmioDescend()`. Also, when you're writing, to ensure that the created chucks are updated correctly, you must ascend from them all before you close the file.

The mmio functions use a `struct`, called `MMCKINFO`, to describe chunks:

```
typedef struct {
    FOURCC ckid;
    DWORD  cksize;
    FOURCC fccType;
    DWORD  dwDataOffset;
    DWORD  dwFlags;
} MMCKINFO;
```

Member	Meaning
`ckid`	The chunk tag.
`cksize`	The size of the remainder of the chunk (after `cksize`).
`fccType`	The form type for the `RIFF` chunk (not used for subchunks).
`dwDataOffset`	The file offset, relative to beginning of file, to chunk's data.
`dwFlags`	Either zero or `MMIO_DIRTY`, which indicates that the chunk's data has been changed.

Here's a snippet of code that reads wave data from a `.wav` file called `Tada.wav`:

```
HMMIO      hFile;   // Multimedia file handle
MMCKINFO   riffChunk;
MMCKINFO   subChunk;

// 1. open the .WAV file with mmioOpen()
hFile = mmioOpen("tada.wav",    // file name
   NULL,                         // pointer to MMIOINFO struct, not used here
   MMIO_READ|MMIO_ALLOCBUF);     // Open for reading, use a buffer

// 2. "descend" into the main "RIFF" chunk with mmioDescend()

// 2a. set the form type of the chunk to find to "WAVE"
riffChunk.fccType = mmioFOURCC('W','A','V','E');
// mmioFOURCC() is a helper function that builds FOURCCs for you

// 2b. descend into the RIFF chunk
mmioDescend(hFile,    // file handle
   &riffChunk,         // Info about chunk to find
   NULL,               // Parent chunk info - not used here
   MMIO_FINDRIFF);     // Find the "RIFF" chunk

// 3. "descend" into the "fmt " sub-chunk with mmioDescend()

// 3a. Set up the subChunk struct to look for a "fmt " chunk
subChunk.ckid = mmioFOURCC('f','m','t',' ');

// 3b. descend into the sub-chunk
mmioDescend(hFile,    // file handle
   &subChunk,          // Info about chunk to find
   &riffChunk,         // Parent chunk info
   MMIO_FINDCHUNK);    // Find the sub-chunk
```

```
// 4. read the format data into a WAVEFORMATEX (or PCMWAVEFORMAT) struct

// 4a.  Make a buffer to hold the format data
unsigned long fSize = subChunk.cksize;

LPWAVEFORMATEX lpFmt = (LPWAVEFORMATEX) new char[fSize];

//  4b. Read the data into the buffer
mmioRead(hFile, (HPSTR)lpFmt, fSize);


// 5. "ascend" out of the "fmt " sub-chunk with mmioDescend()
mmioAscend(hFile, &subChunk, 0);   // Last param is always 0

// 6. "descend" into the "data" sub-chunk with mmioDescend()

// 6a. Set up the subChunk struct to look for a "data" chunk
subChunk.ckid = mmioFOURCC('d','a','t','a');

// 6b. descend into the sub-chunk
mmioDescend(hFile,   // file handle
   &subChunk,         // Info about chunk to find
   &riffChunk,        // Parent chunk info
   MMIO_FINDCHUNK);   // Find the sub-chunk

// 7. read the raw sample data into your sample buffer
//   We'll assume lpData was already GlobalAlloc'ed and that
//   data_size contains its size in bytes
mmioRead(hFile, (HPSTR)lpData, data_size);

// 8. close the .WAV file with mmioClose()
mmioClose(hFile,0);
```

We've left out error-handling for readability. In the **`CPCMWave`** class, which we'll create next, we'll do it right.

CPCMWave: A Class for Handling Simple PCM Wave Audio

Later in this chapter, we're going to build a project demonstrating low-level manipulation of PCM wave audio. To help tie together the concepts we've seen so far, and to make it easier to use PCM wave audio in our program (and in your own projects), it would be nice to have a C++ class that encapsulates PCM wave audio data and the things one typically does with it. Imagine a PCM wave object that wraps around a block of PCM sample data, and has methods for playing, reading from and writing to **`.wav`** files and retrieving some basic information about the wave. The object should also allow direct manipulation of the raw PCM sample data so we can edit it, graph it, synthesize it, or do anything else we please with it.

At the heart of **`CPCMWave`** will be the three major data elements required in order to manage PCM wave audio data: a wave header (**`WAVEHDR`**), a wave format **`struct`** (**`WAVEFORMATEX`**) and a block of raw sample data. So, we'll start off by defining members in **`CPCMWave`** for these elements:

```
//
// CPCMWave.h - Header for class CPCMWave
//

// Include Microsoft's header for the Windows Multimedia System
// NOTE: You must also link with winmm.lib

#include <mmsystem.h>

class CPCMWave:public CObject
{

protected:

   // Number of *samples*, not bytes, allocated in sample buffer
   unsigned long   m_ulDataSize;

   // Wave buffer
   HANDLE          m_hData;
   LPSTR           m_lpData;

   // Wave header
   HANDLE          m_hWaveHdr;
   LPWAVEHDR       m_lpWaveHdr;

   // Wave format
   WAVEFORMATEX    m_pcmWF;
...
```

Notice that we've included handles and pointers each for the header and raw sample buffer, because we'll be **`GlobalAlloc`**'ing them. The **`WAVEFORMATEX`** can just be a member. We've also stuck in an **`unsigned long`** to keep track of the length of the sample data in the buffer.

Now let's give some thought to how we're going to use a **`CPCMWave`** object so we can lay out some member functions. We'll want to check the wave's format with the playback device, play the wave asynchronously, get its playback position during playback, load it from and save it to RIFF **`.wav`** files, access and alter its raw sample data, erase it and set its basic format information. We'll also need a constructor and destructor. Here are some member declarations to handle those functions:

```
...
   public:
   //
   // Construction / Destruction:
   //
```

```
CPCMWave();
virtual ~CPCMWave();

//
// Wave Attribute Settings:
//

// Size of sample buffer - not settable, must pass size into
// CreateSampleBuffer()
// For our purposes, a "sample" is a block of bytes representing all data
// for all channels for one sample period. GetSampleSize() returns the size,
// in bytes, of one such block. GetSampleSize() is computed from
// GetNumChannels() and GetBitsPerSample().
unsigned long   GetNumSamples();
int             GetSampleSize();

// Playback rate
LONG   GetSampleRate();
void   SetSampleRate(LONG new_rate);

// 1=mono, 2=stereo
int  GetNumChannels();
void SetNumChannels(int chans);

// PCM supports 8 and 16
WORD GetBitsPerSample();
void SetBitsPerSample(WORD bps);

//
// Wave Buffer and Header Access:
//

// Provides access to the vector of raw bytes comprising the wave sample data.
LPSTR  GetBuffer();

// Provides access to the wave header
LPWAVEHDR  GetHeader();

// Creates a new sample buffer, discarding any old data first
LPSTR CreateSampleBuffer(LONG num_samples);

// Disposes of (erases) the current sample buffer
void FreeSampleBuffer();

// Indicates whether or not buffer is initialized
BOOL IsBufferReady();


//
// Wave Async Playback Control:
//

// Returns TRUE if current wave data is playable on the WAVE_MAPPER device
BOOL IsPlayable();

// Starts async playback on the WAVE_MAPPER device
UINT Play(CWnd* pAppWnd = NULL);

// Finishes async playback (called from callback function or window message on
// MM_WOM_DONE)
void StopPlaying();

// Returns TRUE if wave is currently playing
BOOL IsPlaying();

// Returns the position of the current playback point (meant for use only
// during playback, ie from a timer callback, etc.)
unsigned long GetPlayPosition(UINT units = TIME_SAMPLES);

//
// Wave RIFF File I/O:
```

```
   //
   BOOL SaveWAV(const CString& fname);
   BOOL LoadWAV(const CString& fname);
...
```

We'll need to add a wave output device handle data member so that we can keep track of the playback device during asynchronous playback:

```
...
   // Wave playback device - only used during playback
   HWAVEOUT   m_hWaveOut;
...
```

And a flag to let us know quickly whether or not the wave is currently playing:

```
...
   // Flag to indicate whether wave is currently playing or not
   BOOL      m_bPlaying;
};
```

That's it for the interface to **CPCMWave**. Before we leave the header, though, let's do a few things. First, let's add some **struct** definitions to help decode the stereo PCM data formats in the sample buffer, so we won't have to remember the byte ordering:

```
// Typedefs to make it easier to interpret sample buffer.
// NOTE: Intel only!  PCM data is little-endian.  Other platforms may need
// to convert.

// Also:
// 8-bit mono is just 1 BYTE per sample
// 16-bit mono is just 1 short per sample (see Intel note above!)

// 8-bit values:      Min = 0      Max = 255      Zero = 128
// 16-bit values:   Min = -32768   Max = 32767      Zero = 0

typedef struct {
   BYTE   left;
   BYTE   right;
} PCMSample_8bitStereo;


typedef struct {
   short   left;
   short   right;
} PCMSample_16bitStereo;
```

Second, some of the member functions are so simple that we can just add them to the header as **inline**s after the class definition:

```
//
// Inline Member Functions:
//

inline CPCMWave::~CPCMWave()
{
   FreeSampleBuffer();
}

inline unsigned long CPCMWave::GetNumSamples()
{
   return m_ulDataSize;
}

inline int CPCMWave::GetSampleSize()
{
```

```
   return m_pcmWF.nBlockAlign;
}

inline LONG CPCMWave::GetSampleRate()
{
   return m_pcmWF.nSamplesPerSec;
}

inline void CPCMWave::SetSampleRate(LONG new_rate)
{
   m_pcmWF.nSamplesPerSec = new_rate;
   // Take care of nAvgBytesPerSec so we don't have to worry about it
   m_pcmWF.nAvgBytesPerSec = new_rate * m_pcmWF.nBlockAlign;
}

inline int CPCMWave::GetNumChannels()
{
   return m_pcmWF.nChannels;
}

inline void CPCMWave::SetNumChannels(int chans)
{
   m_pcmWF.nChannels = chans;
   // Take care of nBlockAlign so we don't have to worry about it
   m_pcmWF.nBlockAlign = (chans * m_pcmWF.wBitsPerSample)/8;
   // ...and fix up nAvgBytesPerSec...
   m_pcmWF.nAvgBytesPerSec = m_pcmWF.nSamplesPerSec*m_pcmWF.nBlockAlign;
}

inline WORD CPCMWave::GetBitsPerSample()
{
   return m_pcmWF.wBitsPerSample;
}

inline void CPCMWave::SetBitsPerSample(WORD bps)
{
   m_pcmWF.wBitsPerSample = bps;
   // Take care of nBlockAlign so we don't have to worry about it
   m_pcmWF.nBlockAlign = (m_pcmWF.nChannels * bps)/8;
   // ...and fix up nAvgBytesPerSec...
   m_pcmWF.nAvgBytesPerSec = m_pcmWF.nSamplesPerSec*m_pcmWF.nBlockAlign;
}

inline HPSTR CPCMWave::GetBuffer()
{
   return m_lpData;
}

inline LPWAVEHDR CPCMWave::GetHeader()
{
   return m_lpWaveHdr;
}


inline BOOL CPCMWave::IsPlaying()
{
   return m_bPlaying;
}


inline BOOL CPCMWave::IsBufferReady()
{
   // If any of these are NULL, buffer is NOT ready
   return (m_hData && m_lpData && m_hWaveHdr && m_lpWaveHdr);
}
```

Now let's move on to the real implementation of the **CPCMWave** class. To start, let's get the constructor out of the way:

```
CPCMWave::CPCMWave()
{

   m_bPlaying = FALSE;

   m_hData = NULL;
   m_lpData = NULL;

   m_hWaveHdr = NULL;
   m_lpWaveHdr = NULL;


   // 1 sec at 11,025 samps/sec
   m_ulDataSize = 11025L;

   //
   // Set up wave format structure
   //
   // Default: 8-bit mono
   //

   m_pcmWF.wFormatTag = WAVE_FORMAT_PCM;
   m_pcmWF.nChannels = 1;
   m_pcmWF.nSamplesPerSec = 11025L;
   m_pcmWF.nAvgBytesPerSec = 11025L;
   m_pcmWF.nBlockAlign = 1;
   m_pcmWF.wBitsPerSample = 8;
   m_pcmWF.cbSize = 0;
}
```

We're just initializing things here, and setting up the wave format to an arbitrary (but functional) default.

Next, let's get **CreateSampleBuffer()** and **FreeSampleBuffer()** in there:

```
LPSTR CPCMWave :: CreateSampleBuffer(long num_samples)
{
   m_ulDataSize = num_samples;

   //
   // Allocate sample buffer of user-supplied length
   //

   m_hData = GlobalAlloc(GMEM_MOVEABLE|GMEM_SHARE,
                                        m_ulDataSize * m_pcmWF.nBlockAlign);
   if (!m_hData)                  // If we didn't get the memory
      return NULL;                // exit

   m_lpData = (LPSTR)GlobalLock(m_hData);
   if (!m_lpData)                  // If we didn't get the lock
   {
      GlobalFree(m_hData);        // free the memory
      m_hData = NULL;              // reset the handle
      return NULL;                 // and exit
   }

   // Allocate Wave header
   m_hWaveHdr = GlobalAlloc(GMEM_MOVEABLE|GMEM_SHARE, (DWORD)sizeof(WAVEHDR));
   if (!m_hWaveHdr)                // If we didn't get the memory
   {
      GlobalUnlock(m_hData);      // Unlock the sample buffer
      GlobalFree(m_hData);        // free the memory
      m_lpData = NULL;
      m_hData = NULL;
      return NULL;                 // and exit
   }
   m_lpWaveHdr = (LPWAVEHDR)GlobalLock(m_hWaveHdr);
   if (!m_lpWaveHdr)               // If we didn't get the lock
   {                               // Undo everything
      GlobalFree(m_hWaveHdr);
```

```
        GlobalUnlock(m_hData);
        GlobalFree(m_hData);
        m_hWaveHdr = NULL;
        m_lpData = NULL;
        m_hData = NULL;
        return NULL;                 // and exit
    }

    // Set up the wave header data
    m_lpWaveHdr->lpData = m_lpData;
    m_lpWaveHdr->dwBufferLength = m_ulDataSize*m_pcmWF.nBlockAlign;
    m_lpWaveHdr->dwFlags = 0L;

    return m_lpData;
    }

void CPCMWave :: FreeSampleBuffer()
{
    // Ensure handles are valid
    ASSERT(m_hData);
    ASSERT(m_hWaveHdr);

    //De-allocate memory
    GlobalUnlock(m_hData);
    GlobalFree(m_hData);
    GlobalUnlock(m_hWaveHdr);
    GlobalFree(m_hWaveHdr);

    // Reset handles and pointers
    m_hData = NULL;
    m_lpData = NULL;
    m_hWaveHdr = NULL;
    m_lpWaveHdr = NULL;
    m_ulDataSize = 0;
}
```

That takes care of buffer and header memory management. Now let's apply what we learned earlier about **.wav** file RIFF I/O and write the **SaveWAV()** and **LoadWAV()** functions:

```
BOOL CPCMWave::SaveWAV(const CString& fname)
{
    //=======================================================
    //
    //  RIFF File I/O:

    HMMIO      hmmio;          // Multimedia System file handle
    MMCKINFO   mmckinfo;       // Chunk info structure
    MMCKINFO   mmckinfoSubChunk;    // Chunk info structure

    //
    // Open the .WAV file for writing
    //

    if (!(hmmio = mmioOpen((LPSTR)(const char*)fname, NULL,
            MMIO_WRITE | MMIO_CREATE | MMIO_ALLOCBUF )))
        return FALSE;

    //
    // Create a RIFF chunk whose form type is 'WAVE'
    //

    mmckinfo.fccType = mmioFOURCC('W','A','V','E');

    if (mmioCreateChunk(hmmio, &mmckinfo, MMIO_CREATERIFF) != 0)
    {
        mmioClose(hmmio,0);
        return FALSE;
    }
```

```
//
// Create a subchunk whose ID is 'fmt '
//

mmckinfoSubChunk.ckid = mmioFOURCC('f','m','t',' ');

if (mmioCreateChunk(hmmio, &mmckinfoSubChunk, 0) != 0)
{
   mmioClose(hmmio,0);
   return FALSE;
}

//
// Write out format info
//

if (mmioWrite(hmmio,(HPSTR)&m_pcmWF, sizeof(PCMWAVEFORMAT)) == -1)
{
   mmioClose(hmmio,0);
   return FALSE;
}

//
// Ascend from subchunk
//

if (mmioAscend(hmmio,&mmckinfoSubChunk,0) != 0)
{
   mmioClose(hmmio,0);
   return FALSE;
}

//
// Create a subchunk whose ID is 'data'
//

mmckinfoSubChunk.ckid = mmioFOURCC('d','a','t','a');

if (mmioCreateChunk(hmmio, &mmckinfoSubChunk, 0) != 0)
{
   mmioClose(hmmio,0);
   return FALSE;
}

//
// Write out wave data
//

if (mmioWrite(hmmio,(HPSTR)m_lpData,m_ulDataSize*m_pcmWF.nBlockAlign)==-1)
{
   mmioClose(hmmio,0);
   return FALSE;
}

//
// Ascend from subchunk
//
if (mmioAscend(hmmio,&mmckinfoSubChunk,0) != 0)
{
   mmioClose(hmmio,0);
   return FALSE;
}

//
// Ascend from  RIFF chunk
//
if (mmioAscend(hmmio,&mmckinfo,0) != 0)
{
   mmioClose(hmmio,0);
   return FALSE;
}
```

```
   //
   // Close file
   //

   mmioClose(hmmio,0);

   //  End RIFF File I/O
   //=======================================================

   return TRUE;}

BOOL CPCMWave :: LoadWAV(const CString& fname)
{
   // Ensure there is no old data
   ASSERT(!IsBufferReady());

   //=======================================================
   //
   //  RIFF File I/O:

   HMMIO      hmmio;          // Multimedia System file handle
   MMCKINFO   mmckinfo;       // Chunk info structure
   MMCKINFO   mmckinfoSubChunk;   // Chunk info structure

   //
   // Open the .WAV file for reading
   //

   if (!(hmmio = mmioOpen((LPSTR)(const char*)fname, NULL,
            MMIO_READ | MMIO_ALLOCBUF )))
      return FALSE;

   //
   // Find a RIFF chunk whose form type is 'WAVE'
   //

   mmckinfo.fccType = mmioFOURCC('W','A','V','E');

   if (mmioDescend(hmmio, &mmckinfo, NULL, MMIO_FINDRIFF))
   {
      mmioClose(hmmio,0);
      return FALSE;
   }

   //
   // Find a subchunk whose ID is 'fmt '
   //

   mmckinfoSubChunk.ckid = mmioFOURCC('f','m','t',' ');

   if (mmioDescend(hmmio, &mmckinfoSubChunk, &mmckinfo, MMIO_FINDCHUNK))
   {
      mmioClose(hmmio,0);
      return FALSE;
   }

   long wFSize = mmckinfoSubChunk.cksize;

   WAVEFORMATEX *pWaveFormat = (WAVEFORMATEX*) new char[wFSize];

   if (!pWaveFormat)
   {
      mmioClose(hmmio,0);
      return FALSE;
   }

   //
   // Read format info
   //
```

```
if (mmioRead(hmmio,(HPSTR)pWaveFormat, wFSize) != wFSize)
{
   delete pWaveFormat;
   mmioClose(hmmio,0);
   return FALSE;
}

// Make sure this is a plain vanilla PCM wave
if (pWaveFormat->wFormatTag != WAVE_FORMAT_PCM)
{
   delete pWaveFormat;
   mmioClose(hmmio,0);
   return FALSE;
}
SetSampleRate(pWaveFormat->nSamplesPerSec);
SetNumChannels(pWaveFormat->nChannels);
SetBitsPerSample(pWaveFormat->wBitsPerSample);

//
// Ascend from subchunk
//
if (mmioAscend(hmmio,&mmckinfoSubChunk,0) != 0)
{
   delete pWaveFormat;
   mmioClose(hmmio,0);
   return FALSE;
}

//
// Find a subchunk whose ID is 'data'
//

mmckinfoSubChunk.ckid = mmioFOURCC('d','a','t','a');

if (mmioDescend(hmmio, &mmckinfoSubChunk, &mmckinfo, MMIO_FINDCHUNK))
{
   delete pWaveFormat;
   mmioClose(hmmio,0);
   return FALSE;
}

//
// Read in wave data
//

// Actual number of raw bytes in wave data
long dSize = mmckinfoSubChunk.cksize;

if (!CreateSampleBuffer(dSize/pWaveFormat->nBlockAlign))
{
   delete pWaveFormat;
   mmioClose(hmmio,0);
   return FALSE;
}

if (mmioRead(hmmio,(LPSTR)m_lpData,dSize) != dSize)
{
   delete pWaveFormat;
   FreeSampleBuffer();
   mmioClose(hmmio,0);
   return FALSE;
}

//
// Close file
//

mmioClose(hmmio,0);
delete pWaveFormat;

//  End RIFF File I/O
```

```
    //=======================================================

    return TRUE;
}
```

Next, we'll take advantage of a feature of the **`waveOutOpen()`** function to create the **`IsPlayable()`** member function:

```
// The Play feature in CPCMWave uses the MCI WAVE_MAPPER device. This function
// determines if the WAVE_MAPPER device is available and if it supports the
// format of the current wave.

BOOL CPCMWave::IsPlayable()
{
   UINT      retval;

   //
   // Get wave output device caps
   //

   retval = IsFormatSupported(&m_pcmWF,WAVE_MAPPER);
   switch (retval)
   {
      case 0:
        // Supports wave format
        break;

      case WAVERR_BADFORMAT:
        // Format not supported by your MCI wave-audio playback device
        return FALSE;

      default:
         // MCI PCM wave-audio playback device unavailable
         return FALSE;
   }
   return TRUE;
}
```

This little helper function is useful by itself, so it's broken out:

```
// Asks the given wave device if it supports the format of the current wave
UINT IsFormatSupported(LPWAVEFORMATEX lpPCMWF, UINT DevID)
{
   return (waveOutOpen(NULL, DevID, lpPCMWF, NULL, NULL, WAVE_FORMAT_QUERY));
}
```

Now let's make the wave play. Unlike in our earlier example code snippet, we'll make the wave play asynchronously if possible, which means that other things can happen in the program while the wave plays in the background. To accomplish this, we need to pass either a window or a function into the **`waveOutOpen()`** call and tell it to notify the window or function when playback is complete so we can clean up. We'll make the **`Play()`** function take an optional parameter, a **`CWnd`** pointer, for a callback window. If the pointer is left **`NULL`**, we'll use a callback function to handle the cleanup automatically. Here's the callback function, which we'll put at the top of our module:

```
// Callback function used to accomplish async playback
void CALLBACK myWaveOutProc(HWAVEOUT hwo, UINT uMsg, DWORD dwInstance,
    DWORD dwParam1, DWORD dwParam2)
{
   if (uMsg == WOM_DONE)
   {
      // Get the PCMWave object out of the instance param
      PCMWave* wave = (PCMWave*)dwInstance;

      // Tell it that playing is done
```

```
      wave->StopPlaying();
   }
}
```

To have a **CWnd** in your app catch the callback, you need to add a **WindowProc()** override to it and check for the message **MM_WOM_DONE**, since ClassWizard doesn't seem to let you catch it. For example, you might add something like this:

```
LRESULT CWaveView::WindowProc(UINT message, WPARAM wParam, LPARAM lParam)
{
   // Clean up wave playback
   if (message == MM_WOM_DONE)
      the_wave.StopPlaying();

   return CScrollView::WindowProc(message, wParam, lParam);
}
```

With that said, here's the **Play()** function:

```
UINT CPCMWave :: Play(CWnd* pAppWnd)
{
   // Can't play if no sample buffer has yet been created
   if (!IsBufferReady())
      return PCMWAVE_ERR_NO_WAVE_BUFFER;

   // Open wave-audio device for playback

   // If window is given, send MM_WOM_DONE to it, else use our callback
   // function to terminate playback
   if (pAppWnd)
   {
      // Use window callback (sends MM_WOM_DONE to window when wave stops
      // playing - donePlaying() must be called then)
      if (waveOutOpen(&m_hWaveOut,WAVE_MAPPER,&m_pcmWF,
         (DWORD)pAppWnd->GetSafeHwnd(), 0L,CALLBACK_WINDOW))
         return PCMWAVE_ERR_OPEN_DEV;     // Error opening wave-audio device

   }
   else
   {
      // Use callback function
      if (waveOutOpen(&m_hWaveOut,WAVE_MAPPER,&m_pcmWF,
         (DWORD)myWaveOutProc,(DWORD)this,CALLBACK_FUNCTION))
         return PCMWAVE_ERR_OPEN_DEV;
   }

   waveOutPrepareHeader(m_hWaveOut,m_lpWaveHdr,sizeof(WAVEHDR));

   //
   // Send wave to wave device for playback
   //
   m_bPlaying = TRUE;

   if (waveOutWrite(m_hWaveOut,m_lpWaveHdr,sizeof(WAVEHDR)))
   {
      // Output error
      waveOutUnprepareHeader(m_hWaveOut, m_lpWaveHdr,sizeof(WAVEHDR));
      waveOutClose(m_hWaveOut);
      m_bPlaying = FALSE;

      // Unable to send wave data to MCI wave-audio playback device
      return PCMWAVE_ERR_WRITE_DATA;
   }
   return 0;
}
```

We've been slinging around some result codes that need to get into **PCMWave.h**:

```
// Error results
#define PCMWAVE_ERR_OPEN_DEV      1
#define PCMWAVE_ERR_WRITE_DATA     2
#define PCMWAVE_ERR_NO_WAVE_BUFFER   3
```

Finally, let's define the **PlayPosition()** function, which can tell us where we are in a wave during playback:

```
unsigned long CPCMWave::PlayPosition(UINT units)
{
   MMTIME mmt;

   mmt.wType = units;

   if (waveOutGetPosition(m_hWaveOut, &mmt, sizeof(MMTIME))!=MMSYSERR_NOERROR)
      return 0;       // Failed to get position

   switch (units)
   {
      // return position in bytes
      case TIME_BYTES:
         return mmt.u.cb;
         break;

      // return position in samples
      case TIME_SAMPLES:
         return mmt.u.sample;
         break;

      // return position in milliseconds
      case TIME_MS:
         return mmt.u.ms;
         break;

      default:
         break;
   }

   // PCMWave only supports the above units
   ASSERT(FALSE);
   return 0;
}
```

Project: WaveScope

Now that we have an easy-to-use class for PCM wave audio, let's take it out for spin. In this project, we'll build an MDI app that opens and graphically displays **`.wav`** files. It will also play the waves, and report some basic statistics on them (like sample rate, number of channels, bitwidth, etc.). Most of the hard stuff is already done by **`CPCMWave`**, so this will be a surprisingly easy program to build.

Step 1

Use AppWizard to create a new project entitled WaveScope. Create the project using the following AppWizard options:

AppWizard Step 1: use the default setting (multiple documents).

AppWizard Step 2: use the default setting (no database support).

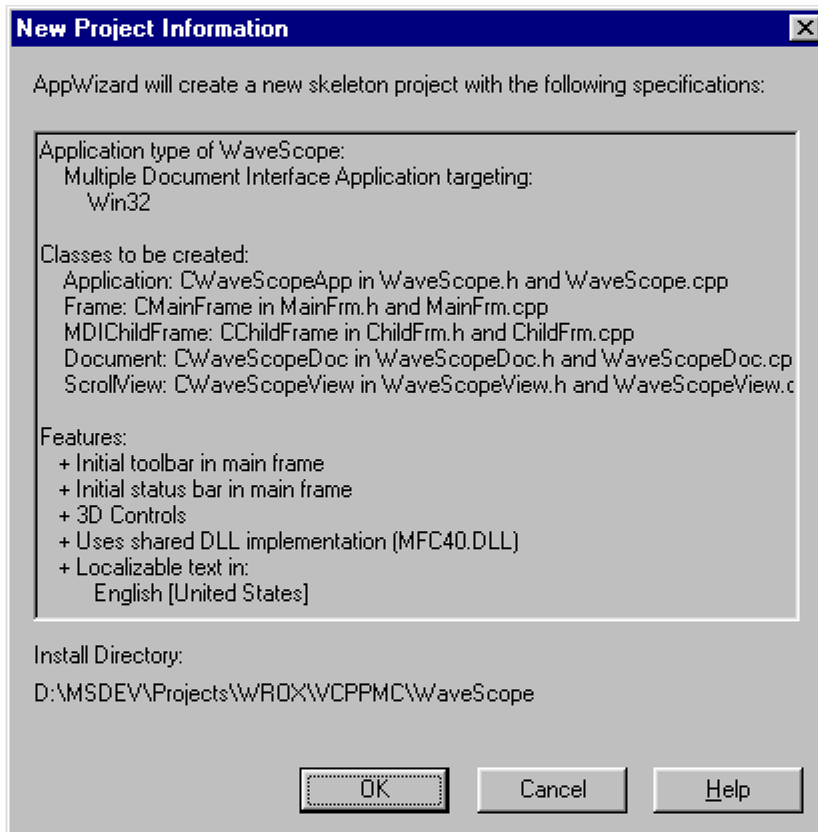
AppWizard Step 3: use the default settings (no OLE support).

AppWizard Step 4: use the default settings, except turn off printing and print preview support (we won't be using it).

AppWizard Step 5: use the default settings.

AppWizard Step 6: use the default settings, except make **`CWaveScopeView`** descend from **`CScrollView`**.

If everything worked all right, you should wind up with a New Project Information dialog that looks like this:



Step 2

Add the **CPCMWave** source files that we created earlier (**PCMWave.h**, **PCMWave.cpp**) to the project. Include **PCMWave.h** at the top of **WaveScopeDoc.h**:

```
// WaveScopeDoc.h : interface of the CWaveScopeDoc class
//
/////////////////////////////////////////////////////////////////////////////

#include "PCMWave.h"

class CWaveScopeDoc : public CDocument
```

Add a **CPCMWave** member to **CWaveScopeDoc**:

```
// Attributes
public:

   CPCMWave   the_wave;
```

Step 3

Next, we need to get the **CPCMWave** loaded out of a **.wav** file. In **CWaveScopeDoc**, add an **OnOpenDocument()** handler and edit it to look like this:

```
BOOL CWaveScopeDoc::OnOpenDocument(LPCTSTR lpszPathName)
{

   // Attempt to open the given file path as a .WAV
   if ((!CDocument::OnOpenDocument(lpszPathName))||
                                         (!m_The_Wave.loadWAV(lpszPathName))
   {
      AfxMessageBox(Error: Unable to open the specified file.");

      return FALSE;
   }


   // Set modified flag to force app through our SaveModified() handler when
   // trying to close document
   SetModifiedFlag(TRUE);

   return TRUE;
}
```

The first part hands the file path to the wave object to open as a **.wav** file. The second part sets the modified flag, which is a trick to force the app through a **SaveModified()** handler we'll write later when trying close the document. We'll need to do that because we don't want the document to close on us while an asynchronous wave playback is occurring.

We also want to make sure that a new document isn't automatically created on startup, so go into **CWaveScopeApp::InitInstance()**, find the command line processing code (near the end), and make the following changes:

```
   // Parse command line for standard shell commands, DDE, file open
   CCommandLineInfo cmdInfo;
   ParseCommandLine(cmdInfo);

   // Dispatch commands specified on the command line
   if (cmdInfo.m_nShellCommand != CCommandLineInfo::FileNew)
      if (!ProcessShellCommand(cmdInfo))
         return FALSE;
```

Step 4

We're going to draw the actual waveform of the **CPCMWave**'s data on the view. To keep things simple, we'll squeeze every wave to fit in the same space, say: 4096 x 256. So, we need to set up the scrolling extents to handle this. Go to **CWaveScopeView::OnInitialUpdate()** and add make it look like this:

```
void CWaveScopeView::OnInitialUpdate()
{
   CScrollView::OnInitialUpdate();
   CSize sizeTotal;

   sizeTotal.cx = 4096;
   sizeTotal.cy = 256;

   SetScrollSizes(MM_TEXT, sizeTotal);
}
```

Step 5

OK, this next bit is a little involved. Now we're going to paint the waveform on the view. We'll be stepping through the raw wave data and plotting the values. Remember that there are four kinds of waves we might have to handle: 8-bit mono, 8-bit stereo, 16-bit mono and 16-bit stereo. We'll draw mono waves in black, and we'll draw stereo waves in two colors: red for the right channel, blue for the left. Also, we'll have to rescale 16-bit data to fit our 256-unit-high view. To start off, we can code the **OnDraw()** member of **CWaveScopeView** as follows:

```
void CWaveScopeView::OnDraw(CDC* pDC)
{
   CWaveScopeDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);

   // Don't do anything if sample buffer not initialized
   if (!pDoc->m_TheWave.IsBufferReady())
      return;

   // Calculate an increment value for stepping through buffer that will
   // "shorten" wave to fit in 4096 pixels, if necessary
   unsigned long SampleCount = pDoc->m_TheWave.GetNumSamples();
   int inc = SampleCount / 4096;
   if (inc == 0)
      inc = 1;

   switch (pDoc->m_TheWave.GetNumChannels())
   {
   case 1:      // Mono
      switch (pDoc->m_TheWave.GetBitsPerSample())
      {
        case 8:         // 8 bit
        {
           // Draw 8-bit mono wave
           BYTE* sample = (BYTE*)pDoc->m_TheWave.GetBuffer(); // Get the sample
           Draw8BitMono(sample, SampleCount, inc, pDC);       // and draw it
        }
        break;

        case 16:        // 16 bit
        {
           // Draw 16-bit mono wave
           short* sample = (short*)pDoc->m_TheWave.GetBuffer(); // Get sample
           Draw16BitMono(sample, SampleCount, inc, pDC);        // and draw it
        }
        break;

        default:
           // Must be 8 or 16 bits!
           ASSERT(FALSE);
```

```
            break;
         };
         break;

      case 2:   // Stereo
         switch (pDoc->m_TheWave.GetBitsPerSample())
         {
           case 8:        // 8 bit
           {
              // Draw 8-bit stereo wave
              PCMSample_8bitStereo* sample =
                 (PCMSample_8bitStereo*)pDoc->m_TheWave.GetBuffer(); // Get sample
              Draw8BitStereo(sample, SampleCount, inc, pDC);         // and draw it

           }
           break;

           case 16:       // 16 bit
           {
              // Draw 16-bit stereo wave
              PCMSample_16bitStereo* sample =
                (PCMSample_16bitStereo*)pDoc->m_TheWave.GetBuffer(); // Get sample
              Draw16BitStereo(sample, SampleCount, inc, pDC);        // and draw it

           }
           break;

           default:
              // Must be 8 or 16 bits!
              ASSERT(FALSE);
              break;
         };
         break;

      default:
         // Must be mono or stereo!
         ASSERT(FALSE);
         break;
      };
}
```

The outer switch statement determines whether we're dealing with a mono or stereo data, and in each case a further switch statement determines whether we have 8 bit or 16 bit samples. We call one of four functions to plot the wave data. Each function has four parameters: an array of samples for the wave, the count of the number of samples, the increment through the samples to fit the plot into the width of the window, and the pointer to the DC. The type of the first parameter is different in each case to accommodate the applicable sample data. The code for the function to draw the 8-bit mono wave is:

```
void CWaveScopeView::Draw8BitMono(const BYTE* sample,
   const unsigned long& count, const int& inc, CDC* pDC)
{
   // Draw 8-bit mono wave
   pDC->MoveTo(0,sample[0]);          // Move to the first sample point

   unsigned long i = 1;
   while (i < count)
   {
      pDC->LineTo(i/inc,sample[i]);   // Draw a line to the next sample point
      i += inc;
   }
}
```

Next, the code for the 16-bit mono function. This will be similar to the previous code, except that we'll be scaling the data to fit in the 256-unit-high view:

```
void CWaveScopeView::Draw16BitMono(const short* sample,
```

```
   const unsigned long& count, const int& inc, CDC* pDC)
{
   // Draw 16 bit mono wave
   unsigned int val = (sample[0] + 32768)/256;

   pDC->MoveTo(0,val);                  // Move to the first sample point

   unsigned long i = 1;
   while (i < count)
   {
      // Convert sample value to 128-offset 8-bit for plotting
      val = (sample[i] + 32768)/256;
      pDC->LineTo(i/inc,val);          // Draw a line to the next sample point
      i += inc;
   }
}
```

The stereo versions are also very similar, except that we'll plot twice: once in blue for the left channel, then again in red for the right. Notice the use of the convenience **struct**s provided in **PCMWave.h** to aid in the interpretation of the stereo sample data. Here's the 8-bit stereo code:

```
void CWaveScopeView::Draw8BitStereo(const PCMSample_8bitStereo* sample,
   const unsigned long& count, const int& inc, CDC* pDC)
{
   // Draw 8 bit stereo wave
   CPen leftPen (PS_SOLID, 1, RGB(0,0,255));          // Create blue pen
   CPen rightPen (PS_SOLID, 1, RGB(255,0,0));         // Create red pen
   CPen* oldPen;

   // Draw the left channel
   oldPen = pDC->SelectObject(&leftPen);          // Select the left pen
   pDC->MoveTo(0,sample[0].left);                 // Move to first sample point

   unsigned long i = 1;
   while (i < count)
   {
      pDC->LineTo(i/inc,sample[i].left);      // Draw line to next sample point
      i += inc;
   }

   // Draw the right channel
   pDC->SelectObject(&rightPen);

   pDC->MoveTo(0,sample[0].right);            // Move to first sample point

   i = 1;
   while (i < count)
   {
      pDC->LineTo(i/inc,sample[i].right);     // Draw line to next sample point
      i += inc;
   }

   pDC->SelectObject(oldPen);                 // Restore the old pen
}
```

In the 16-bit stereo function, we just need to add the scaling of the sample amplitudes:

```
void CWaveScopeView::Draw16BitStereo(const PCMSample_16bitStereo* sample,
                          const unsigned long& count, const int& inc, CDC* pDC)
{
   // Draw 16 bit stereo wave
   CPen leftPen (PS_SOLID, 1, RGB(0,0,255));          // Create blue pen
   CPen rightPen (PS_SOLID, 1, RGB(255,0,0));         // Create red pen
   CPen* oldPen;

   // Draw the left channel
   oldPen = pDC->SelectObject(&leftPen);      // Select the left pen
   // Convert sample value to 128-offset 8-bit for plotting
```

```
    unsigned int val = (sample[0].left + 32768)/256;
    pDC->MoveTo(0,val);                        // Move to first sample point

    unsigned long i = 1;
    while (i < count)
    {
       // Convert sample value to 128-offset 8-bit for plotting
       val = (sample[i].left + 32768)/256;
       pDC->LineTo(i/inc,val);                 // Draw line to next sample point
       i += inc;
    }

    // Draw the right channel
    pDC->SelectObject(&rightPen);              // Select the right pen

    // Convert sample value to 128-offset 8-bit for plotting
    val = (sample[0].right + 32768)/256;
    pDC->MoveTo(0,val);                        // Move to first sample point

    i = 1;
    while (i < count)
    {
       // Convert sample value to 128-offset 8-bit for plotting
       val = (sample[i].right + 32768)/256;
       pDC->LineTo(i/inc, val);                // Draw line to next sample poin
       i += inc;
    }

    pDC->SelectObject(oldPen);                 // Restore the old pen
 }
```

Now you should be able to build the application and run it. You'll be able to open **.wav** files and see their waveforms.

Step 6

Edit the menu **IDR_WAVESCTYPE** and delete the Edit menu. Remove the New, Save and Save As... items from the File menu. Insert a new menu, called Wave, after the File menu. Put two items in it: Info... and Play. Also, open the menu **IDR_MAINFRAME** and delete the New item from the File menu. Open the toolbar **IDR_MAINFRAME** and remove everything except the File Open and Help tools, and add tools for Wave Play and Wave Info.

Add handlers to **CWaveScopeView** for **ID_WAVE_PLAY COMMAND** and **COMMANDUI**, and put the following code there:

```
void CWaveScopeView::OnWavePlay()
{
   GetDocument()->m_TheWave.Play();
}

void CWaveScopeView::OnUpdateWavePlay(CCmdUI* pCmdUI)
{
   // Disable Play button when a wave is playing
   pCmdUI->Enable(!GetDocument()->m_TheWave.IsPlaying());
}
```

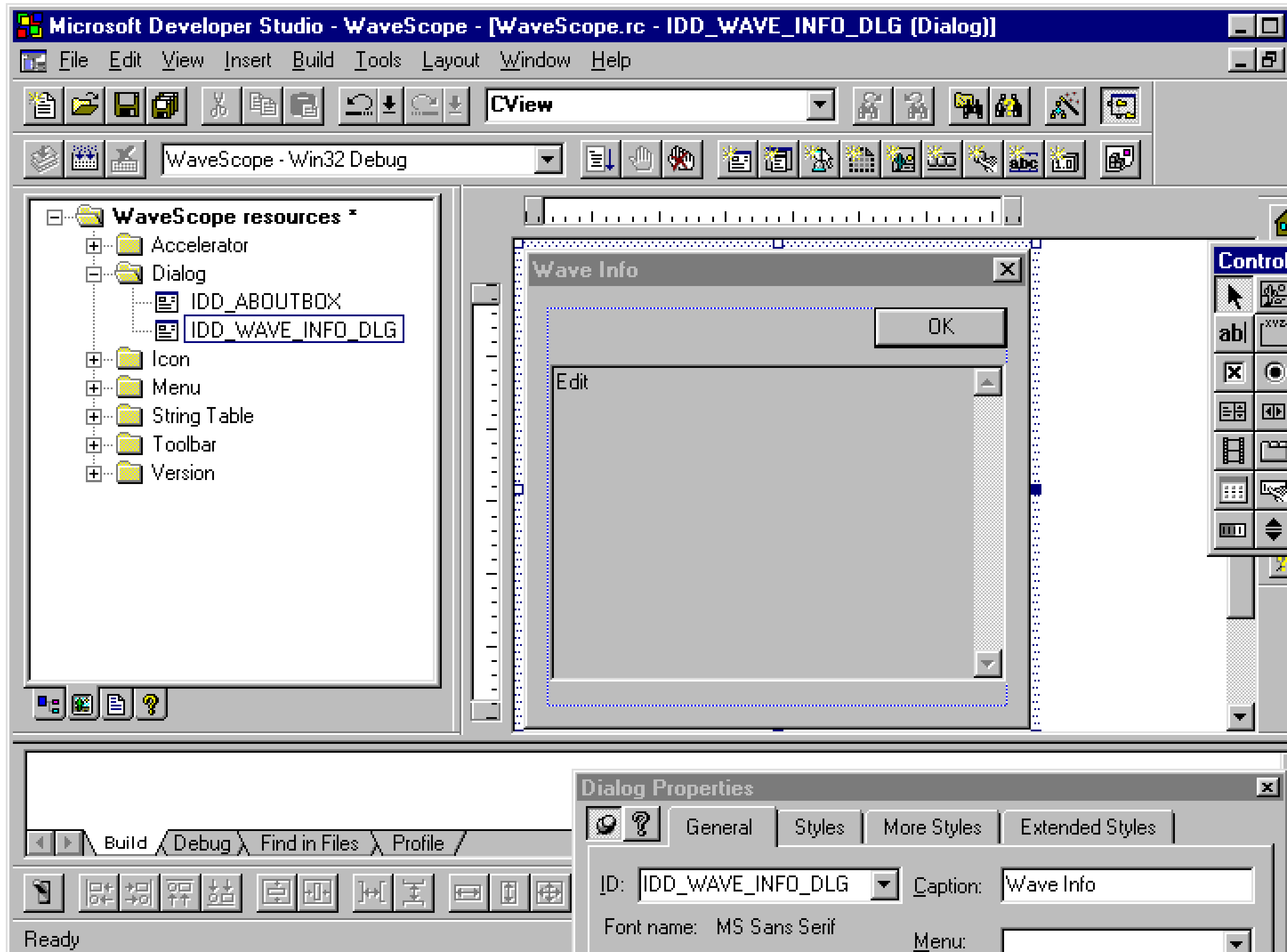
We have to make sure that the document can't be closed during playback, so add a **SaveModified()** handler to **CWaveScopeDoc** and put this code in it:

```
BOOL CWaveScopeDoc::SaveModified()
{
   // Can't close if a wave is currently playing
   return !m_TheWave.IsPlaying();
}
```

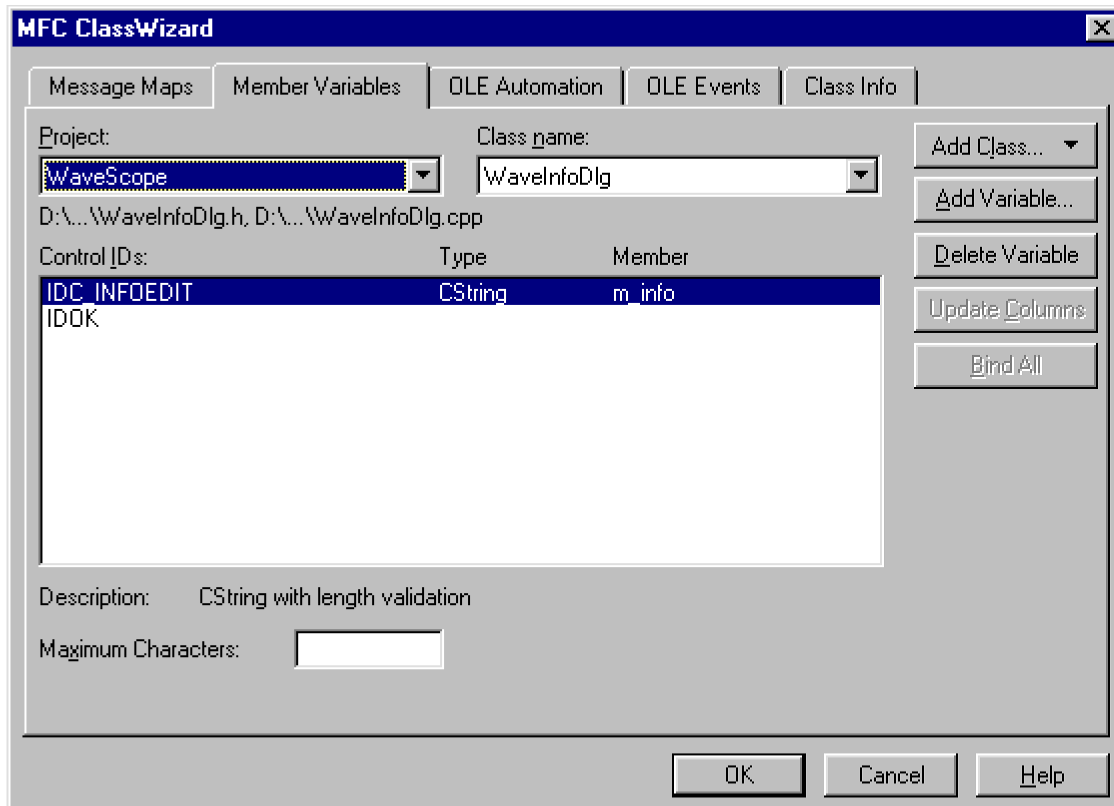
Now you should be able to build the app and play waves.

Step 7

The last feature we're going to add to WaveScope is a simple dialog that reports some statistics on a wave. Create a new dialog box. Set its caption to Wave Info and delete the Cancel button (we won't need one). Put an edit control on it and make it fill most of the dialog. Set the edit control's Multi-line, Vertical Scroll and Read-only styles. It should wind up looking about like this:



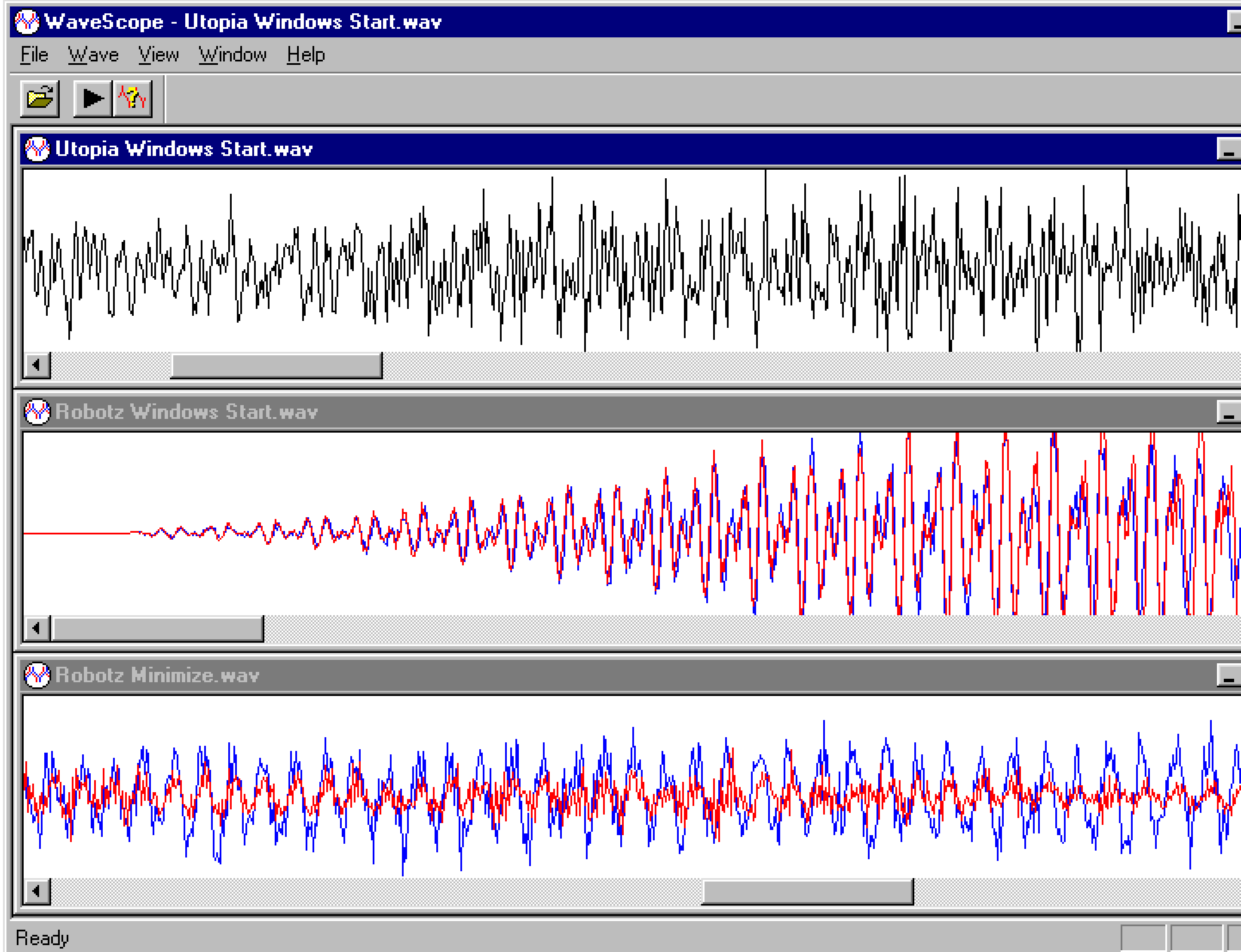
Add a member variable for the value of the edit control. Call it **`m_info`**:



Now, in **CWaveScopeView**, add a handler for **ID_WAVE_INFO COMMAND**, and put the following code in it:

```
void CWaveScopeView::OnWaveInfo()
{
   WaveInfoDlg dlg;
   char buf[30];

   dlg.m_info = "Wave sample rate = ";
   dlg.m_info += itoa(GetDocument()->m_TheWave.GetSampleRate(),buf,10);
   dlg.m_info += " Hz\r\n";

   dlg.m_info += "Wave bitwidth = ";
   dlg.m_info += itoa(GetDocument()->m_TheWave.GetBitsPerSample(),buf,10);
   dlg.m_info += " bits\r\n";

   dlg.m_info += "Wave channels = ";
   dlg.m_info += itoa(GetDocument()->m_TheWave.GetNumChannels(),buf,10);
   dlg.m_info += "\r\n";

   dlg.m_info += "No. of Samples (per channel) = ";
   dlg.m_info += itoa(GetDocument()->m_TheWave.GetNumSamples(),buf,10);
   dlg.m_info += "\r\n";

   dlg.DoModal();
}
```

That's it! Build the app and run it, open some wave files, and the finished product should look something like this:



By the way, under Windows 95, there are usually a few `.wav` files of various formats lying around in your `Windows\Media` directory.

Summary

In this chapter, we covered the basic, built-in multimedia system in Windows. We explored the WINMM MCI and low-level device control APIs.

In the next chapter, we'll look at how we can use device-independent bitmaps in our MFC programs.

Working with Device-independent Bitmaps (DIBs)

In this chapter, we're going to demystify the device-independent bitmap (DIB). DIBs are widely used by the various APIs, such as, DirectX, WinG, WinToon, and OpenGL, so getting a handle on how to use them effectively is very important if you're planning on doing any multimedia development.

Many developers, especially those who have some 16-bit Windows SDK development experience, cringe at the thought of having to work with DIBs. As we'll see in this chapter, new APIs in Win32 make DIBs much easier to handle.

In the first part of this chapter, we'll review the basic concepts of working with DIBs. Since it's impossible to discuss DIBs without doing so, we'll also get into the basics of palette management. Next we'll build a simple wrapper class to hide the mechanics of DIB management, and then we'll build a simple app that loads and displays DIBs. Finally, we'll talk about how to do double-buffered animation in a **`CView`** with an off-screen DIB buffer.

The DIB FAQ

Since so many developers seem to have trouble grasping the basics of DIBs, I thought I'd format the first part of this chapter like an Internet frequently-asked questions (FAQ) list. If you're new to working with DIBs, these are some of the things that could be giving you some trouble.

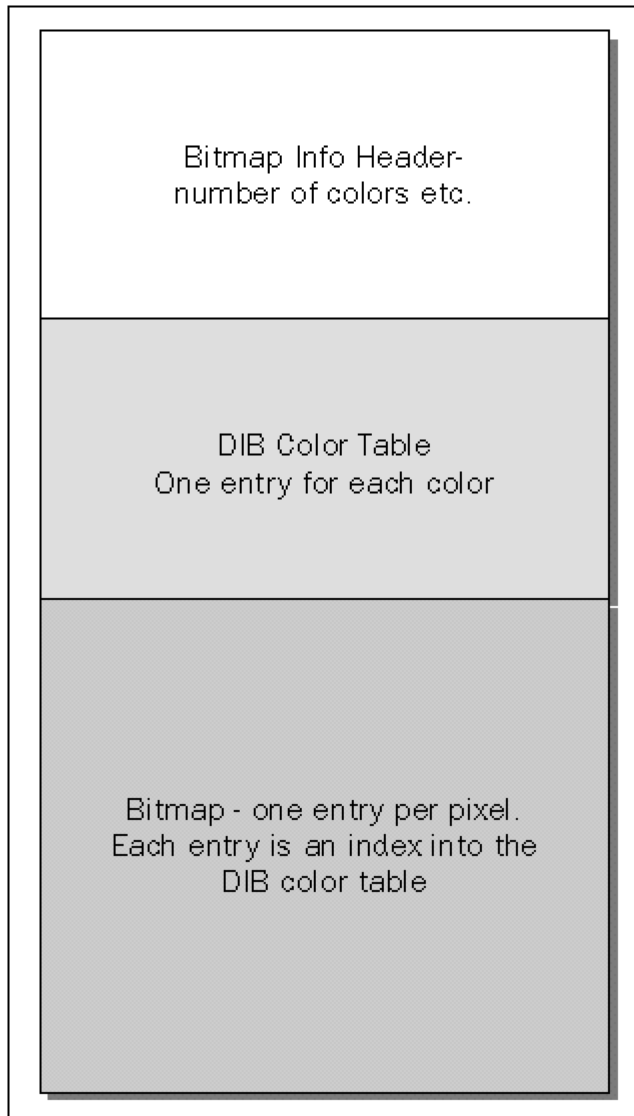
What does DIB stand for?

Device-**i**ndependent **b**itmap.

What makes it device-independent? Why does device independence matter?

A bitmap image is just a block of data that represents what the pixels should look like when the bitmap is rendered to the screen. Originally, bitmaps in Windows were all device-dependent. In other words, they were managed by the device drivers that would be used to display them. You didn't actually have direct access to the bitmap's pixel data; you had to call **`GetBitmapBits()`** to get a copy of the bitmap's pixel data from the driver, and **`SetBitmapBits()`** to put it back. The reason for this indirection was to allow the device driver to store the actual bitmap in the format most convenient for displaying the bitmap on the device hardware. Back then, bitmaps generally came in two color formats: monochrome (1 bit per pixel) and VGA (4 bits per pixel). The color information was absolute and mapped into a fixed table of the VGA colors.

With the advent of OS/2, Microsoft and IBM decided that they needed to develop a more robust means of handling bitmaps. Displays were getting more sophisticated and were supporting 256 colors or more, and the hardware was increasingly supporting palettes. To meet these needs, Microsoft and IBM created the device-independent bitmap (DIB) file format.



DIB pixels don't store raw color data. Instead, they store indices into a color table that is stored along with the bit data in the DIB file. This color table can then be read back out of the DIB file with the bitmap data, and can be turned into a logical palette to be used when the bitmap is displayed. Once it has been loaded into memory, before it can be displayed, the DIB must be turned back into a device-dependent bitmap by the display driver (except when you're using WinG or the new Win32 DIB section (see later) functionality, which can handle raw DIB bits). Also, its palette must be realized in the display DC to ensure that it displays with the correct colors.

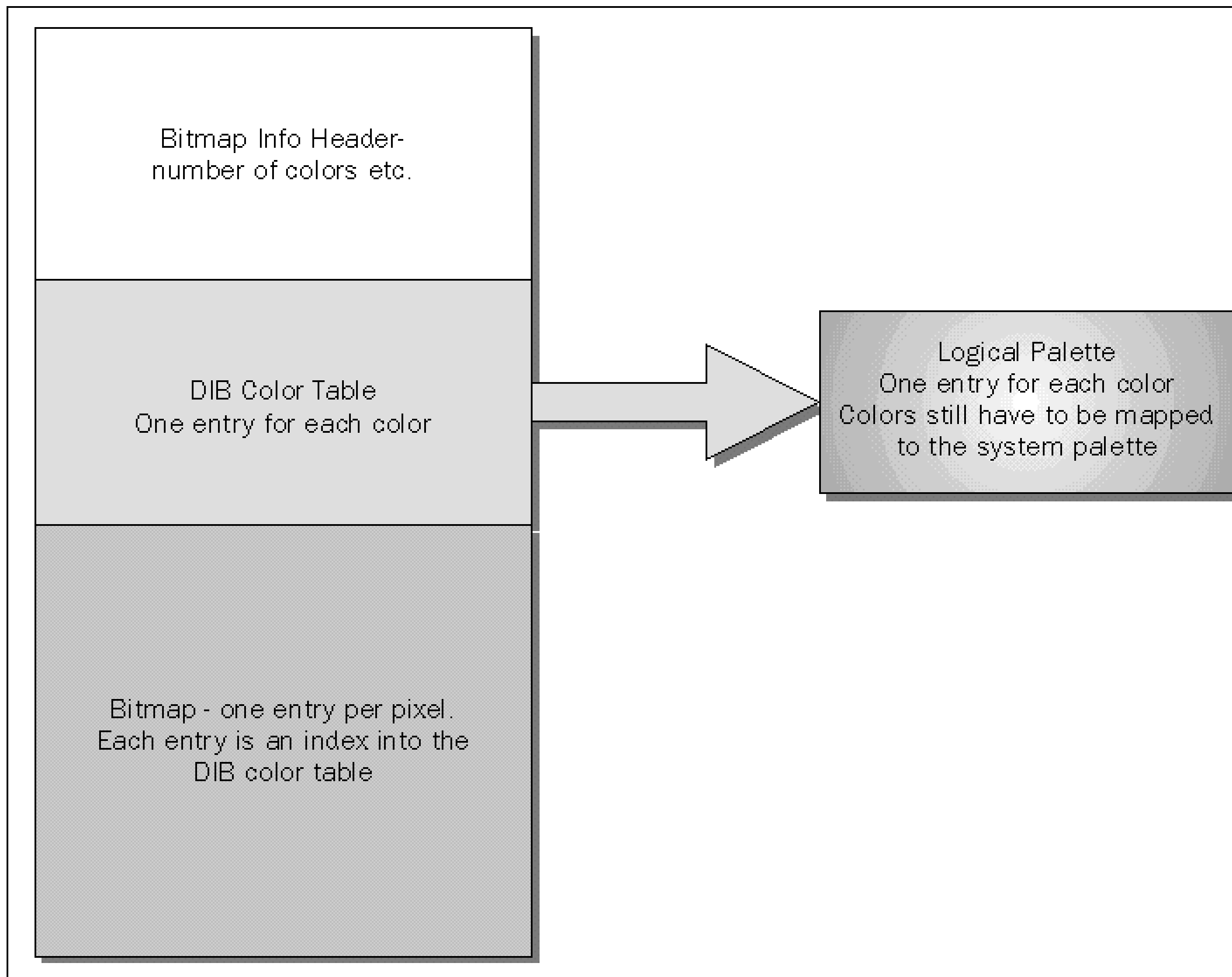
Why are many DIBs stored 'upside-down' (the top-most drawn scan line is at the end of the file)?

When Microsoft and IBM set out to lay down a standard DIB format, they were collaborating on OS/2. At that time, the thought was that GDI should start using an origin in the lower-left corner of the screen, like a normal Cartesian coordinate system. OS/2 adopted this, but it never got picked up by the Windows camp. Since the current DIB format stems from those early days of OS/2, many DIBs are in the 'upside-

down' format. In Win32, however, it's acceptable to store DIB scan lines top-first, so from now on we'll see more Windows-like DIBs.

What is a palette? How is it related to the DIB's color table?

A palette is a table of color values. Palettes allow color information to be removed from the raw image data. Instead of storing an RGB value for each pixel, the bitmap consists of palette indices. This makes the bitmap work something like a 'paint-by-numbers' picture. If the color value in a palette entry is changed, the pixels in the bitmap that reference that entry will then display the new color, without actually having to go through the bitmap and change the pixel values.



Although they both use the same mechanism, a palette is not the same as a DIB color table. The DIB color table maps color values into the DIB, while a palette maps color values into the system palette. We'll come back to the system palette a little later. You use a DIB's color table to construct a logical palette to

use when you're displaying the DIB. When it's loaded from a file, the DIB's color table normally holds the RGB color values to use for displaying the DIB. When a DIB is created in memory and is based on a particular DC, it's better to have the color table contain palette entries from the DC's palette rather than hard color values. This prevents GDI from constantly having to translate the color values into palette entries. Because the relationship between the DC and the DIB is presumably going to be pretty close (the DIB is going to be drawn in this DC a lot) it makes sense to do this palette-color table translation only once, when you create the DIB section. Since the actual palette entries are given, GDI will just use them. You'll notice that `CreateDIBSection()` has an option for doing this, and we'll use it when we get into double-buffered animation later in the chapter.

Why do I see 256-color DIBs so often?

256 colors (8 bits per pixel) is generally enough to achieve serviceable image color quality. Also, the Windows Palette Manager was designed at a time when 256-color display hardware was the wave of the future, so the Palette Manager is oriented around 256-color palettes. It still makes sense to try to hold to 256-color palettes for most purposes, even when a display is available that supports greater color depth. That way, if several 256-color images are displayed at once, the Palette Manager can see to it that their logical palettes appropriately share the system palette. One important point to note is that on a 256-color display, a DIB whose color table contains all 256 entries doesn't actually have access to the full 256-color system palette. 20 colors in the system palette (the first 10 and the last 10) are reserved by Windows. These are reserved to ensure that they are available for painting system-owned items. The colors in these entries are the 16 standard VGA colors, plus 4 more mandated by Microsoft.

There are all these different kinds of palettes: 'logical', 'foreground', 'background' and 'system' palettes. Sort this mess out for me.

The **system palette** is the structure Windows uses to represent the actual palette hardware in the display adapter. The system palette contains all of the colors that are actually being displayed on the entire screen.

A **logical palette** is just a set of colors that a particular application wishes to use to display something (a bitmap, etc.). It has no effect on the actual display until it is realized.
Any apps that want to display color data will try to realize logical palettes to get them into the system palette. However, the active or foreground app should get priority over the others if it wishes to assert a logical palette. The foreground app's palette is the one-and-only **foreground palette**, and the palettes asserted by any other app are *all* **background palettes**. Actually, any app in the z-order can try to realize a logical palette as the foreground palette, but it's considered improper to do so unless the app is truly in the foreground.

> *Just in case you haven't met it before, the z-order is the sequence in which all the currently overlapping windows are displayed. There's an imaginary axis (the z-axis), that sticks out at right angles to the surface of your display screen, along which all the currently displayed windows lie. The relative positions of all the current windows is called the z-order. The window at the bottom of the z-order is overlapped by all the other windows, whereas the one at the top will be displayed over all of the others.*

All of the realized palettes, foreground and background, 'vote' on what colors should wind up in the system palette, but the foreground palette's votes always override the others.

What is an identity palette?

An **identity palette** is a logical palette that contains the 20 reserved system colors (in the 10 first and 10 last entries), along with the colors needed by your app. This means that, on a 256-color video display, you don't actually get to use all 256 colors; the 20 system colors are reserved. You can free them up with the `SetSystemPaletteUse()` function, but that's generally frowned-upon because it can monkey with the appearance of system elements and other running apps as they are for system-wide use and generally assumed to be fixed.

On a display that supports more than 256 colors, you're likely to get all 256 colors in your palette mapped when in the foreground. On a 256-color display, the first 236 will map, then the rest will be unceremoniously truncated. So, you really only get 236 colors to play with. Also, if you make your logical palette structurally match the system palette, you'll get added speed because the system can use your palette as-is; it won't have to remap it when realized. Some paint programs that you can use to edit or create DIB files offer an option to include the system colors in the palette. You should generally select this option because it will prevent you from having to deal with remapping your palette to an identity palette when the image is loaded from the DIB file. The big exception to this rule is DirectDraw. In DirectDraw, you are bypassing the Palette Manager, so you have full access to the whole palette. DIB palettes used with DirectDraw need only include the system colors if they are actually being used by the image.

What does it mean to 'realize' a palette, and why do I have to do it?

Realizing a palette simply causes the system palette and your logical palette to reconcile. You aren't actually using your logical palette until it has been realized (selecting it into the DC is not enough). Realizing the logical palette causes it to map its values into the system palette, adding values as needed. The values are mapped to the system palette in the order in which they appear in the logical palette, so if you care, you should place more important colors near the beginning (in the lower indices) of the palette.

Realizing a palette can involve up to three steps. In the first step, the system palette is scanned to find an exact match for each of the colors in your logical palette. For each exact match, the color in the logical palette is mapped to the system palette entry. In the second step, if there are unmapped colors in your logical palette, the system palette is scanned for unused entries in which remaining logical palette colors can be inserted. If there are still unmapped colors in your logical palette, the entries in the system palette are scanned for a third time for each of the remaining logical palette colors, to determine the 'nearest' entry in the system palette. The nearest is determined by treating the RGB components of the colors as coordinates of notional 3D points and computing the distance between them using the usual geometric method. The degree to which realizing a palette will actually alter the system palette is, of course, partly determined by whether the palette is realized in the foreground or background. The foreground palette takes precedence, and only those system palette entries that have not been taken up by the foreground palette can be used for background palette colors.

Aren't there all kinds of crazy window messages I have to handle to make palettes work correctly in my app?

It's really not too complicated. Here are the messages you need to handle, and what you need to do to handle them:

Message	What's Needed
`WM_PALETTECHANGED`	The system palette has been altered. You will want to re-realize

and possibly repaint in response to this message. You should check first to see whether your app was the trigger for this message, and do nothing if it was. When you're overriding the **CWnd::OnPaletteChanged(CWnd* pFocusWnd)** member function, just check that **pFocusWnd** is not your window before you do anything.

WM_QUERYNEWPALETTE — Your app is being given foreground palette status. You will want to re-realize your palette so it can assert its priority.

Essentially, when either of these messages comes through, you basically want to re-realize your palette and repaint your app's client area if the realization changed:

```
int CMyViewClass :: OnPaletteMessage()
{
   CDC*   pDC = GetDC();

   CPalette* pOld= pDC->SelectPalette(m_pPal, FALSE);
   int result = pDC->RealizePalette();

   // bForceBackground is TRUE because palette should already
   // be mapped
   pDC->SelectPalette(pOld, TRUE);
   pDC->RealizePalette();

   ReleaseDC(pDC);

   // If realization changed, force repaint
   if (result)
      Invalidate();

   return result;
}
```

How do I read in the palette from the DIB file?

You don't actually read palettes from DIB files. Rather, you read in the DIB's header information, which includes its color table, and create a logical palette based on that. The following code will read the bitmap's color table from the file and make a palette from it (adapted from the **CDIB** constructor code that appears later in the chapter):

```
// Calculate the size needed to hold the BITMAPINFO
m_infosize = sizeof(BITMAPINFO) + sizeof(RGBQUAD)*num_entries;

// Allocate a BITMAPINFO
m_hInfo = GlobalAlloc(GHND, m_infosize);
m_lpInfo = (LPBITMAPINFO)GlobalLock(m_hInfo);


// Read BITMAPINFO from the file represented by hfile
_lread(hfile, m_lpInfo, m_infosize);


// Allocate and configure the LOGPALETTE
hlogpal = GlobalAlloc(GHND, sizeof(LOGPALETTE) +
   sizeof(PALETTEENTRY)*num_entries);

lplogpal = (LPLOGPALETTE) GlobalLock(hlogpal);

lplogpal->palVersion = 0x300;
lplogpal->palNumEntries = num_entries;


// Copy entries into LOGPALETTE
```

```
for (int i=0; i<num_entries; i++)
{
   // Entries are stored in BGR, not RGB, order
   lplogpal->palPalEntry[i].peRed = entries[i].peBlue;
   lplogpal->palPalEntry[i].peGreen = entries[i].peGreen;
   lplogpal->palPalEntry[i].peBlue = entries[i].peRed;
   lplogpal->palPalEntry[i].peFlags = 0;
}


// Create the palette
m_palette = new CPalette;
m_palette->CreatePalette(lplogpal);

// Clean up
GlobalUnlock(hlogpal);
GlobalFree(hlogpal);
GlobalUnlock(m_hInfo);
GlobalFree(m_hInfo);
```

What is a DIB section?

The conventional way to make a DIB available for drawing by GDI is to shuttle it into a device-dependent bitmap through the driver, so there are two copies of the bitmap bits: one on the application side for you to work with, and one on the driver side for display. Any time you wish to alter the bits, you must shuttle them back through the driver to make them available again for display. DIB sections, new to Windows NT and 95, eliminate this. A DIB section is a common block of memory that is accessible both to your application and to the display driver. This block holds the one and only copy of the DIB bits. GDI can access it through an **HBITMAP** and your app can directly modify it. Because there is no extra copying of bits going on, working with DIB sections is both extremely fast (suitable for animation, etc.) and very simple, since the DIB section can be manipulated like a conventional device dependent bitmap(DDB) **HBITMAP**.

A side note: the term *section* here refers to one mode of the DIB section in which it can be created based on a memory-mapped file section object. This is not the usual way to work with them, but even so, the name has stuck.

That sounds sort of like WinG to me. What exactly is WinG, and how does it relate to DIB sections?

Prior to the introduction of DIB sections, the WinG API was released to provide an equivalent interim solution. WinG is just a DIB section implementation for Windows 3.1 and Win32s. The **CreateDIBSection()** API completely supersedes WinG for Windows NT and Windows 95. A WinG API is provided for NT and 95 for backward compatibility, but it simply calls **CreateDIBSection()**.

How do I create a DIB section?

You call **CreateDIBSection()**, of course!

```
HBITMAP CreateDIBSection( HDC hdc, CONST BITMAPINFO *pbmi, UINT iUsage,
    VOID **ppvBits,  HANDLE hSection, DWORD dwOffset )
```

Parameter	Meaning
hdc	The device context handle to use when initializing the DIB's colors when **iUsage** is set to **DIB_PAL_COLORS**.
pbmi	The pointer to a **BITMAPINFO struct** describing the DIB.

iUsage	Can be one of the following values: **DIB_PAL_COLORS** - uses the DC identified by **hdc** to initialize the DIB's colors. **DIB_RGB_COLORS** - uses the RGB values in the color table in the **BITMAPINFO** pointed to by **pbmi**.
ppvBits	The pointer to a pointer to receive the address of the DIB section buffer.
hSection	The optional file-mapping object to use to create the DIB section.
dwOffset	The offset within the file mapping of the bitmap's bits.

How do I read a DIB in from a file?

It's now quite simple in the new Windows 95 API to read a DIB file. In fact, you can read in a DIB and create a DIB section for it in one function call: **LoadImage()**.

```
HANDLE LoadImage( HINSTANCE  hinst,  LPCTSTR  lpszName,  UINT  uType,
    int  cxDesired, int  cyDesired, UINT  fuLoad  );
```

Parameter	Meaning
hinst	The handle of the module instance to read from when reading an image stored as a resource (**NULL** when reading from file).
lpszName	The name of resource, or path to file.
uType	The type of image to read: **IMAGE_BITMAP**, **IMAGE_CURSOR**, or **IMAGE_ICON**.
cxDesired	The desired width of icon or cursor. A value of 0 selects the **SM_CXICON** or **SM_CXCURSOR** system metric value. We won't use this parameter in the example. It's set to 0.
cyDesired	The desired height of icon or cursor. A value of 0 selects the **SM_CYICON** or **SM_CYCURSOR** system metric value. We won't use this parameter in the example. It's set to 0.
fuLoad	The flags controlling the image load (listed in the table below). The only two we're interested in are **LR_CREATEDIBSECTION** and **LR_LOADFROMFILE**.

The following table lists the possible values for **fuLoad**. Only the first two (**LR_CREATEDIBSECTION** and **LR_LOADFROMFILE**) are generic; the other flags all Windows 95 only.

fuLoad Value	Meaning
LR_CREATEDIBSECTION	The new image will have been created via **CreateDIBSection()**.
LR_LOADFROMFILE	Read image from a file rather than a resource.
LR_DEFAULTCOLOR	Specifies to load image as a color image and is the default value of **fuLoad**.
LR_DEFAULTSIZE	Use the width or height specified by the system metric values for

	cursors and icons when the **cxDesired** or **cyDesired** values are set to zero.
LR_LOADMAP3DCOLORS	Search the color table for the image and replace Dk Gray, **RGB(128,128,128)**, Gray, **RGB(192,192,192)**, and Lt Gray, **RGB(223,223,223)** with the corresponding 3D colors, **COLOR_3DSHADOW**, **COLOR_3DFACE**, and **COLOR_3DLIGHT**
LR_LOADTRANSPARENT	Replace the entry in the color table corresponding to the color value of the first pixel in the image with the default window color. **LR_LOADTRANSPARENT** takes precedence over **LR_LOADMAP3DCOLORS** if you specify both.
LR_MONOCHROME	Load the image in black and white.
LR_SHARED	Share the image handle if the image is loaded more than once.

A typical call to this function to load a DIB from a file might look like this:

```
HBITMAP   hbmNew;
hbmNew = (HBITMAP)LoadImage(NULL, "FOO.BMP", IMAGE_BITMAP, 0, 0,
   LR_CREATEDIBSECTION|LR_LOADFROMFILE);
```

How do I draw a DIB section onto a DC?   I recall all kinds of complicated calls from Windows 3.1, like StretchDIBits().

Unless you really want to stretch a bitmap, forget about **StretchDIBits()**.DIB sections are designed to work like the old DDBs from GDI's perspective. All you have to do is call good old **BitBlt()**:

```
// destDC is the DC we want  to draw on
// m_bitmap is the DIB section HBITMAP
// m_palette is the palette to use with this bitmap

CDC dcMem;

// Select the palette into the screen DC and realize it
CPalette* pold = destDC->SelectPalette(m_palette, FALSE);
destDC->RealizePalette();

// Create a memory DC compatible with the screen DC
dcMem.CreateCompatibleDC(destDC);

// Put the bitmap bits into the memory DC
 CBitmap* pbmOld = dcMem.SelectObject(&m_bitmap);

// Copy the bitmap bits to the screen DC
destDC->BitBlt(dx, dy, sw, sh, &dcMem, sx, sy, SRCCOPY);

// Put the original bitmap back
dcMem.SelectObject(pbmOld);

// Put the original palette back
destDC->SelectPalette(pold, FALSE);
```

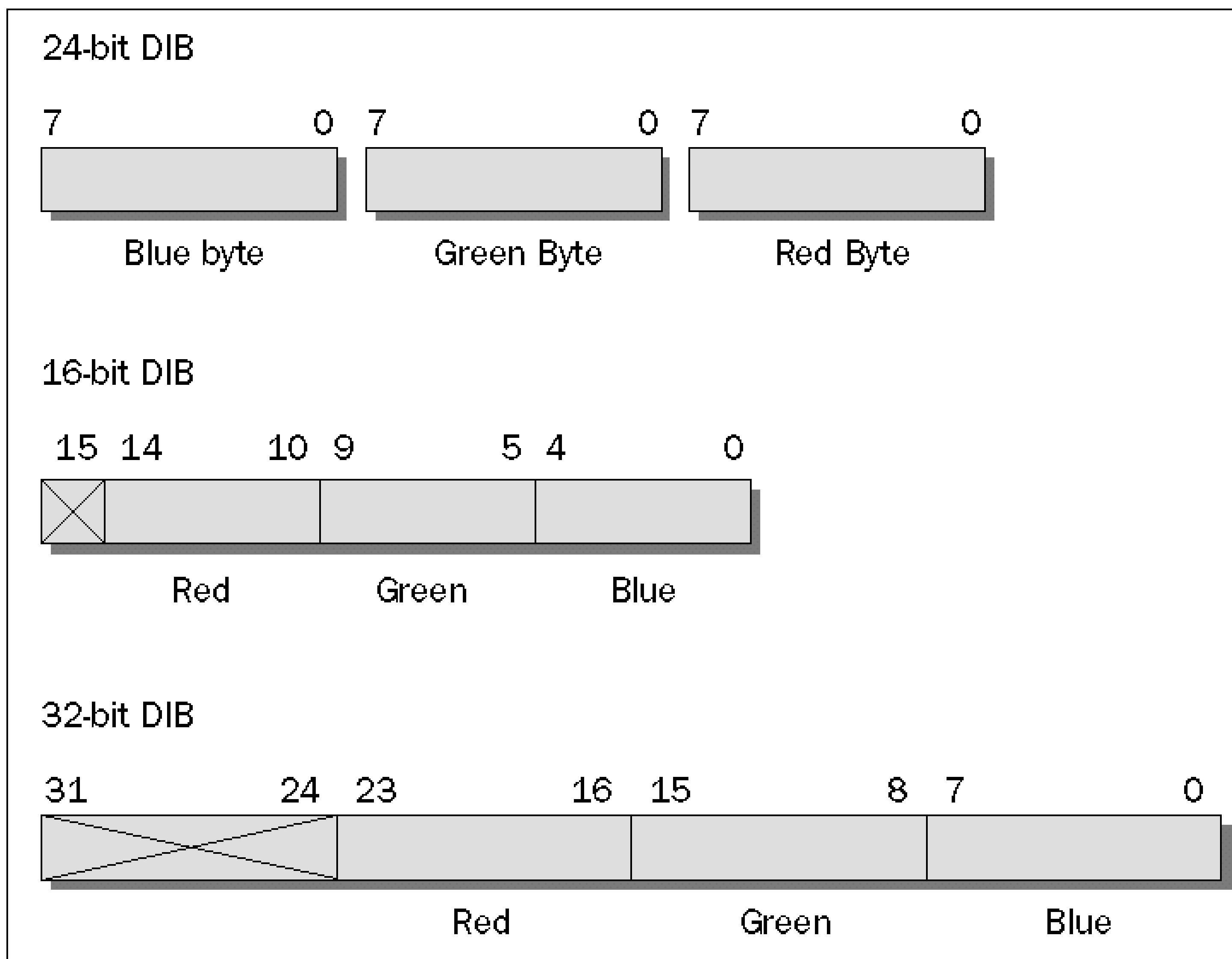
If DIB sections really work similarly to old DDB HBITMAPs, can I wrap a CBitmap around one, and let it manage it for me?

Yes and no. The **CBitmap** class is not going to be able to help you create the DIB section, but once you've

got an **HBITMAP** for the DIB section, you can use **Attach()** to hand it off to the **CBitmap** object. This makes it easier to use MFC GDI wrappers with the bitmap. We'll do this in our **CDIB** class later in the chapter.

What do I do if I want to use a DIB with more than 256 colors?

Go right ahead and use them. The Windows DIB format supports 16, 24 and 32 bit-per-pixel DIBs. Be aware, however, that not all hardware is going to be able to handle all of these formats, and some nasty driver translation might happen to your DIB before it gets displayed. No matter what, though, you're supposed to get something displayable, regardless of what the hardware actually supports. Also be aware that you're leaving the realm of palettes when you use these higher-color formats; the pixel values are raw color values.



CDIB: A Simple DIB Wrapper Class

It would make sense to embody the concepts covered above in a C++ class for handling DIBs. This class could act as a wrapper around a DIB and its color table, providing methods for loading DIBs from `.bmp` files or resources and for drawing the DIB onto a DC. Note that, since we're going to use the **`LR_CREATEDIBSECTION`** functionality of **`LoadImage()`** in this class, it will only work correctly under Windows 95 (until Microsoft fixes the Windows NT version of **`LoadImage()`** to work correctly with DIB sections).

Let's start by laying out a header for the class. We'll need member variables to hold an **`HBITMAP`** and a pointer for the two principal means of accessing the DIB section in memory. We'll also need some members to hold the bitmap's header information, and we'll need one for the logical palette.

In the way of member functions, we'll need accessors for the various member variables we've defined. We'll also need two constructors: one to load in the DIB from a `.bmp` file or from resources, and another to make an empty DIB using **`CreateDIBSection()`** based on a DC (this constructor will be used to make DIB section back buffers for off-screen drawing of animation frames). We'll need a destructor to do some clean-up. Finally, we'll want a function for drawing the DIB onto a DC. We'll **`inline`** the drawing function for maximum speed.

```
// CDIB.H - Class header for DIB object


#ifndef _CDIB_DEFINED_
#define _CDIB_DEFINED_

class CDIB
{
protected:

   // The bitmap's pixel data
   CBitmap        m_bitmap;    // HBITMAP version
   LPVOID         m_lpBits;    // Buffer version

   // Bitmap's header info
   HANDLE         m_hInfo;
   LPBITMAPINFO   m_lpInfo;
   LONG           m_infosize;

   // The bitmap's palette
   CPalette*      m_palette;

   // Error flag
   BOOL           m_pal_loaded;

   // Bitmap's width and height, in pixels
   LONG           m_width, m_height;

public:

   // ctor to load a DIB from either a file or the app's resource data
   CDIB (const char* name);
   CDIB (UINT nResID);

   // Ctor for making a DIB back-buffer based on a sprite or
   // background DIB
   CDIB (const CDC* pDC, CDIB* based_on);


   // Accessors
   CBitmap&      GetBits()          {return m_bitmap;}
   LPVOID        GetRawbits()        {return m_lpBits;}
```

```
   LONG          GetWidth()          {return m_width;}
   LONG          GetHeight()         {return m_height;}
   LPBITMAPINFO  GetInfo()           {return m_lpInfo;}
   long          GetInfoSize()       {return m_infosize;}
   CPalette*     GetPalette()       {return m_palette;}
   BOOL          IsPaletteLoaded()  {return m_pal_loaded;}

   // Draw function
   void Draw(CDC* destDC, int dx, int dy,
      int sx, int sy, int sw, int sh);

   // Dtor
   ~CDIB()
   {
      delete m_palette;

      if (m_hInfo)
      {
         GlobalUnlock(m_hInfo);
         GlobalFree(m_hInfo);
      }
   }
};

inline void CDIB :: Draw(CDC* destDC, int dx, int dy,
            int sx, int sy, int sw, int sh)
{
   CDC dcMem;
   CPalette* pOld;

   // Select the palette into the screen DC and realize it
   // Force background, because palette should already be realized in
   // foreground if necessary
   pOld = destDC->SelectPalette(m_palette, TRUE);
   destDC->RealizePalette();

   // Create a memory DC compatible with the screen DC
   dcMem.CreateCompatibleDC(destDC);

   // Put the bitmap bits into the memory DC
   CBitmap* pbmOld = dcMem.SelectObject(&m_bitmap);

   // Copy the bitmap bits to the screen DC
   destDC->BitBlt(dx, dy, sw, sh, &dcMem, sx, sy, SRCCOPY);

   // Put the original bitmap back
   dcMem.SelectObject(pbmOld);

   // Put the original palette back
   destDC->SelectPalette(pOld, TRUE);
   destDC->RealizePalette();
}

#endif
```

Next, over in the implementation file, we'll build the constructor that reads in a DIB from a file or from resource data. This function first calls the new Windows 95 function, `LoadImage()`, to create the DIB section and initialize its data from the file or resource, and then it read in the palette information using the technique described above.

The constructor is overloaded. The first takes a file name:

```
#include "stdafx.h"
#include "cdib.h"
#include <io.h>

CDIB :: CDIB(const char* name)
{
```

```
HBITMAP the_bits;

// 1. Read bits from file
the_bits = (HBITMAP)LoadImage(NULL, name, IMAGE_BITMAP, 0, 0,
   LR_LOADFROMFILE|LR_CREATEDIBSECTION);

m_bitmap.Attach(the_bits);


// 2. Get palette from file or resource

m_pal_loaded = FALSE;
m_palette = 0;

int hfile;

// Open the file
hfile = _lopen(name, OF_READ);


// Only proceed if file could be opened
if (hfile != -1)
{
   BITMAPINFOHEADER   bminfo;
   BITMAPFILEHEADER   bmfile;
   PALETTEENTRY      entries[256];
   int               num_entries;
   HANDLE            hlogpal;
   LPLOGPALETTE      lplogpal;


   // Read headers and palette entries out of file
   _lread(hfile, &bmfile, sizeof(bmfile));
   _lread(hfile, &bminfo, sizeof(bminfo));
   _lread(hfile, &entries, sizeof(entries));


   if ((bminfo.biSize != sizeof(BITMAPINFOHEADER)) ||
       (bminfo.biBitCount > 8))
   {
      _lclose(hfile);
      return;
      // Bad header or more than 256 colors - can't go on
   }


   // if biClrUsed is 0, palette is using max number of
   // entries for its bitwidth.  Otherwise, biClrUsed
   // specifies the actual number of palette entries in use.
   if (bminfo.biClrUsed == 0)
      num_entries = 1 << bminfo.biBitCount;
   else
      num_entries = bminfo.biClrUsed;


   // Remember the bitmap's width and height
   m_width = bminfo.biWidth;
   m_height = bminfo.biHeight;


   m_infosize = sizeof(BITMAPINFO) + sizeof(RGBQUAD)*num_entries;

   m_hInfo = GlobalAlloc(GHND, m_infosize);

   if (!m_hInfo)
   {
      _lclose(hfile);
      return;
   }

   m_lpInfo = (LPBITMAPINFO)GlobalLock(m_hInfo);
```

```
        if (!m_lpInfo)
        {
            _lclose(hfile);
            return;
        }

        // Rewind file and read whole BITMAPINFO for later use
        _lseek(hfile, 0, SEEK_SET);
        _lread(hfile, m_lpInfo, m_infosize);
        _lclose(hfile);

        // Allocate storage for the LOGPALETTE
        hlogpal = GlobalAlloc(GHND, sizeof(LOGPALETTE) +
            sizeof(PALETTEENTRY)*num_entries);

        if (hlogpal)
        {
            lplogpal = (LPLOGPALETTE)GlobalLock(hlogpal);

            lplogpal->palVersion = 0x300;
            lplogpal->palNumEntries = num_entries;


            // Copy entries into LOGPALETTE
            for (int i=0; i<num_entries; i++)
            {
                lplogpal->palPalEntry[i].peRed   =  entries[i].peBlue;
                lplogpal->palPalEntry[i].peGreen = entries[i].peGreen;
                lplogpal->palPalEntry[i].peBlue  = entries[i].peRed;
                lplogpal->palPalEntry[i].peFlags = 0;
            }


            // Create the palette
            m_palette = new CPalette;
            m_palette->CreatePalette(lplogpal);

            // Clean up
            GlobalUnlock(hlogpal);
            GlobalFree(hlogpal);

            m_pal_loaded = TRUE;
        }
    }
}
```

The second takes a resource ID for a bitmap:

```
CDIB :: CDIB(UINT nResID)
{
    HBITMAP the_bits;

    // 1. Read bits from resource
    the_bits = (HBITMAP)LoadImage(GetModuleHandle(NULL),
          MAKEINTRESOURCE(nResID),
          IMAGE_BITMAP, 0, 0, LR_CREATEDIBSECTION);

    m_bitmap.Attach(the_bits);


    // 2. Get palette from file or resource

    m_pal_loaded = FALSE;
    m_palette = 0;

    // Load palette from resource
    HRSRC hbmres = FindResource(NULL, MAKEINTRESOURCE(nResID), RT_BITMAP);

    if (hbmres)
```

```
{
   LPBITMAPINFOHEADER lpbminfo = (LPBITMAPINFOHEADER)LockResource(
      LoadResource(NULL,hbmres));

   int                  num_entries;
   HANDLE               hlogpal;
   LPLOGPALETTE         lplogpal;


   if (lpbminfo && (lpbminfo->biSize >= sizeof(BITMAPINFOHEADER)) &&
       (lpbminfo->biBitCount <= 8))
   {
      RGBQUAD* entries = (RGBQUAD*)((BYTE*)lpbminfo +
         lpbminfo->biSize);

      // if biClrUsed is 0, palette is using max number of
      // entries for its bitwidth.  Otherwise, biClrUsed
      // specifies the actual number of palette entries
      // in use.
      if (lpbminfo->biClrUsed == 0)
         num_entries = 1 << lpbminfo->biBitCount;
      else
         num_entries = lpbminfo->biClrUsed;


      // Remember the bitmap's width and height
      m_width = lpbminfo->biWidth;
      m_height = lpbminfo->biHeight;

      m_infosize = sizeof(BITMAPINFO) + sizeof(RGBQUAD)*num_entries;

      m_hInfo = GlobalAlloc(GHND, m_infosize);

      if (!m_hInfo)
         return;

      m_lpInfo = (LPBITMAPINFO)GlobalLock(m_hInfo);

      if (!m_lpInfo)
         return;

      // Copy BITMAPINFO
      memcpy(m_lpInfo, lpbminfo, m_infosize);


      // Allocate storage for the LOGPALETTE
      hlogpal = GlobalAlloc(GHND, sizeof(LOGPALETTE) +
         sizeof(PALETTEENTRY)*num_entries);

      if (hlogpal)
      {
         lplogpal = (LPLOGPALETTE)GlobalLock(hlogpal);

         lplogpal->palVersion = 0x300;
         lplogpal->palNumEntries = num_entries;


         // Copy entries into LOGPALETTE
         for (int i=0; i<num_entries; i++)
         {
            lplogpal->palPalEntry[i].peBlue  = entries[i].rgbBlue;
            lplogpal->palPalEntry[i].peGreen = entries[i].rgbGreen;
            lplogpal->palPalEntry[i].peRed   = entries[i].rgbRed;
            lplogpal->palPalEntry[i].peFlags = 0;
         }

         // Create the palette
         m_palette = new CPalette;
         m_palette->CreatePalette(lplogpal);

         // Clean up
```

```
                    GlobalUnlock(hlogpal);
                    GlobalFree(hlogpal);

                    m_pal_loaded = TRUE;
                 }
              }
           }
        }
```

Whew! There's a lot there, but it's pretty easy to understand. It's actually the same thing twice; once for files and once for resources, with slight variations.

The last piece of code is the constructor for creating an empty DIB with **`CreateDIBSection()`**. Since this DIB will usually be used as an offscreen buffer for double-buffered animation, it will have its palette initialized from a DC passed into the constructor, and its palette will be taken from another DIB, usually the background DIB for the animation, or alternatively, a DIB containing sprite graphics. Here's the code:

```
CDIB :: CDIB(const CDC* pDC, CDIB* based_on)
{
   HBITMAP    hbmNew;
   WORD*      pEntry;
   int        i;
   HANDLE     hLogPal;

   m_width = based_on->width();
   m_height = based_on->height();

   // Allocate a BITMAPINFO struct with space for 256 colors
   m_infosize = sizeof(BITMAPINFOHEADER) + (sizeof(WORD) * 256);

   m_hInfo = GlobalAlloc(GHND, m_infosize);

   if (!m_hInfo)
      return;

   m_lpInfo = (LPBITMAPINFO)GlobalLock(m_hInfo);

   if (!m_lpInfo)
      return;


   // Copy the header info from the based-on DIB
   memcpy(m_lpInfo, based_on->info(), sizeof(BITMAPINFOHEADER));


   // Build a color table for the new DIB that maps 1-1
   // with the base DC's palette

   m_lpInfo->bmiHeader.biClrUsed= 0;

   pEntry = (WORD*)((BYTE*)m_lpInfo + sizeof(BITMAPINFOHEADER));

   for (i=0; i<256; i++)
      *pEntry++ = (WORD)i;


   // Construct a logical palette for the DIB

   hLogPal = GlobalAlloc(GHND, sizeof(LOGPALETTE) +
      sizeof(PALETTEENTRY)*256);

   if (hLogPal)
   {
      // Grab a copy of the DC's current palette for later reference
      CPalette* temp_pal = pDC->GetCurrentPalette();

      BYTE* lpLogPal = (BYTE*)GlobalLock(hLogPal);
```

```
        ((LPLOGPALETTE)lpLogPal)->palVersion = 0x300;
        ((LPLOGPALETTE)lpLogPal)->palNumEntries = 256;

        // Copy the palette entries from the DC's palette
        temp_pal->GetPaletteEntries(0,256,
           (PALETTEENTRY*)(lpLogPal + 2 * sizeof(WORD)));

        // Create the palette
        m_palette = new CPalette;
        m_palette->CreatePalette((LPLOGPALETTE)lpLogPal);

        // Clean up
        GlobalUnlock(hLogPal);
        GlobalFree(hLogPal);

        m_pal_loaded = TRUE;
     }


     // Create and attach the DIB section
     hbmNew = CreateDIBSection(pDC->GetSafeHdc(), m_lpInfo, DIB_PAL_COLORS,
        &m_lpBits, NULL, 0);
     m_bitmap.Attach(hbmNew);
  }
```

That's it. Now, let's put the `CDIB` class to work in some sample applications.

Project DIBTest: A Simple MDI DIB Viewer

The purpose of this project is to demonstrate the `CDIB` class in action. In this project, we'll load and draw DIBs, and we'll employ the basic palette-management techniques for an app that displays DIBs, including some additional steps necessary for managing multiple document views with different palettes.

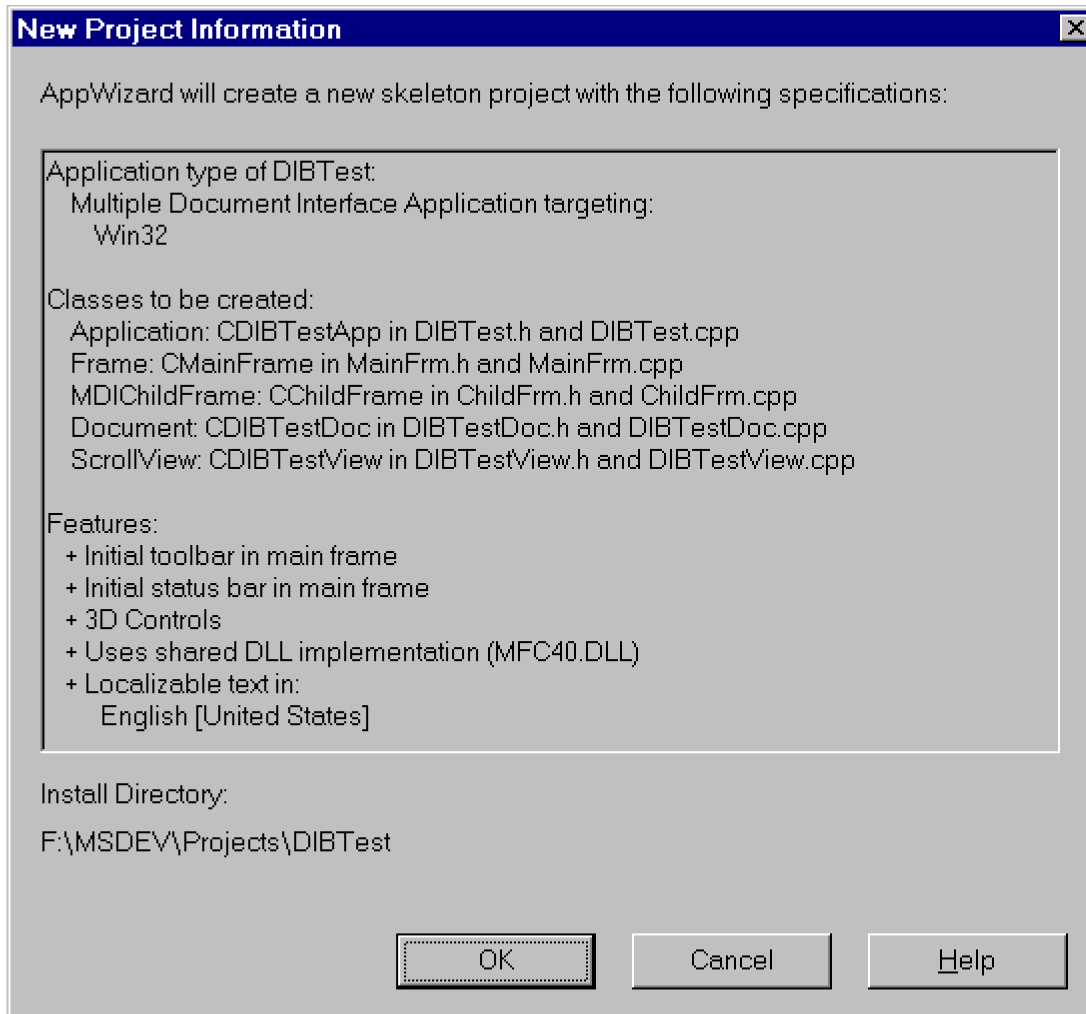
This application allows the user to open `.bmp` files, which it then displays in MDI views. The app properly handles the palette, so that when the individual views are activated, the correct palette is employed to display the foreground view's image. This is also true if the entire app is activated and deactivated. To test this out for yourself, set your video driver to a 256-color mode and load a few other graphics-intensive apps when you're running `DIBTest`.

Step 1

Use AppWizard to create a new project called DIBTest. Create the project using the following AppWizard options:

AppWizard Step 1: use the default setting (multiple documents).
AppWizard Step 2: use the default setting (no database support).
AppWizard Step 3: use the default settings (no OLE support).
AppWizard Step 4: use the default settings, except turn off printing and print preview support (we won't be using it).
AppWizard Step 5: use the default settings.
AppWizard Step 6: use the default settings, except make `CDIBTestView` descend from `CScrollView`.

If everything worked all right, you should wind up with a New Project Information dialog that looks like this:



Step 2

We also want to make sure that a new document is not automatically created on startup, so go into **`CDIBTestApp::InitInstance()`**, find the command line processing code (near the end), and make the following changes:

```
// Parse command line for standard shell commands, DDE, file open
CCommandLineInfo cmdInfo;
ParseCommandLine(cmdInfo);

// Dispatch commands specified on the command line
if (cmdInfo.m_nShellCommand != CCommandLineInfo::FileNew)
   if (!ProcessShellCommand(cmdInfo))
      return FALSE;
```

Step 3

The document class in this app will hold the DIB for us, so we need to add a member variable to it for the DIB. Go to the **`CDIBTestDoc`** declaration and add:

```
// Attributes
public:
   CDIB*   m_DIB;
```

We'll need to properly initialize this pointer in the constructor and delete it in the destructor:

```
///////////////////////////////////////////////////////////////////////////////
// CDIBTestDoc construction/destruction

CDIBTestDoc::CDIBTestDoc()
{
   m_DIB = 0;
}

CDIBTestDoc::~CDIBTestDoc()
{
   delete m_DIB;
}
```

When a new document is file is opened, we'll have to load the DIB. Use ClassWizard to add an **OnOpenDocument()** handler:

```
BOOL CDIBTestDoc::OnOpenDocument(LPCTSTR lpszPathName)
{
   if (!CDocument::OnOpenDocument(lpszPathName))
      return FALSE;


   the_dib = new CDIB(lpszPathName);

   if (!m_DIB->IsPaletteLoaded() || ! m_DIB->GetRawbits() ||
      ! m_DIB->GetInfo())
   {
      AfxMessageBox("Unable to read DIB file.");
      return FALSE;
   }


   return TRUE;
}
```

Step 4

Now on to the **CDIBTestView** class. This is a **CScrollView** descendant, so we'll have to set the scrolling ranges in the **OnInitialUpdate()** handler:

```
void CDIBTestView::OnInitialUpdate()
{
   CScrollView::OnInitialUpdate();
   CSize sizeTotal;

   sizeTotal.cx = GetDocument()->m_DIB->GetWidth();
   sizeTotal.cy = GetDocument()->m_DIB->GetHeight();

   SetScrollSizes(MM_TEXT, sizeTotal);
}
```

Of course, we'll need to draw the DIB in the view's **OnDraw()** handler:

```
void CDIBTestView::OnDraw(CDC* pDC)
{
   CDIBTestDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);

   pDoc-> m_DIB->Draw(pDC, 0, 0, 0, 0, pDoc-> m_DIB->GetWidth(),
      pDoc-> m_DIB->GetHeight());
}
```

This is an MDI app, and each view will have its own document containing the DIB, each with its own palette. Whenever the view is activated, it must assert its palette, so we'll make a function that asserts the view's palette, and we'll call it within an **OnActivateView()** handler:

```
void CDIBTestView::OnActivateView(BOOL bActivate, CView* pActivateView,
   CView* pDeactiveView)
{
   // Realize this view's palette in the foreground if activating,
   // or in the background if deactivating
   realizePalette(!bActivate);

   CScrollView::OnActivateView(bActivate, pActivateView, pDeactiveView);
}

int CDIBTestView::realizePalette(BOOL bForceBackground)
{

   // Locate the active view's document
   CDIBTestDoc* pDoc = GetDocument();


   // Get the palette from the active doc's dib
   CPalette* pPal = (CPalette*)pDoc-> m_DIB->GetPalette();


   // Realize the palette

   CDC*   pDC = GetDC();

   CPalette* pOld= pDC->SelectPalette(pPal, bForceBackground);
   int result = pDC->RealizePalette();

   // bForceBackground is TRUE because palette should already
   // be mapped
   pDC->SelectPalette(pOld, TRUE);
   pDC->RealizePalette();

   ReleaseDC(pDC);

   // If realization changed, force repaint
   if (result)
      Invalidate();

   return result;

}
```

Step 5

While we're on the subject of palettes, the app needs to handle system palette messages and respond using the palette from the active view. To do this, we'll add a function to **CMainFrame** that tells the active view to realize its own palette, and we'll call it from **OnPaletteChanged()** and **OnQueryNewPalette()** handlers for the **WM_PALETTECHANGED** and **WM_QUERYNEWPALETTE** messages respectively:

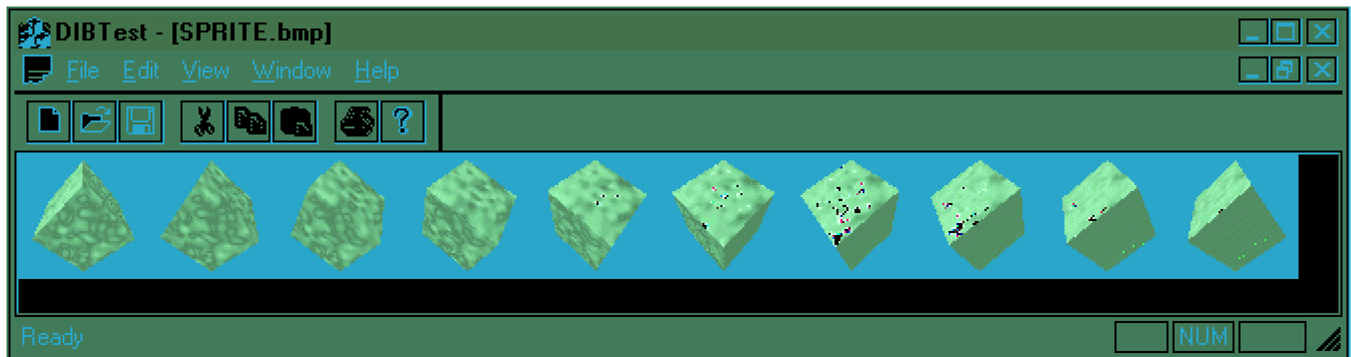
```
int CMainFrame::realizeActivePalette(CWnd* pFocusWnd)
{
   // Get the active view
   CDIBTestView* pView = (CDIBTestView*)GetActiveView();

   // Quit if this is the focuswnd or if there is no active view
   if ((!pView)||(pView == pFocusWnd))
      return 0;


   // Realize the view's palette in the foreground
   return pView->realizePalette(FALSE);
```

```
}

void CMainFrame::OnPaletteChanged(CWnd* pFocusWnd)
{
   CMDIFrameWnd::OnPaletteChanged(pFocusWnd);

   if (pFocusWnd != this)
      realizeActivePalette(pFocusWnd);
}

BOOL CMainFrame::OnQueryNewPalette()
{
   realizeActivePalette(this);

   return CMDIFrameWnd::OnQueryNewPalette();
}
```

Step 6

Use App Studio to remove any menus and toolbar items that we're not using.

That's it! The finished app should look something like this:



DIBs and Animation

We started off this chapter with a general discussion of DIBs. In the course of that discussion, the subject of bitmap drawing speed came up a few times. WinG, for instance, was developed as an interim solution to speeding up bitmap drawing, while `CreateDIBSection()` brought the WinG techniques into the mainstream API. So far, we've only shown how to draw DIBs in a very static fashion; speed has not been an issue. One of the most powerful uses of DIB section bitmaps is to perform smooth, flicker-free animation. To understand why, let's first look at the problems facing a developer of a multimedia application that needs to do high-quality animation.

The Need for Double-buffered Drawing

Animation is the process of making graphic elements move around on the screen in a realistic way. Those graphic elements are controlled by program logic that determines their positions and appearance throughout the animation sequence. When the program wishes to update the display, the real problems begin. As the graphic elements are moved, they need to be removed from their former positions and redrawn at their new locations. A primitive animation program might erase the area of the screen at an element's old position, replacing any background graphics 'underneath' the element, and then redraw the element in the new place. This results in the element briefly disappearing from the screen between the time it's erased and redrawn, which makes the animation flicker. One possible solution to this is to carefully keep track of the 'dirty' regions represented by the old and new positions and to draw the new position before erasing the old one. This could stop the flickering, but there's still another problem. You can't actually draw every element onto the scene simultaneously; each element is drawn one at a time. With fast graphics and a few elements, you can generate the illusion of simultaneity, but as the number of elements increases, this illusion begins to fall apart. The lag between the drawing of the first and last elements becomes perceivable.

Basically, any way you look at it, trying to perform animation by drawing directly to the screen is the wrong approach. There's is another way—one that game developers have been using for years—that gets around all of these problems. It's called **double-buffering**, and basically consists of assembling the entire scene in a bitmap in memory rather than on the screen (called a **back buffer** or **offscreen bitmap**) and then blitting the entire, finished scene frame to the screen DC all at once. That way, all of the dirty work of the scene's animation is entirely hidden from the viewer, so you get no flickering, no lag between drawing of elements, etc.

Double-buffering and CView

Windows provides no direct or inherent support for double-buffering (excluding the new DirectDraw API, which we cover in detail later in the book), and until the advent of `CreateDIBSection()` (or immediately before that, WinG) the complexity and speed problems involved discouraged many Windows developers from trying it. Now, however, it's actually quite easy to do. In fact, we've already added a constructor to our `CDIB` class just for this purpose. We want to be able to create a back buffer for a plain vanilla MFC `CView` descendant class, and use DIBs for the various graphic elements to be animated. Basically, all we have to do to make a back buffer for the `CView` is to grab the `CView`'s DC and construct a `CDIB` from it. We'll also use one of the scene element DIBs so that we can make sure the palette entries in the `CView` and the back buffer are in agreement with the scene DIBs' palette.

We want to make drawing with a back buffer as easy as drawing to the view's DC, so we'll create a DIB section based on the screen DC and put it into an off-screen DC for drawing. Then we can just draw to the offscreen DC rather than the screen DC, and assemble the scene however we like. When it's time to update the screen, we'll blit the whole thing to the screen. The easiest way to see how this works is just to

do it, so let's build a simple example program to try it out.

Project AnimTest: Double-buffered Animation with a CView

This project demonstrates the basic principles of double-buffered animation with DIBs and a **`CView`** descendant. We'll use the **`CDIB`** class in two modes: to hold the graphic elements used to assemble the scene, and also to act as the off-screen buffer in which the scene will be assembled before being blitted onto the screen.

This very simple application reads in a background image DIB and a DIB containing ten frames of sprite animation. It sets a timer to go off every 0.1 second and, on each timer message, it generates a fresh animation frame in the off-screen buffer and blits the assembled buffer to the screen. The result is smooth, flicker-free animation.

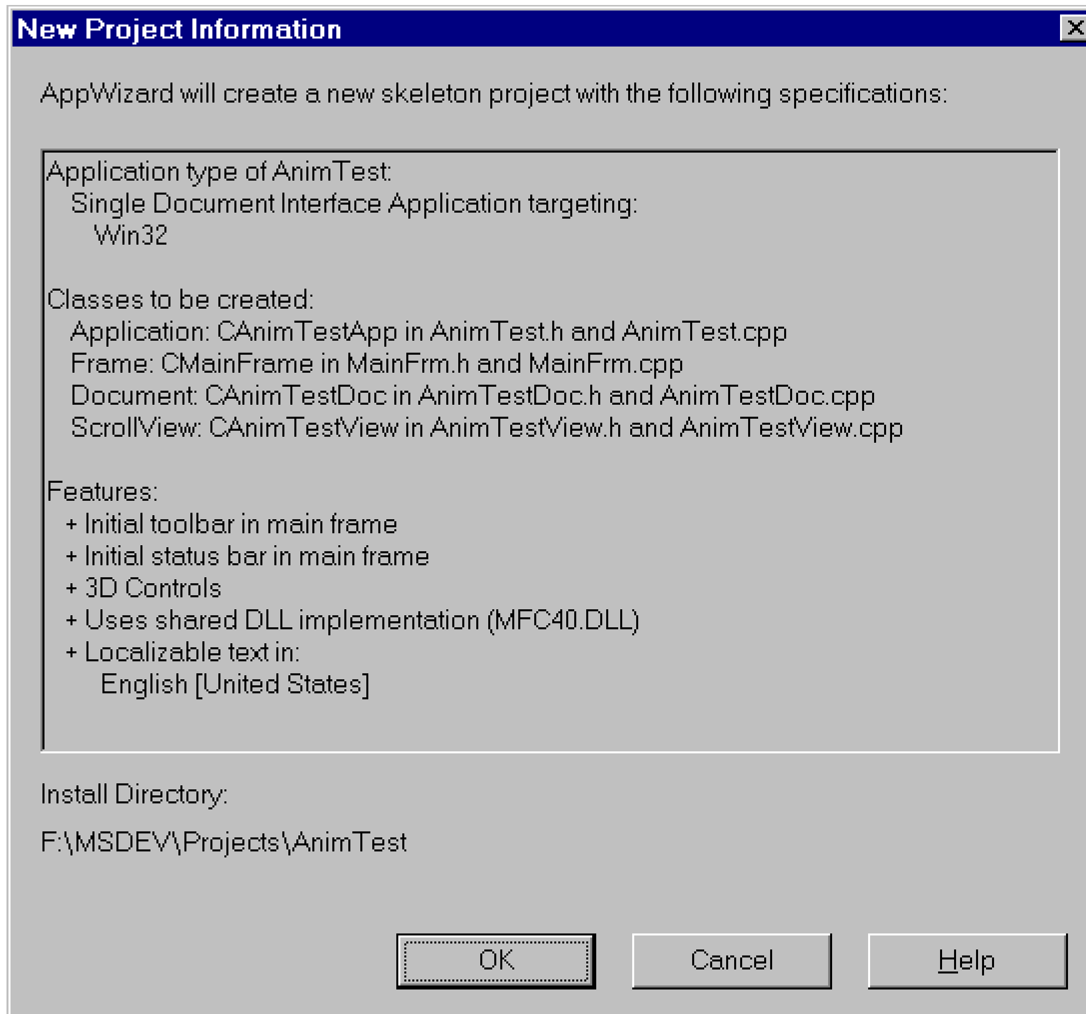
This is a very simple example. It covers in the most uncluttered, direct manner the basic concepts you need to get double-buffered animation going in an MFC-based app. Some improvements left to the reader for 'extra credit' might be using different ROPs with **`BitBlt()`** to achieve odd-shaped or transparent sprites, and optimized invalidation by tracking the dirty areas more carefully for improved speed so we're not re-blitting areas of the back buffer that didn't change.

Step 1

Use AppWizard to create a new project called AnimTest. Create the project using the following AppWizard options:

AppWizard Step 1: make this a single-document interface app.
AppWizard Step 2: use the default setting (no database support).
AppWizard Step 3: use the default settings (no OLE support).
AppWizard Step 4: turn off printing and print preview support (we won't be using it), and set the File Menu MRU count to 0.
AppWizard Step 5: use the default settings.
AppWizard Step 6: use the default settings, except make **`CAnimTestView`** descend from **`CScrollView`**.

If everything worked all right, you should wind up with a New Project Information dialog that looks like this:



Step 2

The bulk of the work in this app is done in the **CAnimTestView** class and we first need to add the following member variables to the class definition:

```
// Attributes
public:

   CDIB*   m_poffscr;   // A DIB for the back-buffer
   CDIB*   m_psprite;   // A DIB for the sprite
   CDIB*   m_pbkgnd;    // A DIB for the background bitmap

   // Animation members
   int m_frame;        // Current frame number for sprites
   int m_xpos;         // X position offset register
   int m_xvel;         // X velocity register
```

We'll also add some inline helper functions to the class deal with the animation timer:

```
   void setTimer() {SetTimer(DELAY_TIMER, DELAY_TIME, NULL);}
   void killTimer() {KillTimer(DELAY_TIMER);}
```

And definitions for the constants used by these helpers, at the top of the header:

```
#define DELAY_TIMER   1000
#define DELAY_TIME   100      // 0.1 sec delay time (10 frames/sec)
```

We can initialize the pointers to the DIBs in the constructor:

```
CAnimTestView::CAnimTestView()
{
   // Initialize the DIB pointers
   m_poffscr = 0;
   m_psprite = 0;
   m_pbkgnd = 0;
}
```

The destructor also needs to clean up the DIBs at shutdown:

```
CAnimtestView::~CAnimtestView()
{
   delete m_poffscr;
   delete m_psprite;
   delete m_pbkgnd;
}
```

The **`OnInitialUpdate`** handler has to do several things. First, it will load in the animation graphics into **`CDIB`**s—a backdrop bitmap and a bitmap containing ten images to use as sprite animation frames. Then it needs to set the scrolling to match the size of the backdrop bitmap. Next, it must grab the view's DC and construct the back buffer DIB. Then it initializes the variables used to keep track of the current state of the animation and it starts the animation timer. Finally, it renders the initial scene to the back buffer.

```
void CAnimTestView::OnInitialUpdate()
{
   CScrollView::OnInitialUpdate();
   CSize sizeTotal;


   // WROX 1: Load sprite and background bitmaps from resources
   m_psprite = new CDIB(IDB_SPRITE);
   m_pbkgnd = new CDIB(IDB_BACKGROUND);

   // Set the scrolling to match the size of the background DIB
   sizeTotal.cx = m_pbkgnd->GetWidth();
   sizeTotal.cy = m_pbkgnd->GetHeight();
   SetScrollSizes(MM_TEXT, sizeTotal);


   // WROX 2: Prepare the back buffer DIB

   // Get the view's DC
   CDC* pDC = GetDC();

   CPalette *pOld = pDC->SelectPalette(m_pbkgnd->GetPalette(), TRUE);
   pDC->RealizePalette();

   // Create the back-buffer bitmap based on the view's DC and the
   // background DIB
   m_poffscr = new CDIB(pDC,m_pbkgnd);

   pDC->SelectPalette(pOld,TRUE);
   pDC->RealizePalette();

   ReleaseDC(pDC);

   // Start the sprite animation timer
   setTimer();
```

```
   m_frame = 0;
   m_xpos = 0;
   m_xvel = 2;

   RenderScene();
}
```

In the first part of the code, denoted by **WROX 1** in the comment, we create **CDIB** objects for the sprite bitmap and the background using the first **CDIB** constructor we saw earlier. The **CDIB** constructor automatically retrieves the data for the bitmap from the resource identified by the first argument in each case, since the second argument is **FALSE**. In the second part of the code indicated by **WROX 2** in the comment, we create another **CDIB** object using the second **CDIB** constructor. This creates the bitmap using the pointer to the DC passed as the first argument, and the background **CDIB** object accessed through the pointer passed as the second argument. Finally, after starting the timer, the initial values for the current sprite frame number, the initial position offset, and the velocity of the sprite.

Step 3

There are three functions involved in actually drawing the scene. **RenderScene()** assembles the scene in the back buffer, the **OnDraw()** handler blits the back buffer to the screen DC, and the **OnTimer()** handler triggers re-rendering of the scene every time the timer goes off.

```
void CAnimTestView::RenderScene()
{
    CDC dcOSB;

   // Grab the view's DC
   CDC* pDC = GetDC();

   // Create memory DCs compatible with the screen DC
   dcOSB.CreateCompatibleDC(pDC);

   // Put the offscreen bitmap bits into a memory DC
   CBitmap* pbmOld2 = dcOSB.SelectObject(&(m_poffscr->GetBits()));

   // Calculate the x offset to the current sprite frame
   int frame_offset = m_frame*64;

   // Blit the background DIB onto the offscreen buffer
   m_pbkgnd->Draw(&dcOSB, 0,0, 0,0, m_pbkgnd->GetWidth(),
      m_pbkgnd->GetHeight());

   // Top row of blocks
   m_psprite->Draw(&dcOSB, 28+m_xpos, 30, frame_offset,0, 64, 62);
   m_psprite->Draw(&dcOSB, 188+m_xpos, 30, frame_offset,0, 64, 62);
   m_psprite->Draw(&dcOSB, 348+m_xpos, 30, frame_offset,0, 64, 62);

   // Bottom row of blocks
   m_psprite->Draw(&dcOSB, 28+m_xpos, 120, frame_offset,0, 64, 62);
   m_psprite->Draw(&dcOSB, 188+m_xpos, 120, frame_offset,0, 64, 62);
   m_psprite->Draw(&dcOSB, 348+m_xpos, 120, frame_offset,0, 64, 62);

   // Clean up
   dcOSB.SelectObject(pbmOld2);
   ReleaseDC(pDC);

   // Move the blocks over one tick
   m_xpos += m_xvel;

   if ((m_xpos > 200) || (m_xpos < 1))
      m_xvel = -m_xvel;

   // Advance the sprite frame, wrapping if necessary
   m_frame++;
   if (m_frame > 9)
```

```
      m_frame = 0;
}

void CAnimTestView::OnDraw(CDC* pDC)
{
   // Draw the back buffer to the screen
   m_poffscr->Draw(pDC, 0,0, 0,0, m_poffscr->GetWidth(),
      m_poffscr->GetHeight());
}

void CAnimTestView::OnTimer(UINT nIDEvent)
{
   // Render the next frame
   RenderScene();
   // Invalidate, but don't erase background
   Invalidate(FALSE);

   CScrollView::OnTimer(nIDEvent);
}
```

Step 4

The app needs to handle system palette messages and respond using the palette from the active view. To do this, we'll add a function to **`CMainFrame`** that tells the active view to realize its own palette, and we'll call it from **`OnPaletteChanged()`** and **`OnQueryNewPalette()`** handlers:

```
int CMainFrame::realizeActivePalette(CWnd* pFocusWnd)
{
   // Get the active view
   CAnimTestView* pView = (CAnimTestView*)GetActiveView();

   // Quit if this is the focuswnd   or if there is no active view
   if ((!pView)||(pView == pFocusWnd))
      return 0;


   // Realize the view's palette in the foreground
   return pView->realizePalette(FALSE);

}

void CMainFrame::OnPaletteChanged(CWnd* pFocusWnd)
{
   CFrameWnd::OnPaletteChanged(pFocusWnd);

   if (pFocusWnd != this)
      realizeActivePalette(pFocusWnd);
}

BOOL CMainFrame::OnQueryNewPalette()
{
   realizeActivePalette(this);

   return CFrameWnd::OnQueryNewPalette();
}
```

To make this work, we'll also have to implement the **`realizePalette()`** function for the view class:

```
int CAnimtestView::realizePalette(BOOL bForceBackground)
{

   // Get the palette from the offscreen dib
   CPalette* pPal = m_poffscr->GetPalette();

   // Realize the palette
```

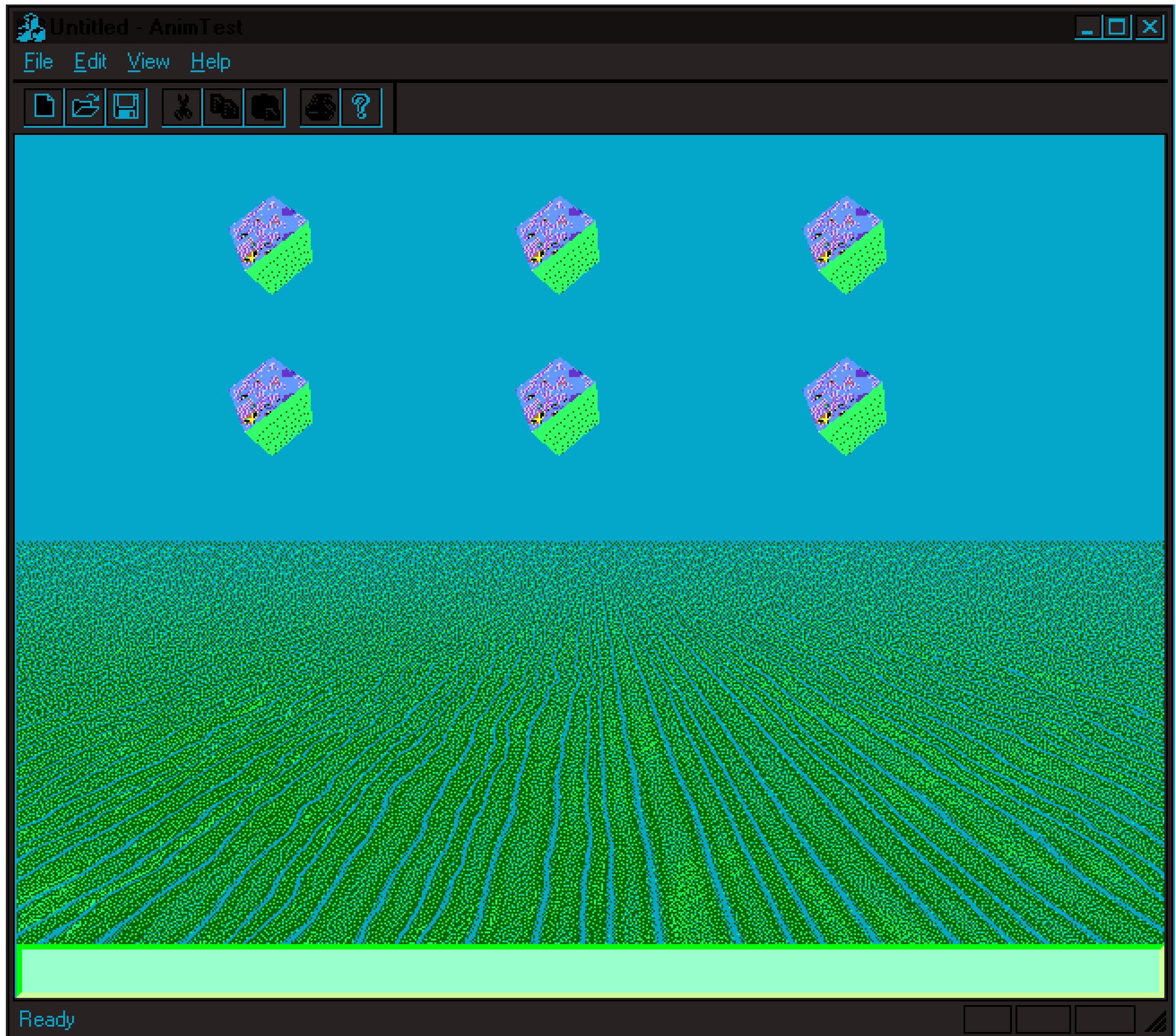
```
    CDC*   pDC = GetDC();

    CPalette* pOld= pDC->SelectPalette(pPal, bForceBackground);
    int result = pDC->RealizePalette();

    // bForceBackground is TRUE because palette should already
    // be mapped
    pDC->SelectPalette(pOld, TRUE);
    pDC->RealizePalette();

    ReleaseDC(pDC);

    // If realization changed, force repaint
    if (result)
       Invalidate();

    return result;

}
```

Step 5

Finally, we need to include the bitmaps for the animation graphics in the application's resources. We're going to use graphics borrowed from the DirectX examples later in the book.

Add two bitmaps to the resources, naming them **IDB_SPRITE** and **IDB_BACKGROUND** and set the Colors: general property to 256. Next, open `Sprite.bmp` in any paint package (Paint is fine), select the whole image, copy it to the clipboard and paste it into the **IDB_SPRITE** resource, resizing the resource when you do. Repeat this for `Bkgnd.bmp`, pasting it into **IDB_BACKGOUND**.

That's it. The finished app should look like this:



Summary

In this chapter, we've demystified two of the key elements of Windows multimedia programming: the device-independent bitmap, and the techniques for palette management. We've built a class to encapsulate what we learned, and built a simple DIB viewer with the new class. We also used the `CDIB` class to investigate double-buffered animation techniques.

The Windows 95 Game SDK (DirectX)

Virtually since its inception in 1981, MS-DOS has been the Intel world's platform of choice for game development. This isn't due to convenient game-programmer friendly features in MS-DOS, but rather to the fact that MS-DOS can be easily side-stepped or dumped altogether by a game program that wants to get at the raw hardware. Because it's hard to shove out of the way, Windows has long been shunned by the high-performance arcade game development community.

When Microsoft set out to change the operating system landscape with Windows 95, it had a challenge ahead of it. Since game developers were entrenched in the MS-DOS (or more precisely, non-Windows) world, and Microsoft wanted to de-emphasize MS-DOS in favor of Windows 95, it had to do something to entice those developers into Windows for the first time.

Enter the Windows 95 Game SDK, which provides access to a new kind of Microsoft API, **DirectX**. DirectX provides a sophisticated, high-performance mechanism for accessing display, audio, input and communications devices for use in real-time games written as native Windows apps. Yes, real, live and fairly conventional Windows apps! Sounds improbable? Read on.

The Design of a Simple Arcade Game: WROXBlox!

Over the course of this chapter, we're going to develop a DirectX game. To get an idea of the kinds of problems that game developers face, let's first talk a bit about the design of a simple game, in this case, the game we'll build, called WROXBlox! To understand the way the DirectX APIs are constructed, we need to see how they'll be used. When we've got the anatomy of the game laid out, we'll look at what kinds of system services we need in order to make it work.

The WROXBlox! Game Concept

The game we'll build will be a break-out style arcade game, in which the player controls a paddle (at the bottom of the screen) to direct a ball toward a bunch of blocks. The object is to knock out all of the objects with the ball without letting the ball fall down past the paddle. Clearing out the blocks ends a level, and game play restarts on a new level with a slightly faster ball. The player will get a certain number of balls, say three, and when they're all gone, the game is over. Pretty simple, right?

The Game Storyboard

.Actually, there's a bit more to this than meets the eye. Think about the arcade games that you've played in the past, I mean the big, stand-up games in a real arcade. When you play a simple game like the one we've just described, play usually progresses through a number of phases, or **states**:

Splash Screen—the stuff on the screen when you first walk up to it.

Level Start—when a level begins, you need a second or two to get oriented.

Play—the game action, when you're blasting aliens, or, in our case, knocking blocks

Pause—what happens when you pause the game (not an arcade feature, but a common one on PC games).

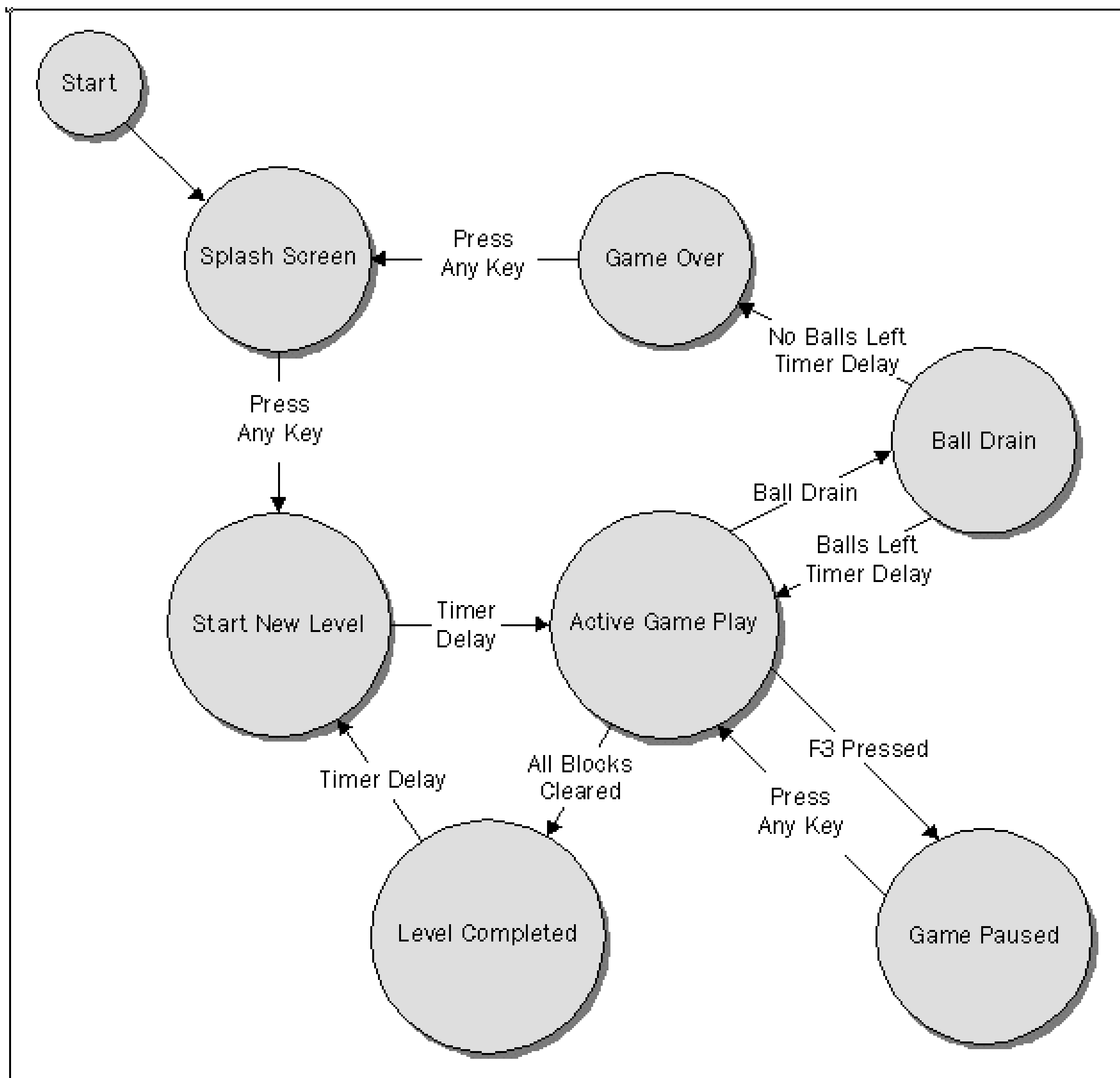
Drain—what happens when you lose a life/ship/ball/whatever.

Finish Level—the reward when you successfully clear out a level.

Game Over—what happens when you lose your last life/ship/ball/whatever.

Just about every shoot-em-up arcade game goes through these states, and sometimes a few more (like the cartoons in

Pac-Man). These states fit together in a typical flow, which can easily be expressed as a simple finite-state machine:



Now let's take a look at these states in the context of WROXBlox! and what we want to do in each one.

The Splash Screen State

The splash screen is to a video game what the cover is to a book; it usually shows the game's logo, copyright information, creators and maybe a few characters or scenes from the game itself. The splash screen is the initial state for the game. When the game first starts, it goes into this state, and it returns here whenever a game session ends.

When we're in the Splash Screen state in WROXBlox!, we'll want to do the following things:

- Paint the Splash Screen bitmap onto the display.
- Wait for the player to press any key, which takes us to the Level Start state.

The Level Start State

It would be a bit disconcerting to our players if we dropped them right into the game action without warning after

they have pressed a key to exit the Splash Screen state. We need to give them a second or two to reach for the controls and steel their nerves. To give them a hint of what's in store, we'll draw the game backdrop and anything else we deem appropriate, plus a warning to get ready.

Here's what we'll do in the WroxBlox Level Start state:

Paint the game backdrop on the screen.

Paint a 'Get Ready!' message over it.

Set a timer for a second or so. When the timer expires, start the game action by going to the Play state.

The Play State

While we're in the Play state, the player and the game are battling it out. There's an important concept to grasp here, pertaining to the timing of what happens in the Play state. While the game is in play, a single sequence of actions will be repeated over and over, many times per second. We'll call one pass through this sequence of actions a **game slice**. A game slice is like a round of turns in a board game, where every player gets a turn. During a game slice, the following stuff has to happen:

Check for input from the user (via keyboard or joystick).

Move and animate all sprites as needed. (**Sprites** are all of the little moving objects in the game—the blocks, the ball, the paddle.)

Check for collisions between sprites and act accordingly. For instance, if the ball strikes a block, blow it up and score the hit.

Watch for ball drain. If the ball moves off the bottom of the screen, go to the Ball Drain state.

Watch for the Pause key. If the user presses it, go to the Pause state.

If all the blocks are cleared out, go to the Finish Level state.

Draw everything in its new state and place.

Repeating these steps over and over again gives us the game's action.

The Pause State

When the phone rings in the middle of your WROXBlox! game, you'll want to put the game on hold while you take the call. We'll make the *F3* key pause the action. Here's what happens when WROXBlox! is in the Pause state:

Draw everything in its current state, but don't move or animate anything.

Draw a 'Paused' message on top to indicate that the game is paused.

Wait for the player to press any key.

The Ball Drain State

The main object of WROXBlox! is to keep the ball in play. If, heaven forbid, the ball should slip past the player's paddle, we have to acknowledge that fact. What we do is up to us (making a rude noise of some sort seems appropriate). Also, we should delay the game for a second or two to let the loss sink in. So, here's the drill for the Ball Drain state:

Make a rude noise.

Set a timer and keep the game animation going for a second or so with no ball, to taunt the player.

When the timer expires, subtract a ball, reset the ball position and go back to the Play state, or if all balls are gone, go to the Game Over state.

The Finish Level State

If the player manages to clear out a whole level, they deserve a pat on the back and a quick breather. In the Finish Level state, we will:

Paint the game backdrop.

Paint a 'Nice Work!' message over it.

Set a timer for a second or so, and when it expires, reset all sprites, up the difficulty slightly, and go back to the Level Start state.

The Game Over State

No arcade game would be complete without the idiomatic 'Game Over' message that appears when the player makes that final mistake. Our Game Over state will:

Paint the game backdrop.

Paint a 'Game Over' message over it.

Wait for the player to press any key, at which point we go back to the Splash Screen state.

Game Program Architecture

OK, the game concept is pretty well laid out. Now, how do we go about translating it into a program? Well, let's lay out some pseudocode for the state machine we've just described and add some start-up and shutdown stuff to it:

```
Initialize any devices
Load and  initialize the game's media resources - bitmaps/sprites, sounds, etc.

while (no request to quit)
{
   switch (game state)
   {
      case Splash Screen:
         // Paint the splash screen bitmap onto the display.
         // Wait for the player to press any key.
         // When a key is pressed go to the Level Start state.
         break;

      case Level Start:
         // Paint the game backdrop on the screen.
         // Paint a "Get Ready!" message over it.
         // Set a timer for a second or so.
         // When the timer expires, go to the Play state.
         break;

      case Play:
         // Game slice:
         // Check for input from the user (via keyboard or joystick).
         // Move and animate all sprites as needed.
         // Check for collisions between sprites and act accordingly.
         // Watch for ball drain -
         //    if the ball drains, go to the Ball Drain state
         // Watch for the Pause key - if pressed, go to the Pause state
         // Watch for finished level -
         //    if all the blocks are cleared, go to the Finish Level state
         // Draw everything in its new state and place.
         break;

      case Pause:
         // Draw everything, but don't move or animate anything.
         // Draw a "Paused" message on top.
         // Wait for the player to press any key.
         break;

      case Drain:
         // Make a rude noise.
         // Set a timer.
         // Draw and animate everything except ball.
         // When the timer expires:
         // subtract a ball
         // reset the ball position
         // go back to the Play state, or
         // if all balls are gone, go to the Game Over state.
         break;

      case Finish Level:
```

```
         // Paint the game backdrop.
         // Paint a "Nice Work!" message over it.
         // Set a timer for a second or so.
         // When timer expires:
         //    reset all sprites
         //    up the difficulty slightly
         //    go back to the Level Start state
         break;

      case Game Over:
         // Paint the game backdrop.
         // Paint a "Game Over" message over it.
         // Wait for the player to press any key.
         // Go back to Splash Screen state.
         break;
   }
}

Deinitialize devices and clean up media resources
```

The DirectX API

We've now gone about as far as we can with WROXBlox! without getting into DirectX. So far, everything we've said about our example program applies to game programming on just about any platform. Now we need to see how to write WROXBlox! As a Windows app with DirectX. The DirectX API will provide the mechanisms we need to do the things we've specified in our game.

DirectX is made up of several subordinate APIs, each for handling a different aspect of game device control:

- **DirectDraw** provides APIs for directly controlling your video display hardware.
- **DirectSound** provides APIs for directly controlling your audio hardware.
- **DirectInput** provides APIs for querying joystick and other multi-axis game controllers.
- **DirectPlay** provides APIs for easily communicating over networks and modems in a protocol-independent fashion for multiplayer games.

A Note about DirectX and COM

The DirectDraw, DirectSound, and DirectPlay APIs are all built atop the Component Object Model (COM). Yes, this is the same COM that forms the basis for OLE 2. Wait! Don't glaze over just yet. It's not nearly as bad as you think. The philosophy behind choosing COM for DirectX was to provide operating-system-level object wrappers for the raw hardware devices handled by these APIs. Rather than making it hard to use, this actually makes DirectX elegantly simple. Note that COM isn't OLE. It's simply an interface specification model for object-oriented APIs. In fact, all of the hard stuff about COM is hidden from you as a DirectX programmer. You make simple function calls to create the COM objects that wrap the devices, and pointers to those objects are returned to you to use. You can then treat them as if they were C++ objects with the appropriate DirectX API functions as members! That's all there is to it, as we'll see.

When you use the DirectX API, you'll create one COM object to wrap around your video adapter (an **`IDirectDraw`** object), another to wrap around your sound board (an **`IDirectSound`** object), and yet another to handle multiplayer communications across an chosen medium for (an **`IDirectPlay`** object). Then you call member functions on those objects to make things happen. Some of these objects need other, subordinate objects in order to do their stuff (for instance, your **`IDirectDraw`** object will use palette objects and drawing surface objects). The master object also provides everything necessary to create and work with the subordinate objects. Note that the one exception to this is the DirectInput API; as yet, there isn't a COM interface for DirectInput. In fact, there's hardly a DirectInput API at all. It's really just the **`joyGetPosEx()`** function from the MMSYSTEM API.

DirectDraw

Let's face it, for games, GDI just doesn't cut it. Microsoft's first answer to this problem was WinG, which was a nice first step, but hardly a complete solution. What you really want as a game programmer is to have low-level access directly to the video adapter on the target platform.

The DirectDraw API gives you an object-oriented way to access the features of your video adapter. DirectDraw provides mechanisms for blitting directly into and out of the memory resident on the video board. It allows you to allocate memory into blocks, called **surfaces**, onto which graphics can be rendered off-screen. Then, these surfaces can be 'flipped' onto the display through another DirectDraw call. Surfaces can be set up to store pieces of graphic material that will be used to assemble the game display image (sprites, backdrops, etc.), either in the video adapter RAM (if there's room) or in system RAM. Also, you can easily swap or alter color palettes to get all kinds of interesting effects.

Another thing to note is that, whenever possible, DirectDraw tries to use your video hardware to perform its operations, but also provides emulation in software for features not supported by a particular board.

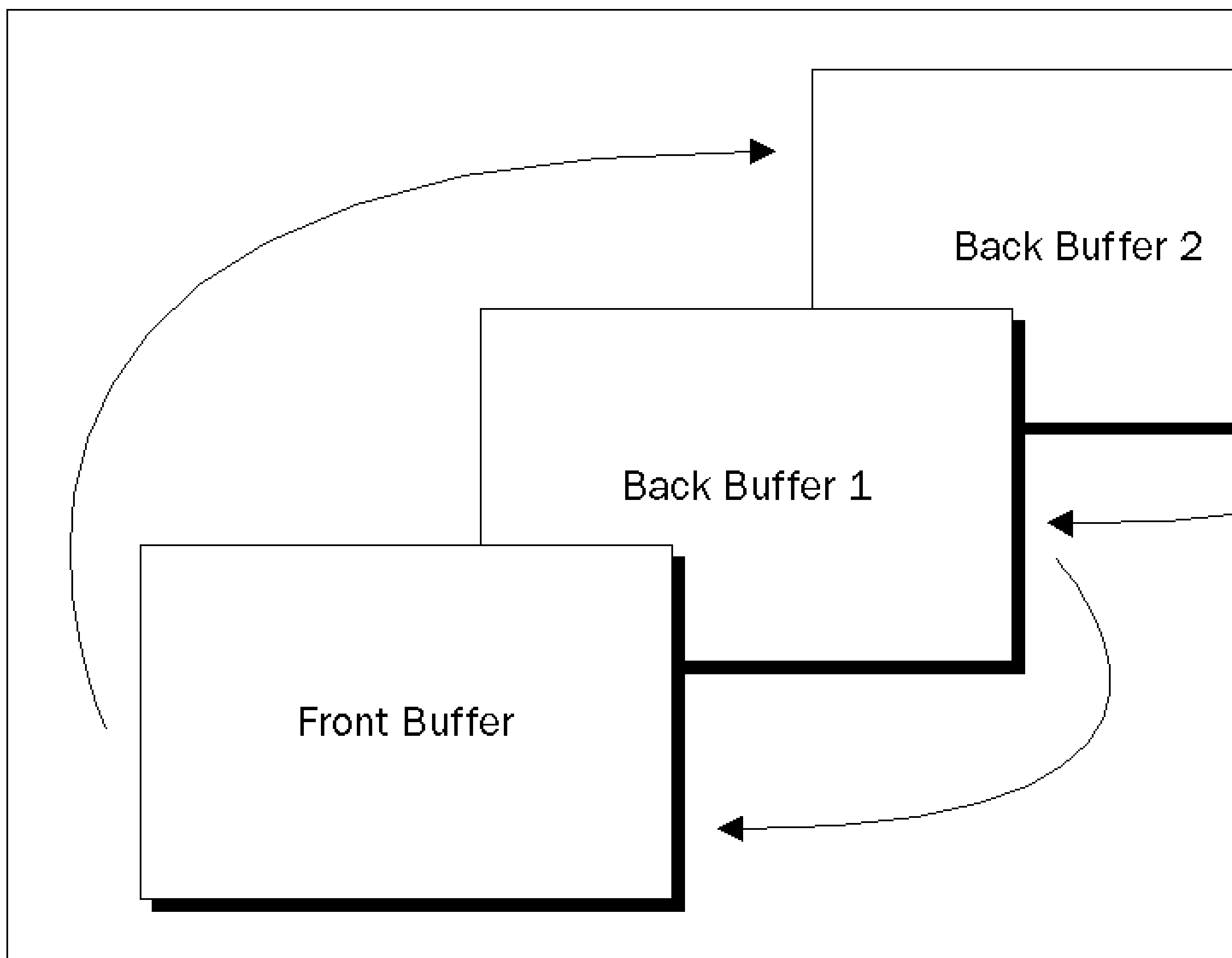
Basically, DirectDraw does just about everything you need to make game graphics hum, without making you learn all that weird VGA register stuff. Pinch yourself; we're still talking about writing Windows apps in C++, not DOS apps in assembler, and we're going to be able to do all this!

Off-screen Drawing and Surfaces

Good, smooth game animation works like film, in that complete images, or **frames**, are sequentially flashed on the screen to make up the action. To accomplish this, you really need to be able to assemble an entire frame image in memory, while the previous frame is being displayed, a technique called **double-buffering**. Without double-buffering, the user gets to watch the assembly on-screen, which is considered unacceptable in commercial gaming circles (the technical term is, I believe, **lame**). Any game API that is going to be taken seriously must support this ability. Better yet, for added speed, it should allow the memory resident on board the video adapter to be used for this purpose. DirectDraw does all of this and more.

In many ways, you can think of DirectDraw as a fancy object-oriented memory manager for the RAM on your video board. DirectDraw allows you to allocate blocks of video board memory, called **surfaces**, for use by your program. Surfaces are used for storing bitmap data. Storing bitmap data directly in the video adapter's RAM makes it available to the video hardware for display and accelerated graphics operations like hardware blitting and page flipping.

Typically, there are two kinds of surfaces that you might use in a DirectDraw app: the **flipping surface** (which represents the actual display) and **off-screen surfaces** for storing and staging scene components. The flipping surface is special because it's made up of two or more surfaces, rather than just one. These surfaces are the front buffer and one or more back buffers. All of the surfaces in a flipping surface are created at the same time, with one call, and are called **implicit surfaces** because of this. You assemble your scene on a back buffer, and when you're ready to show it, you tell DirectDraw to 'flip' it to the front. This rotates the order of the implicit surfaces, making the previous front buffer a back buffer and advancing the order of any other back buffers. Imagine a flip-book animation mounted on a Rolodex and you'll get the idea.



DirectDraw usually just has to change a hardware register on the video board to point to the right block of surface memory in order to do a flip, so this is a very fast operation. In WROXBlox!, we'll use just two implicit surfaces (front, plus one back), which is a common configuration.

You'll create one flipping surface group in your app, and as many off-screen surfaces as you need (and video memory will hold). In WROXBlox!, there will be two off-screen surfaces: one to hold all the game elements (backdrop, sprite cels, etc.) and another to hold the splash panel graphic.

Palettes

DirectDraw is usually used in 8-bits-per-pixel mode, where the values in memory for each pixel represent not absolute colors, but IDs of entries in a 256-color palette. The palette entry maps to the actual display color. Palettes are loaded and handled independently of bitmaps, allowing all kinds of palette-manipulation tricks. If you've had any exposure to the palette-management hassles normally facing Windows programs, you'll be relieved to discover that DirectDraw apps generally have the display palette (and, in fact, the entire display adapter) to themselves. Working with palettes in this kind of environment should be a welcome change, with no palette reconciliation woes.

For a more detailed explanation of palettes have a look at Chapter 12, Working with DIBs.

Setting up DirectDraw

Let's look at how to actually get DirectDraw up and ready for use. There are several basic steps here:

Create the main window for your app. You need to do this before you initialize DirectDraw, because some of the DirectDraw operations will reference it.

Create the **IDirectDraw** object for the app.

Get exclusive access to the video hardware.

Select a video mode.

Create the front and back buffers.

Create any additional surfaces you want.

Set the palette for the front buffer.

Creating the DirectDraw Object

Since all other DirectDraw operations will be done through the one and only **IDirectDraw** object for our app, we'll need to create it first. The function, **DirectDrawCreate()**,handles all of the messy details of this for us. We hand it a pointer variable, and it fills it with a pointer to a new **IDirectDraw**. If there's a problem, it will be reported in the return value. Here's the specification for the function:

```
HRESULT DirectDrawCreate( GUID FAR * lpGUID, LPDIRECTDRAW FAR *lplpDD,
   IUnknown FAR *pUnkOuter )
```

Parameter	Meaning
lpGUID	GUID of the driver to use; usually **NULL**, which tells DirectDraw to use the active display driver.
LplpDD	Points to the pointer variable to be filled with the pointer to the new DirectDraw object.
PUnkOuter	For future use. Must be **NULL**.

So, the creation of the DirectDraw object looks something like this:

```
LPDIRECTDRAW   lpMyDDObject;
HRESULT      dderr;

dderr = DirectDrawCreate(NULL, &lpMyDDObject, NULL);
```

There are several ways **DirectDrawCreate()** can fail:

Error Result Code	Meaning
DDERR_INVALIDPARAMS	A bad parameter was given to the function.
DDERR_INVALIDDIRECTDRAWGUID	An invalid driver ID was specified.
DDERR_GENERIC	Unspecified error. Call Bill Gates.
DDERR_OUTOFMEMORY	Not enough system memory to complete the operation.
DDERR_NODIRECTDRAWHW	The driver doesn't support the hardware operations required by the DirectDraw object.
DDERR_DIRECTDRAWALREADYCREATED	The app has already created a DirectDraw object for this driver. You can only have one at a time.
DD_OK	No error.

Basically, anything except **DD_OK** is fatal, and you can't continue with the app.

SetCooperativeLevel()

Once you've got a valid **IDirectDraw** object, you need to configure the display hardware to fit your needs for your game. The first thing you'll want to do is to get its undivided attention, because you'll want to seize full control of the display away from any other processes (including Windows) while your game is running. The **SetCooperativeLevel()** member function on the **IDirectDraw** object accomplishes this:

```
HRESULT IDirectDraw :: SetCooperativeLevel(HWND hWnd, DWORD dwFlags)
```

hWnd is the **HWND** of your app's main window. The **dwFlags** parameter is used to indicate the degree of exclusivity you wish to impose over the display hardware. The most common flags are:

Flag	Meaning
DDSCL_EXCLUSIVE	The app requests exclusive use of the display.
DDSCL_FULLSCREEN	The app will be responsible for the entire primary display surface.
DDSCL_ALLOWREBOOT	The app will allow a *Ctrl-Alt-Delete* operation to occur.
DDSCL_NOWINDOWCHANGES	The app can't be minimized or switched away.
DDSCL_NORMAL	Allows the app to function as a normal Windows app.

The operation might look like this:

```
HRESULT   dderr;

dderr = lpMyDDObject->SetCooperativeLevel (hMainWnd,
   DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN |
   DDSCL_ALLOWREBOOT | DDSCL_NOWINDOWCHANGES);
```

Once again, there are several ways that this could fail:

Error Result Code	Meaning
DDERR_INVALIDOBJECT	The DirectDraw object is invalid.
DDERR_INVALIDPARAMS	A bad parameter was passed to the function.
DDERR_EXCLUSIVEMODEALREADYSET	The app already has exclusive mode set.
DDERR_OUTOFMEMORY	There isn't enough system memory to complete the operation.
DDERR_HWNDALREADYSET	The **HWND** has already been set and can't be changed once surfaces have been created.
DDERR_HWNDSUBCLASSED	The **HWND** passed to **SetCooperativeLevel()** must not be subclassed. Subclassing prevents DirectDraw from properly restoring states.
DD_OK	No error.

Once again, anything other than **DD_OK** is fatal.

Setting Video Modes

Now you've got the display all to yourself. The next thing you'll want to do is to force the resolution and color depth of the display to fit your game's needs. The most common DirectX display resolution (probably because so many adapters support it, and it's not too big to be slow) is 640x480x8-bits-per-pixel. This gives you a standard VGA resolution playfield with a respectable 256-color palette to play with. The **SetDisplayMode()** member function does the job:

```
HRESULT IDirectDraw :: SetDisplayMode(DWORD dwWidth, DWORD dwHeight,
   DWORD dwBpp)
```

Note that, prior to making this call, we must have called **SetCooperativeLevel()** with **DDSCL_EXCLUSIVE**, or we'll get an error result (**DDERR_NOEXCLUSIVEMODE**).

dwWidth and **dwHeight** are the dimensions of the resolution you want (like 640 x 480) and **dwBpp** is the number of bits-per-pixel (like 8). In practical use, it looks something like this:

```
HRESULT   dderr;

dderr = lpMyDDObject->SetDisplayMode(640, 480, 8);
```

This call can return the following:

Error Result Code	Meaning
DDERR_INVALIDOBJECT	The DirectDraw object is invalid.
DDERR_INVALIDPARAMS	A bad parameter was passed to the function.
DDERR_GENERIC	Unspecified error. Call Bill Gates.
DDERR_UNSUPPORTED	This action isn't supported.
DDERR_INVALIDMODE	The requested resolution or mode isn't allowed.
DDERR_LOCKEDSURFACES	Mode can't be changed when surfaces are locked.
DDERR_WASSTILLDRAWING	You can't do this when a blit is still in progress.
DDERR_SURFACEBUSY	Another process is using the surface.
DDERR_NOEXCLUSIVEMODE	Cooperative level must be exclusive to perform this action.
DD_OK	No error.

This call failing isn't necessarily fatal to the app, but anything other than **DD_OK** means that the mode change didn't occur.

Creating Surfaces

Next, we need to set up a flipping surface, to give us something to draw on, and some off-screen surfaces to hold any bitmap data for sprites or scenery that we need to assemble the game frames.

All the surfaces are created by calling the same **IDirectDraw** member function, **CreateSurface()**, which looks like this:

```
HRESULT IDirectDraw::CreateSurface( LPDDSURFACEDESC lpDDSurfaceDesc,
   LPDIRECTDRAWSURFACE FAR *lplpDDSurface, IUnknown FAR *pUnkOuter)
```

Parameter	Meaning
lpDDSurfaceDesc	Points to a **DDSURFACEDESC struct** that describes the surface to be created.
lplpDDSurface	Points to a pointer variable that will point to the new surface if all goes well.
pUnkOuter	For future use. Must be **NULL**.

The function returns one of the following:

Error Result Code	Meaning

DDERR_INVALIDOBJECT	The DirectDraw object is invalid.
DDERR_INVALIDPARAMS	A bad parameter was passed to the function.
DDERR_NOEXCLUSIVEMODE	Cooperative level must be exclusive to perform this action.
DDERR_OUTOFVIDEOMEMORY	There's not enough video memory to complete the operation.
DDERR_NODIRECTDRAWHW	A hardware-only operation was attempted on an emulation-only driver.
DDERR_NOCOOPERATIVELEVELSET	**SetCooperativeLevel()** not called before this operation.
DDERR_INVALIDCAPS	Incorrect device capabilities were specified.
DDERR_INVALIDPIXELFORMAT	An invalid pixel format for the device was specified.
DDERR_NOALPHAHW	An alpha-channel operation was requested, but no alpha-acceleration hardware is installed.
DDERR_NOFLIPHW	The device doesn't support surface flipping.
DDERR_NOZBUFFERHW	The device doesn't support Z-buffer blitting.
DDERR_OUTOFMEMORY	There's not enough system memory available to complete the operation.
DDERR_PRIMARYSURFACEALREADYEXISTS	A primary surface has already been created by this process.
DDERR_NOEMULATION	There's no software emulation installed.
DDERR_INCOMPATIBLEPRIMARY	The request to create primary surface doesn't match the existing primary surface.
DD_OK	No error.

There's all kinds of junk in the **DDSURFACEDESC struct**, but, for now, here's how we need to set it up to make a flipping surface:

```
DDSURFACEDESC         sdesc;          // Surface description struct
LPDIRECTDRAWSURFACE   lpDDSPrimary;   // Pointer to the new front surface

//
// Create the front flipping surface with 1 back buffer
//

// dwSize field: always set to the size of the DDSURFACEDESC struct
sdesc.dwSize = sizeof( DDSURFACEDESC );

// dwFlags field: indicates which other fields hold valid info
sdesc.dwFlags = DDSD_DDSCAPS | DDSD_BACKBUFFERCOUNT;

// ddsCaps field: this is a struct with one field, dwCaps: holds flags
// describing the kind of surface to make
// In this case, we're making the front (primary) flipping surface,
// which is a complex surface (has implicit surfaces)
sdesc.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE | DDSCAPS_FLIP |
   DDSCAPS_COMPLEX;

// Flipping surface will have one back buffer
sdesc.dwBackBufferCount = 1;

// Make it
lpMyDDObject->CreateSurface( &sdesc, &lpDDSPrimary, NULL );
```

By the end of all this, we've got our flipping surface and back buffer. Now let's make a surface to hold off-screen image data. This requires another call to **CreateSurface()**, but this time with the **DDSCAPS_OFF-SCREENPLAIN**

flag. Let's say we want a 100x200 off-screen surface:

```
DDSURFACEDESC          sdesc;             // Surface description struct
LPDIRECTDRAWSURFACE    lpDDSOffscreen;    // Pointer to the new offscreen
                                          // surface

// Size is always size of the DDSURFACEDESC struct
sdesc.dwSize = sizeof(DDSURFACEDESC);

// Caps, height, and width fields are valid
sdesc.dwFlags = DDSD_DDSCAPS | DDSD_HEIGHT |DDSD_WIDTH;

// This is going to be an offscreen surface
sdesc.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;

// Set the width and height
sdesc.dwWidth = 100;
sdesc.dwHeight = 200;

// Create it
lpMYDDObject->CreateSurface(&sdesc, &lpDDSOffscreen, NULL);
```

Usually, you'll use off-screen surfaces to hold bitmap data loaded from resources or files. Microsoft provides some handy utility functions with the DirectX SDK to make it easier to work with bitmap resources in your game. These functions live in the `Ddutil.h` and `Ddutil.cpp` files in the `Samples\Misc` directory of the SDK. One of these functions, **`DDLoadBitmap()`**,reads in a bitmap from a file or from the application's resource data, creates an off-screen surface big enough to hold it, copies the bitmap into the surface and returns a pointer to the new surface. That's just what we need. Here's the spec for the **`DDLoadBitmap()`** function:

```
#include "ddutil.h"

LPDIRECTDRAWSURFACE DDLoadBitmap(IDirectDraw *pdd, LPCSTR szBitmap,
   int dx, int dy)
```

Parameter	Meaning
pdd	The pointer to the main DirectDraw object.
SzBitmap	The name of the bitmap resource or file from which to load.
dx	The width of the image to load; usually 0, which tells **`DDLoadBitmap()`** to get it from the file.
dy	The height of the image to load; usually 0, which tells **`DDLoadBitmap()`** to get it from the file.

To actually draw things, however, we'll pretty much always be working with the back buffer, not the front buffer, to which we now have a pointer. So, we need to get a hold of the back buffer, which is an implicit surface of the front buffer. The **`GetAttachedSurface()`** member of **`IDirectDrawSurface`** will do the trick:

```
HRESULT IDirectDrawSurface::GetAttachedSurface(LPDDSCAPS lpDDSCaps,
   LPLPDIRECTDRAWSURFACE FAR *lplpDDAttachedSurface)
```

Parameter	Meaning
lpDDSCaps	Flags to indicate what to do.
lplpDDAttachedSurface	Points to a pointer to be filled with the address of the attached surface.

The function returns one of the following:

Error Result Code	Meaning
DDERR_INVALIDOBJECT	The DirectDraw object is invalid.
DDERR_INVALIDPARAMS	A bad parameter was passed to the function.

DDERR_SURFACELOST	The surface memory has been released (probably by another process). **Restore()** must be called to regain access to the surface.
DDERR_NOTFOUND	No such item exists.
DD_OK	No error.

And here's what we have to do:

```
DDSCAPS              ddscaps;     // Surface capabilities struct
LPDIRECTDRAWSURFACE  lpDDSBack;   // Pointer to the new back surface

// Get a pointer to the back buffer - this is the one we'll draw on
ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
lpDDSPrimary->GetAttachedSurface(&ddscaps, &lpDDSBack);
```

That takes care of the flipping surface.

Palettes

You control the color in your game by setting the palette associated with the front flipping surface. Keep in mind that any graphics used to assemble images on the back buffer will eventually wind up on the front buffer, so the current palette also applies in a sense to those surfaces. When you start up your game, you'll need to set the initial palette used by the front buffer, and you can change the palette at any time. In WROXBlox! we'll use a different palette for the splash screen than for game play, so whenever we put up or take down the splash screen, we'll need to change the palette.

To set the palette used by the front buffer, we call the **IDirectDrawSurface:: SetPalette()** function:

```
HRESULT IDirectDrawSurface::SetPalette (LPDIRECTDRAWPALETTE lpDDPalette)
```

lpDDPalette is a pointer to the **IDirectDrawPalette** to use. But how do you get one of those in the first place? Typically, the palette you'll want to use is the palette from the bitmap that contains the artwork for your game. Bitmap files (and resources) contain palette information, and you'll want to read that out and wrap an **IDirectDrawPalette** around it. Once again, there's a convenient function for this in Ddutil.h:

```
LPDIRECTDRAWPALETTE DDLoadPalette (IDirectDraw *pdd, LPCSTR szBitmap)
```

Parameter	Meaning
pdd	A pointer to the main **IDirectDraw** object for the app.
szBitmap	The name of the bitmap resource or file from which to load the palette.

In WROXBlox!, we'll load the artwork and splash screen palettes at startup and switch them using **SetPalette()** as needed.

Drawing onto Surfaces

Now you have everything you need to draw on the back buffer.

Blt() and BltFast()

To achieve reasonably complex scenes at high speed, rather than taking time to render objects from scratch each time, most games blit prerendered scene components onto each frame. DirectDraw is centered around this principle, and provides two kinds of blitter support. The **IDirectDrawSurface::Blt()** function is a general-purpose blitter

with all kinds of options and effects, while **`IDirectDrawSurface::BltFast()`** is a no-nonsense, stripped-down blitter, optimized for plain rectangular copy and transparent blits. **`BltFast()`** is only faster than **`Blt()`** when software-emulation is in effect. The complexities of **`Blt()`** are beyond the scope of this chapter, and we'll only be using **`BltFast()`** in WROXBlox!, so here's the scoop on it:

```
HRESULT IDirectDrawSurface:: BltFast(DWORD dwX, DWORD dwY,
   LPDIRECTDRAWSURFACE lpDDSrcSurface, LPRECT lpSrcRect,
   DWORD dwTrans)
```

Parameter	**Meaning**
`dwX`	The destination X coordinate.
`dwY`	The destination Y coordinate.
`lpDDSrcSurface`	The surface from which to copy bits.
`lpSrcRect`	The rectangle on the source surface from which to copy bits.
`dwTrans`	Flags. See below.

The flags can be one of the following values:

Flag	**Meaning**
`DDBLTFAST_SRCCOLORKEY`	Copy using source's color key.
`DDBLTFAST_DESTCOLORKEY`	Copy using destination's color key.
`DDBLTFAST_NOCOLORKEY`	Straight copy.
`DDBLTFAST_WAIT`	If hardware is busy, wait until it's available, then blit.

The function returns one of the following:

Error Result Code	**Meaning**
`DDERR_INVALIDOBJECT`	The DirectDraw object is invalid.
`DDERR_INVALIDPARAMS`	A bad parameter was passed to the function.
`DDERR_GENERIC`	Unspecified error. Call Bill Gates.
`DDERR_UNSUPPORTED`	This action isn't supported.
`DDERR_WASSTILLDRAWING`	You can't do this when a blit is still in progress. In your code, you should loop and retry this call until this condition clears (or another error occurs).
`DDERR_SURFACEBUSY`	Another process is using the surface.
`DDERR_SURFACELOST`	The surface memory has been released (probably by another process). You must call **`Restore()`** to regain access to the surface.
`DDERR_EXCEPTION`	The action tripped an exception.
`DDERR_INVALIDRECT`	A rectangle was specified incorrectly.
`DDERR_NOBLITHW`	No blitter hardware is installed.
`DD_OK`	No error.

Let's say we want to copy a 20x40 piece of an off-screen surface to the 100,200 on the back buffer:

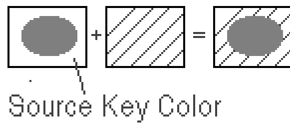
```
// Set up a RECT for the source area
```

```
RECT rSrc;
rSrc.top = 0;
rSrc.left = 0;
rSrc.bottom = 39;
rSrc.right = 19;

// Blit
lpDDSBack->BltFast(100, 200, lpDDSOffscreen, &rSrc,
   DDBLTFAST_SRCCOLORKEY | DDBLTFAST_WAIT);
```

Color Keys

What's all this about color keys and transparency? Here's the scoop. Basically, a key color is a transparent color. In other words, during transparent, or keyed, blits, pixels with a key color are not copied. The bits from the other surface 'show through' the key color after a transparent blit. Each surface can have color key values set for it that get used when the surface is involved in a transparent blit. You can specify a range of palette colors (a **color space**) to be a key, but Microsoft recommends that you use single key colors whenever possible.

There are two kinds of keys for each surface: **source** and **destination**. The source key specifies what pixels are transparent when the surface is being copied *from*. The destination key specifies which pixels can get copied over when the surface is being copied *to*.



By default, there are no keys and all pixels are fair game for copying and clobbering.
The **`IDirectDrawSurface::SetColorKey()`** function lets us specify a range of palette entries to key on:

```
HRESULT IDirectDrawSurface:: SetColorKey(DWORD dwFlags,
   LPDDCOLORKEY lpDDColorKey)
```

Parameter	Meaning
dwFlags	Flags to indicate the kind of key to set (see below).
lpDDColorKey	Pointer to a **DDCOLORKEY struct** that specifies the range of palette entries that make up the key

The **dwFlags** can be:

Flag	Meaning
DDCKEY_DESTBLT	The key specifies a destination key.
DDCKEY_SRCBLT	The key specifies a source key.
DDCKEY_COLORSPACE	The key is range of entries, not just one.

The return value is one of:

Error Result Code	Meaning
DDERR_INVALIDOBJECT	The DirectDraw object is invalid.
DDERR_INVALIDPARAMS	A bad parameter was passed to the function.

DDERR_GENERIC	Unspecified error. Call Bill Gates.
DDERR_UNSUPPORTED	This action isn't supported.
DDERR_WASSTILLDRAWING	You can't do this when a blit is still in progress. In your code, you should loop and retry this call until this condition clears (or another error occurs).
DDERR_SURFACELOST	The surface memory has been released (probably by another process). You must call **Restore()** to regain access to the surface.
DD_OK	No error.

The **DDCOLORKEY struct** just contains fields for the high and low end of the palette range to use as the key. If you're using a single palette entry as a key, set both of these fields to the ID of that entry. Let's say we want palette entry #255 to be the source color key for some off-screen surface:

```
DDCOLORKEY   ddkey;

ddkey.dwColorSpaceLowValue = 255;
ddkey.dwColorSpaceHighValue = 255;

lpDDSOffscreen->SetColorKey(DDCKEY_SRCBLT, &ddkey);
```

From now on, the color in palette entry 255 won't be copied from this surface in transparent blits.

Sprites

In a game, there are typically lots of little objects moving around on the screen at once—player characters, bad guys, bullets, rocks, shards of exploding things, etc. Managing all of these objects (animating them, moving them and checking for collisions between them) is a big part of the logic of many games. Having a nice, encapsulated way to deal with these objects would be great. In mythical terms, a **sprite** is a little magical being or spirit. In game programming, a sprite is a wrapper around one of those pesky little objects we've just been discussing, and you can use one to hide a lot of the tedium of handling these objects. Later on, we'll design and build a simple sprite around a piece of surface memory to make it easier to manage.

A sprite needs to keep track of its current position, its velocity, what piece of which surface contains its image data, how many animation cels it has, what cel it's currently displaying, and anything else we want to know about it. We need to be able to blit a sprite onto a surface, move it, flip to its next animation cel, check for collisions with other sprites, and hide it or show it.

DirectDraw comes into play with the sprite only when we go to draw it. We'll use **BltFast()** to copy the sprite's current image to the back buffer. Generally, since sprites can have strange shapes, we'll use transparency when we're blitting them.

If you've cheated and looked ahead in the DirectDraw documentation, you may have seen something in there called an **overlay**. Overlays provide a better way to deal with sprites than just using **BltFast()** from one surface to another. They use video hardware features to make special surfaces into hardware-controlled sprites that are managed entirely in the video hardware. That sounds great! Why, then, aren't we going to use overlays for sprites here? Because the current version of DirectDraw doesn't provide any HEL (hardware-emulation layer) support for overlays, which means we'd have to develop two kinds of sprites and check hardware capabilities to see if the target platform supports overlays. For our purposes, since **BltFast()** is plenty fast enough, that's more trouble than it's worth. Hopefully, Microsoft will include overlay emulation in a future release of the DirectDraw (or everyone will chuck their old video boards and upgrade to cards that support overlays).

We'll get into more details about sprites later in the chapter when we'll actually build a C++ sprite class.

Flipping the Primary Surface

Once your drawing on a back surface is complete, you'll want to flip it to the front for all the world to see (otherwise, what was the point of all this?). To do this, you use the **IDirectDrawSurface::Flip()** function:

```
HRESULT IDirectDrawSurface::Flip (LPDIRECTDRAWSURFACE
   lpDDSurfaceTargetOverride, DWORD dwFlags)
```

Parameter	Meaning
lpDDSurfaceTargetOverride	The surface to which to flip. If there is only one back buffer, or if you just want to flip to the next buffer in rotation, set this to **NULL**.
dwFlags	There is only one flag option: **DDFLIP_WAIT**.

Setting **dwFlags** to **DDFLIP_WAIT** causes the function to wait if the hardware is busy, until it's capable of flipping. If not set, **Flip()** fails and returns **DDERR_WASSTILLDRAWING** if the hardware was busy.

In practice, it looks like this:

```
// Tell the primary surface to flip
lpDDSPrimary->Flip(NULL, DDFLIP_WAIT);
```

Shutting Down DirectDraw

Eventually, all the valiant Earth forces will succumb to the menace of the galactic invaders, or all the balls will drain away, or maybe the boss will insist that whoever's playing your game actually gets some real work done, and it will be time to clean up and exit the app. Basically, all you need to do is call the **Release()** COM member function (no parameters) on your primary flipping surface (but not its implicit surfaces), any auxiliary surfaces, and the main **IDirectDraw** object, and you're done:

```
// Release the flipping surface
lpDDSPrimary->Release();

// Release the DirectDraw main object
lpMyDDObject->Release();
```

DirectSound

Macintosh users got a charge out of poking fun at PC gamers a few years back, before sounds boards were standard-issue. A great new Mac game had just been released, called Dark Castle, which featured sampled-audio sounds (a novelty back then). The joke going around was that Dark Castle had been ported to the PC, and you could hear the bats go 'beep', crashing thunder go 'beep', rolling barrels and dropping rocks go 'beep', etc. Now the tables have turned, and PCs frequently come equipped with sound boards that surpass the capabilities found in the Mac. When you're writing games, you'll want to tap into that hardware and make it sing, which is where DirectSound comes in. DirectSound lets you easily fire off PCM wave audio sounds from your game.

There are two kinds of COM objects to deal with in the DirectSound API: the main sound hardware wrapper, **`IDirectSound`**, and the wave audio buffer, **`IDirectSoundBuffer`**. **`IDirectSoundBuffer`** is little messy, but Microsoft have provided a serviceable wrapper around it, called **`HSNDOBJ`** (found in **`Samples\Misc\Dsutil.h`** provided with the GDK), which we'll use here to simplify things a bit.

Setting up DirectSound

Let's look at what's involved in setting up DirectSound for use.

Creating the DirectSound Object

The first order of business is to create an **`IDirectSound`** object, which will act as our base of operations for everything else we do with DirectSound:

```
HRESULT DirectSoundCreate(GUID FAR * lpGuid, LPDIRECTSOUND * ppDS,
   IUnknown FAR *pUnkOuter )
```

Parameter	**Meaning**
`lpGuid`	The GUID of the driver to use; usually **`NULL`**, which tells DirectSound to use the active sound driver.
`ppDS`	Points to the pointer which will receive the address of the new **`IDirectSound`** object.
`pUnkOuter`	For future use. Must be **`NULL`**.

Very simply, here's how it's typically called:

```
LPDIRECTSOUND    lpMyDSObject;

DirectSoundCreate(NULL, &lpMyDSObject, NULL);
```

Note that you must call **`SetCooperativeLevel()`** before you actually try to play sounds, or your app will fail.

Playing Sounds

Now we need to get some sounds set up to play, and then play them.

Creating Sound Objects

The object that represents an individual PCM audio sound in DirectSound is an **`IDirectSoundBuffer`**. In the interest of getting the quickest start possible with DirectSound, we're going to use helper functionality that Microsoft provides in the samples that come with the DirectX SDK (remember `Ddutil.h`?). This helper stuff lives in `Dsutil.h` and `Dsutil.cpp` in the `Samples\Misc` directory of the SDK. These files define a handy wrapper around **`IDirectSoundBuffer`**, including functionality to load buffers from `.wav` audio resources, which is something we'll need for the WROXBlox! app. This may seem like a cop-out, but, frankly, you can get quite far with the `Dsutil.h` functionality and never plunge into the depths of **`IDirectSoundBuffer`**. When we talk about panning, we'll have to briefly poke into **`IDirectSoundBuffer`**, though.

The `Dsutil.h` wrapper is based on a **struct**, called a **SndObj**. All the functions operate on this object instead of directly on **IDirectSoundBuffer**.

To load in a sound for use, we call **SndObjCreate()**:

```
#include "dsutil.h"  // dsutil.cpp needs to be compiled into your project

HSNDOBJ SndObjCreate(LPDIRECTSOUND pDS, LPCSTR szName, int nConcurrent)
```

Parameter	Meaning
pDS	The pointer to the app's main **IDirectSound** object.
szName	The name of the sound resource to load.
nConcurrent	The number of concurrent instances of this sound you expect to be able to play.

Here's how you might call it:

```
#include "dsutil.h"

HSNDOBJ hSnd;

hSnd = SndObjCreate (lpMyDSObject, "gunshot", 10);
```

This creates a **SndObj** from a wave resource, called 'gunshot', that we expect might be playing up to ten times simultaneously in the heat of game play.

Playing Sound Objects

One of the beauties of DirectSound is its ability to play sounds in the background asynchronously with no need for intervention or management on your part. You just tell it what to play, and it goes. Using the **SndObj** wrapper, we can play a sound with **SndObjPlay()**:

```
BOOL SndObjPlay(SNDOBJ *pSO, DWORD dwPlayFlags)
```

Parameter	Meaning
pSO	The **HSNDOBJ** to play.
dwPlayFlags	Flags that affect the playback. There is only one currently defined, **DSBPLAY_LOOPING**, which repeats the wave over and over,

Here's a typical call:

```
// Play the sound once
SndObjPlay (hSnd, 0);
```

Looping

If the **DSBPLAY_LOOPING** flag is set, the sound will play over and over until we explicitly tell it to stop with the **SndObjStop()** function:

```
BOOL SndObjStop(SNDOBJ *pSO)
```

pSO is the **HSNDOBJ** to squelch.

Panning

An advanced feature supported by **IDirectSoundBuffer** is **panning**, or setting the relative left-to-right location of the sound in the stereo pattern. Obviously, this feature only makes a difference when stereo sound is available, which is pretty typical these days.

Panning involves calling the **IDirectSoundBuffer** member function **SetPan()**. **SetPan()** takes one parameter, which is an integer ranging from -10,000 to +10,000. 10,000 pans the sound to the extreme left, 0 pans it to center, and +10,000 pans it to the extreme right. The value is in hundredths of a dB attenuation of the opposite channel. In other words, +4,000 means that the right channel is all the way on, and the left channel is attenuated by 40 dB.

Panning sounds can really enhance the audio in your game, but it was sadly left out of the **SndObj** wrapper. Luckily, it's pretty easy to get an actual **IDirectSoundBuffer** from a **SndObj**, using **SndObjGetFreeBuffer()**, and work with it:

```
// Get an actual IDirectSoundBuffer to play with
IDirectSoundBuffer *pDSB = SndObjGetFreeBuffer(hMySnd);

// Set panning slightly left
pDSB->SetPan(-1000);

// Play
pDSB->Play( 0, 0, 0);
```

Note that you can't use **SndObjPlay()***, as this plays havoc with the buffers.*

Shutting Down DirectSound

When the game is shutting down, you'll need to clean up DirectSound by calling **SndObjDestroy()** on each of your **HSNDOBJ**s, and by calling **Release()** on the **IDirectSound** object itself:

```
// Destroy game sounds
SndObjDestroy(hExplosionSnd);
SndObjDestroy(hGunshotSnd);
SndObjDestroy(hPlayerDieSnd);

// Release the DirectSound object
lpMyDSObject->Release();
```

DirectInput

In the first release of the DirectX API, the DirectInput API is just a repackaging of the Windows Multimedia System's joystick functions. There is currently no COM object interface for this API.

joyGetPosEx()

The main joystick function in DirectInput is **joyGetPosEx()**. This function has been greatly expanded over its multimedia system predecessor, **joyGetPos()**, to include support for up to 32 controllers, each with 32 buttons and 6 axes of control. Clearly, Microsoft has virtual-reality controllers in mind here. The joystick support in Windows 95 is quite sophisticated; Windows 95 actually handles the extents and calibration for you from the control panel, so your game rarely, if ever, will need to go through that hassle. The **joyGetPosEx()** function looks like this:

```
MMRESULT joyGetPosEx(UINT uJoyID, LPJOYINFOEX pji);
```

Parameter	Meaning
uJoyID	The ID of the joystick to query. **JOYSTICKID1** and **JOYSTICKID2** are predefined constants to use here.
pji	The pointer to a **JOYINFOEX struct** that passes flags into and results out of the call.

The `joyGetPosEx()` call returns one of the following:

Result	Meaning
JOYERR_UNPLUGGED	The requested joystick isn't plugged in (or an acoustic joystick is being used).
MMSYSERR_BADDEVICEID	The joystick ID given is invalid.
MMSYSERR_INVALPARAM	A bad parameter was passed to the call.
MMSYSERR_NODRIVER	There is no joystick driver installed.
JOYERR_NOERROR	Call succeeded.

The **JOYINFOEX struct** is where all the action takes place here. Let's look inside it:

```
struct JOYINFOEX {
    DWORD dwSize;
    DWORD dwFlags;
    DWORD dwXpos;
    DWORD dwYpos;
    DWORD dwZpos;
    DWORD dwRpos;
    DWORD dwUpos;
    DWORD dwVpos;
    DWORD dwButtons;
    DWORD dwButtonNumber;
    DWORD dwPOV;
    DWORD dwReserved1;
    DWORD dwReserved2;
};
```

Field	Purpose
dwSize	Always set to the size of the **JOYINFOEX struct**.
dwFlags	Flags indicating information requested or valid information returned in the structure.
dwXpos	The current X-axis (Axis 1) position.
dwYpos	The current Y-axis (Axis 2) position.
dwZpos	The current Z-axis (Axis 3) position.
dwRpos	The current rudder control (Axis 4) position.
dwUpos	The current Axis 5 position.
dwVpos	the current Axis 6 position.
dwButtons	The bitfield containing the states of the 32 possible buttons. A set of flag definitions, **JOY_BUTTON1** through **JOY_BUTTON32**, is provided to make reading these easier.
dwButtonNumber	The current button number that is pressed.
dwPOV	The position, in hundredths of a degree, of the point-of-view control (sometimes called the **hat**).
dwReserved1	Unused.
dwReserved2	Unused.

The principal flags that can be passed into the **dwFlags** field are:

Flag	Meaning

JOY_RETURNALL	Returns everything (except raw data).
JOY_RETURNBUTTONS	Returns the button state information.
JOY_RETURNCENTERED	Centers the neutral joystick position on each axis.
JOY_RETURNPOV	Returns the discrete position of the POV hat, as one of the following values: **JOY_POVBACKWARD** **JOY_POVCENTERED** **JOY_POVFORWARD** **JOY_POVLEFT** **JOY_POVRIGHT**
JOY_RETURNPOVCTS	Returns the position of the POV hat in hundredths of a degree.
JOY_RETURNR	Returns the rudder (Axis 4) position.
JOY_RETURNRAWDATA	Returns data in raw, uncalibrated form.
JOY_RETURNU	Returns the Axis 5 position.
JOY_RETURNV	Returns the Axis 6 position.
JOY_RETURNX	Returns the X-axis (Axis 1) position.
JOY_RETURNY	Returns the Y-axis (Axis 2) position.
JOY_RETURNZ	Returns the Z-axis (Axis 3) position.
JOY_USEDEADZONE	Makes the dead spot in the middle of each axis bigger. Returns the same value for all points in the dead zone.

Getting values back from a joystick controller is easy; just call **joyGetPosEx()** and pass it a **JOYINFOEX struct** with flags set that tell it what you want to know:

```
// Set up a JOYINFOEX, ask for X and Y position of joystick
JOYINFOEX   joyinfo;
joyinfo.dwFlags = JOY_RETURNX|JOY_RETURNY;

// Query joystick #1
joyGetPosEx(JOYSTICKID1, &joyinfo);

// the JOYINFOEX comes back with the requested values filled in
nJoyXPos = joyinfo.dwXpos;
nJoyYPos = joyinfo.dwYpos;
```

Checking for Joystick Availability

Since not all systems will have joysticks attached, it's a good idea for your game to check whether there is one before it drops the user into the game. The easy way to do this is to make a dummy call to **joyGetPosEx()** and check the return value:

```
// See if Joystick #1 is available
JOYINFOEX   joyinfo;
joyinfo.dwFlags = 0;

if (joyGetPosEx(JOYSTICKID1, &joyinfo) != JOYERR_NOERROR)
{
   // No joystick!
...
}
```

Querying Joystick Capabilities

If you want to know what features are supported by a joystick and what the ranges of the axes are, you can call the **joyGetDevCaps()** function:

```
MMRESULT joyGetDevCaps(UINT uJoyID, LPJOYCAPS pjc, UINT cbjc)
```

Parameter	Meaning
`uJoyID`	The ID of the joystick to query.
`pjc`	The pointer to a **`JOYCAPS struct`** to fill.
`cbjc`	The size of the **`JOYCAPS struct.`**

The **`JOYCAPS struct`** looks like this:

```
struct JOYCAPS {
    WORD wMid;
    WORD wPid;
    CHAR szPname[MAXPNAMELEN];
    UINT wXmin;
    UINT wXmax;
    UINT wYmin;
    UINT wYmax;
    UINT wZmin;
    UINT wZmax;
    UINT wNumButtons;
    UINT wPeriodMin;
    UINT wPeriodMax;
    UINT wRmin;
    UINT wRmax;
    UINT wUmin;
    UINT wUmax;
    UINT wVmin;
    UINT wVmax;
    UINT wCaps;
    UINT wMaxAxes;
    UINT wNumAxes;
    UINT wMaxButtons;
    CHAR szRegKey[MAXPNAMELEN];
    CHAR szOEMVxD[MAXOEMVXD];
};
```

Wow, that's a lot of information. The stuff you really care about, though, is this:

Field	Meaning
`wXmin`	The minimum X value.
`wXmax`	The maximum X value.
`wYmin`	The minimum Y value.
`wYmax`	The maximum Y value.
`wNumButtons`	The number of buttons supported by the joystick.
`wCaps`	The features implemented on the joystick:
	`JOYCAPS_HASZ` —has a 3rd axis.
	`JOYCAPS_HASR` —has a 4th axis.
	`JOYCAPS_HASU` —has a 5th axis.
	`JOYCAPS_HASV` — has a 6th axis.
	`JOYCAPS_HASPOV` —has a POV hat.
	`JOYCAPS_POV4DIR` —POV hat can return discrete position.
	`JOYCAPS_POVCTS` —POV hat can return continuous position (100ths of a degree).

Here's how you would use **joyGetDevCaps()** to get the X-axis extents of the joystick:

```
JOYCAPS   joycaps;

// Query the joystick caps
joyGetDevCaps(JOYSTICKID1, &joycaps, sizeof(JOYCAPS));

// Get the min and max X values
nJoyLeft = joycaps.wXmin;
nJoyRight = joycaps.wXmax;
```

You can also use Windows messages to handle the joystick, but **joyGetPosEx()** fits better with the game slice model and gives quicker and finer control.

A Note about Keyboard Control

Many great games (like Doom) still work quite well, or even best, when they are controlled from the keyboard, but DirectInput doesn't provide any keyboard input support. The best thing to use for that is the good old Windows API **GetAsyncKeyState()** function, which simply tells you whether the requested key is up or down:

```
int GetAsyncKeyState(int nKeyCode)
```

The **nKeyCode** parameter is the Windows virtual key code for the key to be tested. There is a virtual key code for each key on the keyboard. (Check the online help in Visual C++ under SDKs, Win32 SDK, Win32 Programmer's Reference, Appendices, Virtual-Key Codes for a full listing.) The main thing to remember about **GetAsyncKeyState()** is that the return value's most significant bit (MSB) indicates the instantaneous state of the requested key; 1 if it's currently down, 0 if it is up. So, the cheap trick for reading this state is to treat the MSB as a sign bit and check for negativity:

```
// Is the left-arrow key down?
If (GetAsyncKeyState(VK_LEFT) < 0)
{
   // Move ship to the left one click, etc.
...
}
```

Why not use the Windows key messages, like **WM_CHAR**, **WM_KEYDOWN** and **WM_KEYUP**? Well, for one thing, it's easier to integrate **GetAsyncKeyState()** processing into a game slice function, as we'll see later. If we use the messages, we'll have to keep track of key states between messages ourselves. It's just much cleaner and more straightforward to use **GetAsyncKeyState()**. In WROXBlox! we'll use both, but for different purposes. Game play controls will all use **GetAsyncKeyState()**, but non-game-play keystrokes, like keys for pausing or toggling options, will use Windows key messages. We'll also use the Windows messages when we're waiting for the user to press any key to continue.

DirectPlay

Increasingly, computer games are allowing multiple players to play against each other in real-time. Sophisticated games allow users to hook up their systems for multiplayer games over modems and over various network protocols, like IPX, NETBEUI and TCP/IP. This makes for a more challenging and exciting game, but can also make it much more complicated to program.

The DirectPlay API simplifies this by providing a uniform communication model that is independent of the communication medium. We're going to breeze through this here, but we'll give a full code example at the end of the chapter.

Determining the Available Service Providers

Each communication medium supported by DirectPlay is represented by **a service provider**. The first step in establishing a DirectPlay session is to determine the service provider on which to create the session. We can use the **`DirectPlayEnumerate()`** function to build a list of the available service providers on the target platform:

```
HRESULT DirectPlayEnumerate( LPDPENUMCALLBACK lpCallback,
   LPVOID lpContext )
```

Parameter	Meaning
`lpCallback`	The address of a callback function to call when each service provider is identified.
`lpContext`	The user-defined data to pass into the callback function.

The function **`DP_OK`** if successful, or **`DPERR_GENERIC`** or **`DPERR_EXCEPTION`** if an error occurs.

This function uses a callback function, which must be written by you, to handle each service provider it identifies. Your callback function could be used to add items to a list box or something similar. The callback function prototype looks like this:

```
BOOL FAR PASCAL EnumCallback( LPGUID lpGUID,
      LPSTR lpDriverDescription,
      DWORD dwMajorVersion,
      DWORD dwMinorVersion,
      LPVOID lpContext)
```

Parameter	Meaning
`lpGUID`	The pointer to a GUID identifying the service provider driver.
`lpDriverDescription`	The string containing the text name of the service provider.
`dwMajorVersion`	The driver major revision number.
`dwMinorVersion`	The driver minor revision number
`lpContext`	The user-defined data passed into the **`DirectPlayEnumerate()`** function.

You should normally return **`TRUE`** from your callback function. If you return **`FALSE`**, enumeration will stop.

Constructing an IDirectPlay Object

Once you've selected a service provider to use for DirectPlay communication, you can build an **`IDirectPlay`** object to manage the connection, using **`DirectPlayCreate()`**:

```
HRESULT DirectPlayCreate(LPGUID lpGUID, LPDIRECTPLAY FAR *lplpDP,
   IUnknown FAR *pUnkOuter )
```

Parameter	Meaning
lpGUID	The pointer to the GUID of the service provider to use.
lplpDP	The pointer to a pointer to fill with the address of the new **IDirectPlay** object.
pUnkOuter	Not used. Must be **NULL**.

The function returns one of the following:

Error Result Code	Meaning
DPERR_GENERIC	Unspecified error. Call Bill Gates.
DPERR_EXCEPTION	The action tripped an exception.
DPERR_UNAVAILABLE	The service or session isn't available.
DP_OK	No error.

Establishing a Session

The first player into the game needs to establish a session to which other players can then connect. We can do this with the following function:

```
HRESULT IDirectPlay::Open(LPDPSESSIONDESC lpDesc)
```

Parameter	Meaning
lpDesc	The pointer to a **DPSESSIONDESC struct** describing the session to open.

The **DPSESSIONDESC struct** looks like this:

```
typedef struct {
    DWORD  dwSize;
    GUID   guidSession;
    DWORD  dwSession;
    DWORD  dwMaxPlayers;
    DWORD  dwCurrentPlayers;
    DWORD  dwFlags;
    char   szSessionName[DPSESSIONNAMELEN];
    char   szUserField[DPUSERRESERVED];
    DWORD  dwReserved1;
    char   szPassword[DPPASSWORDLEN];
    DWORD  dwReserved2;
    DWORD  dwUser1;
    DWORD  dwUser2;
    DWORD  dwUser3;
    DWORD  dwUser4;
} DPSESSIONDESC;
```

Field	Meaning
dwSize	The size of the **DPSESSIONDESC struct**.
guidSession	The game's **GUID**.
dwSession	The session identifier.
dwMaxPlayers	The maximum number of players allowed in the session.
dwCurrentPlayers	The current player count in the session.
dwFlags	**DPOPEN_OPENSESSION**—open an existing session. **DPOPEN_CREATESESSION**—create a new session

szSessionName	The text name of the session.
szUserField	Optional user-defined data.
dwReserved1, dwReserved2	Reserved. Do not use.
szPassword	Optional session password.
dwUser1-4	Optional user-defined data.

When you're creating a new session, set the **dwFlags** field to **DPOPEN_CREATESESSION**.

Opening an Existing Session

If a session is already started and you want to join in, you must first find the session to join using:

```
HRESULT IDirectPlay :: EnumSessions( LPDPSESSIONDESC lpSDesc,
   DWORD dwTimeout,
   LPDPENUMSESSIONSCALLBACK lpEnumCallback,
   LPVOID lpContext,
   DWORD dwFlags)
```

Parameter	Meaning
lpSDesc	The pointer to a **DPSESSIONDESC struct** containing the **GUID** of the game.
dwTimeout	The maximum time to wait for the service provider to respond with session information.
lpEnumCallback	The address of the callback function to call when each session is found.
lpContext	The pointer to user-defined data to pass into the callback.
dwFlags	**DPENUMSESSIONS_AVAILABLE** - list sessions currently accepting players. **DPENUMSESSIONS_ALL** - list all sessions for this game type

The function returns one of the following:

Error Result Code	Meaning
DPERR_INVALIDOBJECT	The DirectPlay object is invalid.
DPERR_INVALIDPARAMS	A bad parameter was passed to the function.
DP_OK	No error.

When the appropriate session to join has been selected, it is joined by calling **IDirectPlay::Open()** with the **DPOPEN_OPENSESSION** flag.

Creating Players

Once you've created or joined the session, you'll want to establish a player identity in the game for each instance:

```
HRESULT IDirectPlay :: CreatePlayer( LPDPID lppidID,
   LPSTR lpPlayerFriendlyName, LPSTR lpPlayerFormalName,
   LPHANDLE lpEvent)
```

Parameter	Meaning
lppidID	Pointer to a variable to hold the unique identifier for the player in the session
lpPlayerFriendlyName	Nickname for the player
lpPlayerFormalName	Player's full name

lpEvent	Pointer to an event to trigger for messages addressed to this player - can be **NULL**.

This returns one of the following:

Error Result Code	Meaning
DPERR_INVALIDOBJECT	The DirectPlay object is invalid.
DPERR_INVALIDPARAMS	A bad parameter was passed to the function.
DPERR_GENERIC	Unspecified error. Call Bill Gates.
DPERR_NOCONNECTION	Unable to establish service communications.
DPERR_CANTCREATEPLAYER	Couldn't create new player.
DPERR_CANTADDPLAYER	Couldn't insert player into session.
DP_OK	No error.

Receiving Messages

A DirectPlay message is a block of data that can contain anything you want it to. There are some system-defined messages with predetermined content, but, aside from those, anything is fair game. There are two ways to receive DirectPlay messages. One is to spawn an event thread and use **WaitForSingleObject()** to process incoming messages. The other (simpler, but less efficient) way is to poll for waiting messages during the app's idle time. We'll use that method in the example later on in the chapter.

To find out how many messages are waiting for a player, call:

```
HRESULT IDirectPlay :: GetMessageCount( DPID pidID, LPDWORD lpdwCount)
```

Parameter	Meaning
pidID	The ID of the player for whom to check messages.
lpdwCount	The pointer to a variable to receive the message count.

This returns one of the following:

Error Result Code	Meaning
DPERR_INVALIDOBJECT	The DirectPlay object is invalid.
DPERR_INVALIDPARAMS	A bad parameter was passed to the function.
DPERR_INVALIDPLAYER	Player ID specified isn't valid.
DP_OK	No error.

To receive message data, call:

```
HRESULT IDirectPlay :: Receive( LPDPID lppidFrom, LPDPID lppidTo,
   DWORD dwFlags, LPVOID lpvBuffer, LPDWORD lpdwSize)
```

Parameter	Meaning
lppidFrom	The pointer to variable to receive ID of player from whom message was sent.
lppidTo	The pointer to variable to receive the ID of player who is intended recipient.

dwFlags	**DPRECEIVE_ALL** gets the first message in the queue.
	DPRECEIVE_TOPLAYER looks for messages to **lppidTo**
	DPRECEIVE_FROMPLAYER looks for messages from **lppidFrom**
	DPRECEIVE_PEEK gets the message (as constrained by other flags) but leaves it on the queue
lpvBuffer	The address of buffer where message is to be copied.
lpdwSize	The size of message buffer.

Returns one of the following:

Error Result Code	Meaning
DPERR_INVALIDOBJECT	The DirectPlay object is invalid.
DPERR_INVALIDPARAMS	A bad parameter was passed to the function.
DPERR_GENERIC	Unspecified error. Call Bill Gates.
DPERR_NOMESSAGES	There are messages available to be received.
DPERR_BUFFERTOOSMALL	The buffer provided was too small to receive the message data.
DP_OK	No error.

System Messages

Any messages received from the player whose PID is 0 are system messages. DirectPlay defines **struct**s for these messages, and the buffer pointer you receive can be cast to a **DPMSG_GENERIC struct** pointer to determine which system message you've got. The **dwType** member of **DPMSG_GENERIC** might be one of the following:

dwType	Meaning
DPSYS_ADDPLAYER	A player has joined the game.
DPSYS_DELETEPLAYER	A player has left the game.
DPSYS_SESSIONLOST	The game session was lost.

Each of these types has an appropriate **struct** containing more information. We'll see this in the example later in the chapter.

Player Messages

The format of the message data you send from player-to-player is totally up to you. You might only need one kind of message, or you might want to set up a hierarchy of **struct**s or even C++ objects to pass around.

Sending Messages

To send a message to another player, use the **IDirectPlay::Send()** function:

```
HRESULT Send( DPID pidFrom, DPID pidTo, DWORD dwFlags, LPVOID lpvBuffer,
   DWORD dwBuffSize)
```

Parameter	Meaning
pidFrom	The ID of the recipient player (0 to broadcast to all players in session).
pidTo	The ID of sender player.

dwFlags	**DPSEND_GUARANTEE**—use the most reliable means to send message. **DPSEND_HIGHPRIORITY**—send with maximum priority. **DPSEND_TRYONCE**—send like a datagram; one-shot, no error detection
lpvBuffer	The pointer to the buffer containing data to send.
dwBuffSize	The size of the message buffer

This returns one of the following:

Error Result Code	Meaning
DPERR_INVALIDOBJECT	The DirectPlay object is invalid.
DPERR_INVALIDPARAMS	A bad parameter was passed to the function.
DPERR_INVALIDPLAYER	The player ID specified isn't valid.
DPERR_BUSY	The message queue is currently full.
DP_OK	No error.

Shutting Down DirectPlay

To gracefully exit a session, destroy your player object:

```
HRESULT IDirectPlay::DestroyPlayer(DPID pidID)
```

pidID is the ID of the player to destroy.

This returns one of the following:

Error Result Code	Meaning
DPERR_INVALIDOBJECT	The DirectPlay object is invalid.
DPERR_INVALIDPLAYER	The player ID specified isn't valid.
DP_OK	No error.

To close down DirectPlay altogether, **Release()** the main **IDirectPlay** object.

A DirectPlay Example: DXChat

In this section, we'll build a simple chat program to demonstrate the basics of DirectPlay. One instance of this app can initiate a chat session ('gather'), and other instances can connect to it. To chat, type text into the control next to the Send button and press Send. The message will show up in the control below (the **log window**), along with any responses from other chatters. The nickname of the sending chatter appears with each message.

Step 1: AppWizard

Use AppWizard to create a new app, called dxchat. Specify the following:

This will be an SDI app.
Turn off the printing / print preview support; we won't need it.
Set the Recent Files to 0.
Make the **CDxchatView** class descend from **CFormView**.

Step 2: The Document Class

Add members to the document class to keep track of the main **IDirectPlay** object and the ID of the local 'player', and a member to keep track of the app's main view for later reference:

```
LPDIRECTPLAY   m_pTheDPObject;
DPID           m_localChatter;
CDxchatView*   m_pMainView;
```

Set all of these to **NULL** in the constructor. Add a function to shut down DirectPlay:

```
void CDxchatDoc :: stopSession ()
{
   if (m_pTheDPObject != NULL)
   {
      // Destroy local player
      if (m_localChatter != 0)
         m_pTheDPObject->DestroyPlayer(m_localChatter);

      // Shut down DirectPlay
      m_pTheDPObject->Release();
      m_pTheDPObject = NULL;

      theApp.m_pDoc = NULL;

      SetTitle("(Not connected)");
   }
}
```

Call this function from the destructor to make sure DirectPlay is cleaned up on exit.

Make the following modifications to the application menu bar:

Remove all items except Exit from the File menu.
Add Gather Session…, Join Session… and Send Message items to File menu.
Remove all items from the Edit menu, then add in a Clear Log Window item.

Make the following modifications to the application toolbar:

Remove all tool buttons except the Help button.
Add an eraser button, hooked up to the Edit/Clear Log Window command.

In the Project Settings dialog for the project, go to the Link tab and add dplay.lib.

Add command handlers to **CDxchatDoc** for the gather, join, send and clear menu items that you've just added:

```
void CDxchatDoc::OnGather()
{
   CSessionDlg dlg;

   // If there is an existing session, stop it
   stopSession ();


   // Post the Gather Session Dialog to get info about the new session
   if (dlg.DoModal() == IDOK)
   {
      // Create a DirectPlay object around the service provider
      if (DirectPlayCreate(dlg.m_lpguid, &m_pTheDPObject, NULL) == DP_OK)
      {
         // Create a session
         DPSESSIONDESC   sdesc;

          memset(&sdesc, 0x00, sizeof(DPSESSIONDESC));
```

```
         sdesc.dwSize = sizeof(DPSESSIONDESC);
         sdesc.dwFlags = DPOPEN_CREATESESSION;
         sdesc.dwMaxPlayers = 10;
         sdesc.guidSession = CHAT_GUID;
         strcpy(sdesc.szSessionName,dlg.m_session);

         if (m_pTheDPObject->Open(&sdesc)  == DP_OK)
         {
            // Allow joining
            m_pTheDPObject->EnableNewPlayers(TRUE);

            // Join the session locally
            if (m_pTheDPObject->CreatePlayer(&m_localChatter,
               (char*)(const char*)dlg.m_nickname,
               (char*)(const char*)dlg.m_realname, NULL) == DP_OK)
            {
               m_pMainView->MessageBox("Session started."
                  "  Other chatters may now connect.");
               theApp.m_pDoc = this;
               SetTitle(dlg.m_session);
            }
            else
               m_pMainView->MessageBox("Unable to create local player.");

         }
         else
            m_pMainView->MessageBox("Unable to create session.");
      }
      else
         m_pMainView->MessageBox("Unable to initialize DirectPlay.");

   }

}

void CDxchatDoc::OnJoin()
{
   CJoinDlg dlg;
   CJoin2Dlg dlg2(this);

   // If there is an existing session, stop it
   stopSession ();


   // Post the Join Session Dialog to get info about the new session
   if (dlg.DoModal() == IDOK)
   {
      // Create a DirectPlay object around the service provider
      if (DirectPlayCreate(dlg.m_lpguid, &m_pTheDPObject, NULL) == DP_OK)
      {

         // Get the session ID
         if (dlg2.DoModal() == IDOK)
         {

            // Open the session
            DPSESSIONDESC   sdesc;

            memset(&sdesc, 0x00, sizeof(DPSESSIONDESC));

            sdesc.dwSize = sizeof(DPSESSIONDESC);
            sdesc.dwFlags = DPOPEN_OPENSESSION;
            sdesc.guidSession = CHAT_GUID;
            sdesc.dwSession = dlg2.m_sessionid;


            // Try to connect to session
            if (m_pTheDPObject->Open(&sdesc)  == DP_OK)
            {
               // Allow joining
               m_pTheDPObject->EnableNewPlayers(TRUE);
```

```
               // Join the session locally
               if (m_pTheDPObject->CreatePlayer(&m_localChatter,
                  (char*)(const char*)dlg.m_nickname,
                  (char*)(const char*)dlg.m_realname, NULL) == DP_OK)
               {
                  m_pMainView->MessageBox("Chat session joined.");
                  SetTitle(dlg2.m_session);
                  theApp.m_pDoc = this;
               }
               else
                  m_pMainView->MessageBox(
                     "Unable to create local player.");

            }
            else
               m_pMainView->MessageBox("Unable to connect to session.");

         }
      }
      else
         m_pMainView->MessageBox("Unable to initialize DirectPlay.");
   }
}

void CDxchatDoc::OnSend()
{
   // Send text in message edit to the current session

   CString sendtxt = m_pMainView->getSendText();
   CString localtxt = "<me>: ";

   const char* buf = sendtxt.GetBuffer(257);


   m_pTheDPObject->Send(m_localChatter, 0, DPSEND_GUARANTEE, (LPVOID)buf,
      sendtxt.GetLength() + 1);

   localtxt += sendtxt;
   m_pMainView->addText(localtxt);

   sendtxt.ReleaseBuffer();
}



void CDxchatDoc::OnEditClear()
{
   m_pMainView->clearText();
}
```

Add a function to handle the DirectPlay message traffic:

```
void CDxchatDoc::processDPMessages()
{
   DWORD num_msgs;
   long i;

   if (m_pTheDPObject->GetMessageCount(m_localChatter, &num_msgs)!= DP_OK)
      return;

   for (i=0;i<(long)num_msgs;i++)
   {
      DPID   idFrom, idTo;
      char   msg[256];
      DWORD   msglen = 256;

      if (m_pTheDPObject->Receive(&idFrom, &idTo, DPRECEIVE_ALL, msg,
         &msglen) == DP_OK)
      {
```

```
            if (idFrom == 0)
            {
               DPMSG_GENERIC*   pDPGen = (DPMSG_GENERIC*)msg;

               // Message from the name server
               switch (pDPGen->dwType)
               {
                  case DPSYS_ADDPLAYER:
                     {
                        DPMSG_ADDPLAYER* pDPAddP = (DPMSG_ADDPLAYER*)msg;

                        // No need to do this if it's us
                        if (pDPAddP->dpId != m_localChatter)
                        {
                           CString msgstr = pDPAddP->szShortName;
                           msgstr += " has joined in the chat.";

                           m_pMainView->MessageBox(msgstr);
                        }
                     }
                     break;

                  case DPSYS_DELETEPLAYER:
                     m_pMainView->MessageBox(
                        "A chatter has left the session.");
                     break;

                  case DPSYS_CONNECT:
                     m_pMainView->MessageBox("Connected to chat server.");
                     break;

                  case DPSYS_SESSIONLOST:
                     {
                        m_pMainView->MessageBox(
                           "Connection to chat server was lost.");
                        stopSession();
                     }
                     break;
               };
            }
            else if (idTo == m_localChatter)
            {
               CString txt;

               // A text message

               // Get the sender's nickname
               char* buf = txt.GetBuffer(256);
               DWORD nicklen = 255;
               m_pTheDPObject->GetPlayerName(idFrom, buf, &nicklen,
                  NULL, NULL);
               txt.ReleaseBuffer();

               txt += ": ";
               txt += msg;

               // Show the text
               m_pMainView->addText(txt);
            }
         }
      }
   }
```

Define the GUID for the game:

```
   GUID CHAT_GUID = {
      0x01483ce0,
      0x6f72,
      0x11cf,
      { 0x81,0xf0,0x44,0x45,0x53,0x54,0x00,0x00 }
```

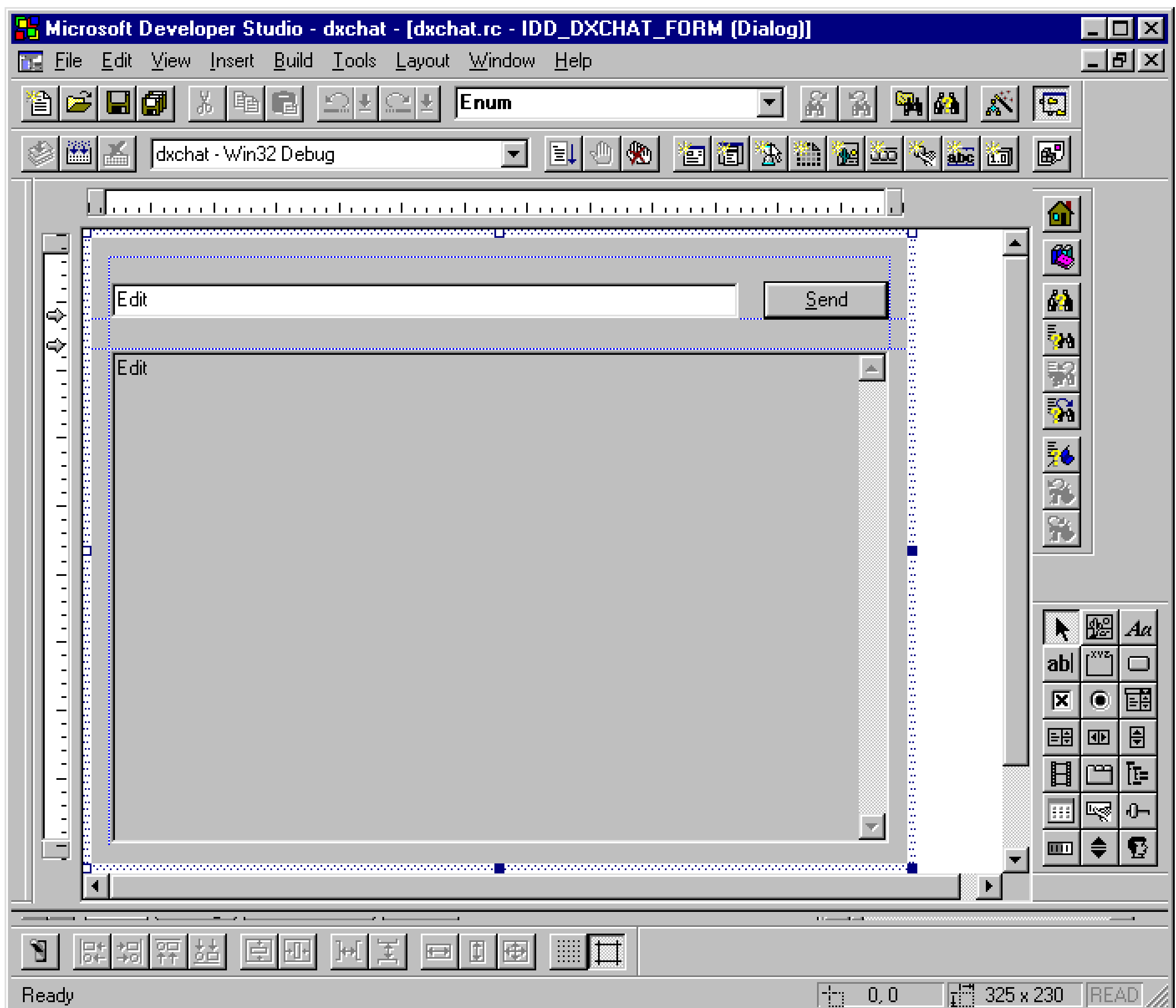
```
};
```

Add a member to **CDxchatApp** to keep track of the document and add an **OnIdle()** handler to deal with DirectPlay message dispatching:

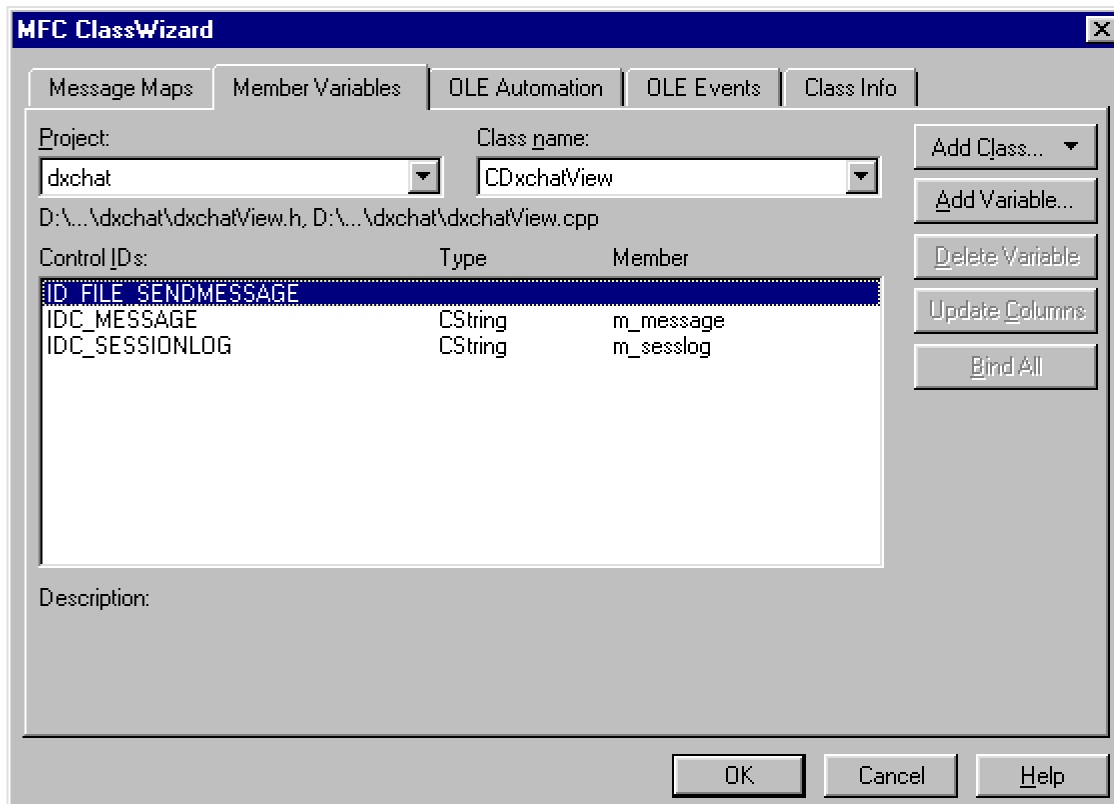
```
BOOL CDxchatApp::OnIdle(LONG lCount)
{
   // If the doc is up and running, call its DirectPlay message processor
   if (m_pDoc)
      m_pDoc->processDPMessages();

   return CWinApp::OnIdle(lCount);
}
```

Step 3: The Main View

Lay out the form to look like this:



Use ClassWizard to add member variables for the controls:



Add an **OnInitialUpdate()** function:

```
void CDxchatView::OnInitialUpdate()
{
   CFormView::OnInitialUpdate();

   GetDocument()->m_pMainView = this;
   GetDocument()->SetTitle("(Not connected)");
}
```

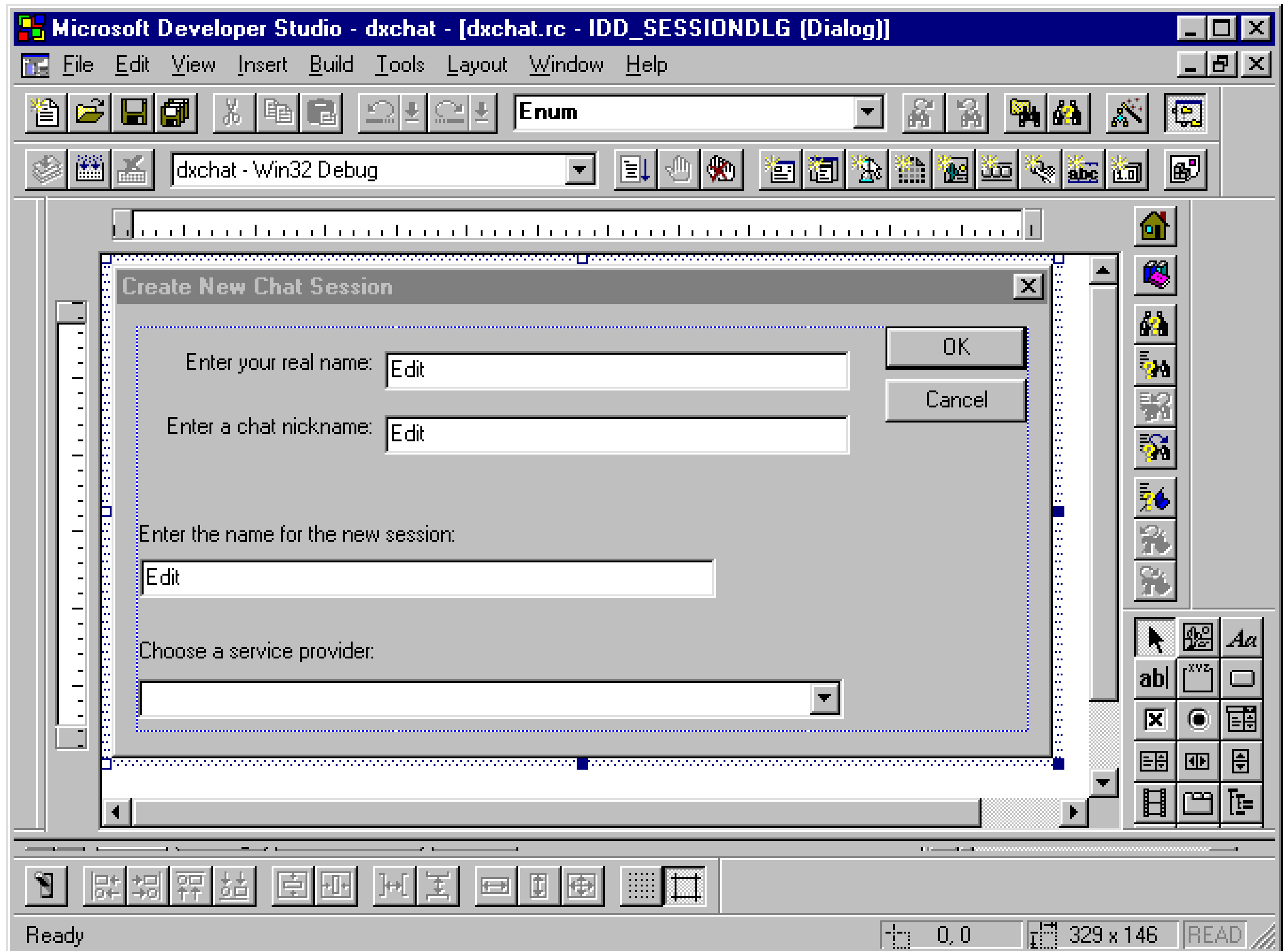
Add some helper functions to simplify working with the controls:

```
CString   CDxchatView :: getSendText()
{
   UpdateData();
   return m_message;
}

void CDxchatView :: addText (const char* new_txt)
{
   UpdateData();
   m_sesslog += new_txt;
   m_sesslog += "\r\n";
   UpdateData(FALSE);
}

void CDxchatView :: clearText ()
{
   m_sesslog = "";
```

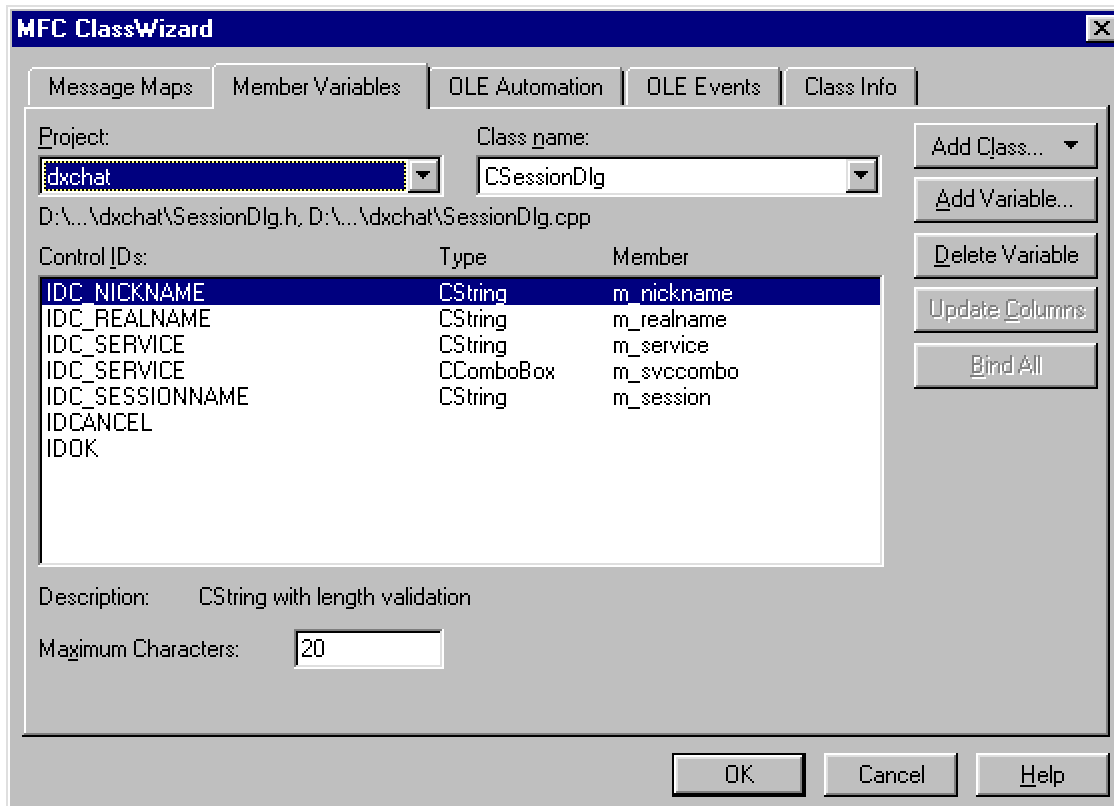
```
        UpdateData(FALSE);
    }
```

Step 4: The Create Session Dialog

Add a dialog for collecting the information necessary to establish a new session. Make it look like this:



Use ClassWizard to make a class for it, and to add member variables for its controls:



Add an **`OnInitDialog()`** handler and, in it, enumerate the available DirectPlay service providers:

```
// Enum proc used to add available service providers to combo box
BOOL FAR PASCAL ServiceEnumProc (LPGUID lpGUID, LPSTR pszName,
   DWORD majver, DWORD minver, LPVOID lpData)
{
   ((CSessionDlg*)lpData)->addToList(pszName, (DWORD)lpGUID);

   return TRUE;
}

// Add one service provider to the combo box - used by enum function
void CSessionDlg :: addToList (const char* svcname, DWORD lpGUID)
{
   int idx = m_svccombo.AddString(svcname);
   m_svccombo.SetItemData(idx,lpGUID);
}

BOOL CSessionDlg::OnInitDialog()
{
   CDialog::OnInitDialog();


   if (DirectPlayEnumerate(ServiceEnumProc, this) == DP_OK)
   {
      m_svccombo.SetCurSel(0);
   }
   else
   {
      MessageBox (
         "DirectPlay Error: Unable to enumerate service providers.");
      EndDialog(FALSE);
   }
```

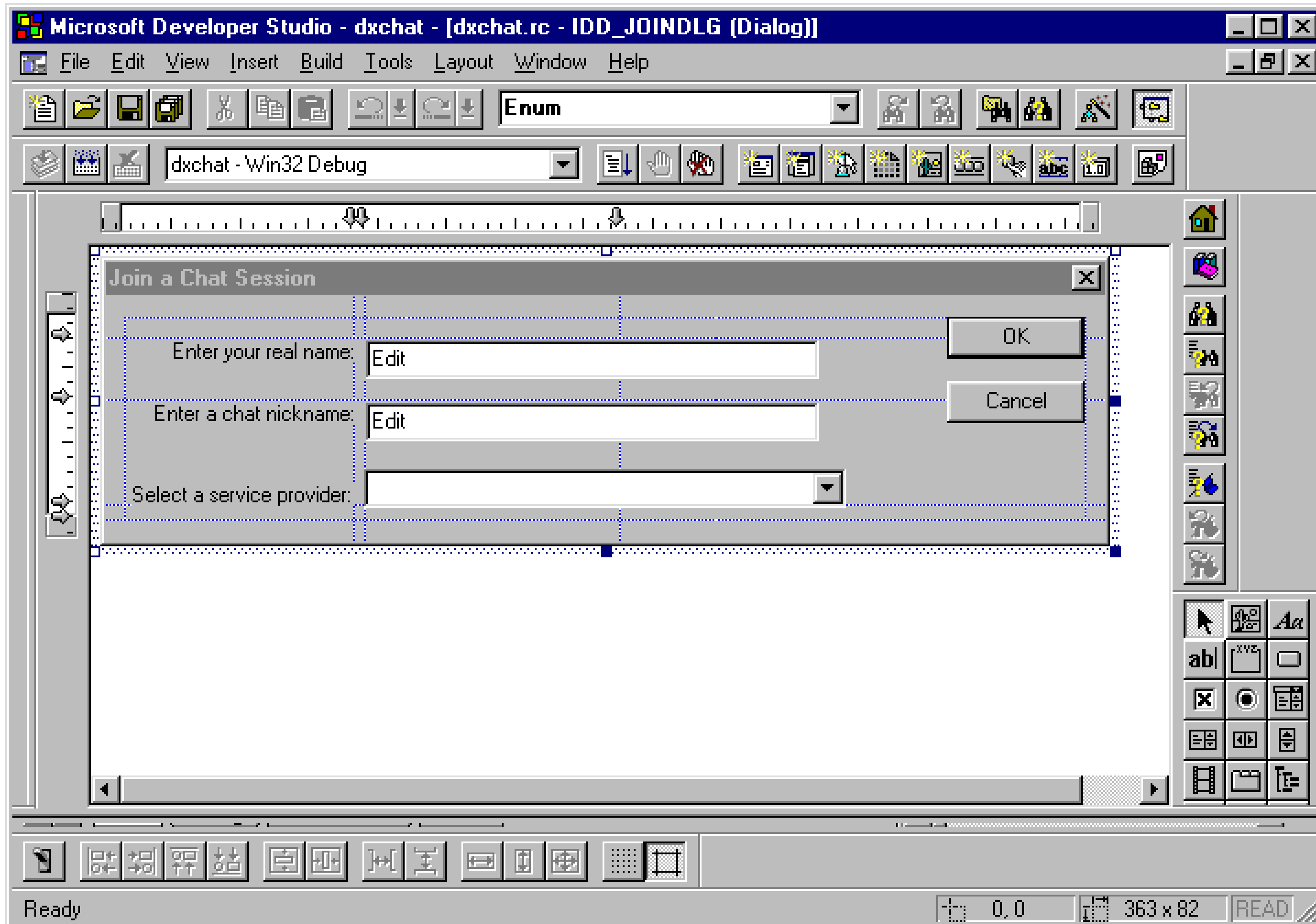
```
        return TRUE;  // return TRUE unless you set the focus to a control
    }
```

Add an **OnOK()** handler to get the GUID out of the combo box and store it in a member variable:

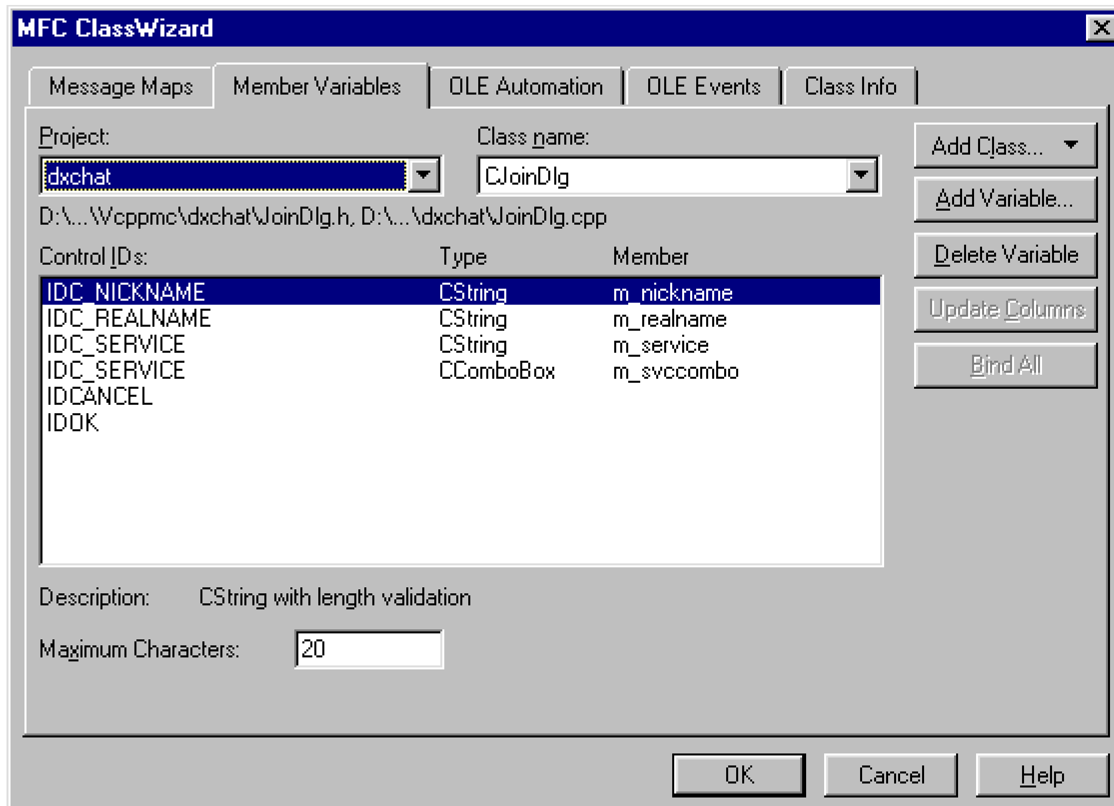
```
    void CSessionDlg::OnOK()
    {
        int idx = m_svccombo.GetCurSel();
        m_lpguid  = (LPGUID)m_svccombo.GetItemData(idx);

        CDialog::OnOK();
    }
```

Step 5: The Join Session 1 Dialog

Add a dialog to start the process of connecting to a session by getting the service provider and user identification information. Make it look like this:



Use ClassWizard to make a class for it, and to add member variables for its controls:



Add an **`OnInitDialog()`** handle and, in it, enumerate the available DirectPlay service providers:

```
// Enum proc used to add available service providers to combo box
BOOL FAR PASCAL JoinServiceEnumProc (LPGUID lpGUID, LPSTR pszName, DWORD majver, DWORD
minver, LPVOID lpData)
{
   ((CJoinDlg*)lpData)->addToSvcList(pszName, (DWORD)lpGUID);

   return TRUE;
}

// Add one service provider to the combo box - used by enum function
void CJoinDlg :: addToSvcList (const char* svcname, DWORD lpGUID)
{
   int idx = m_svccombo.AddString(svcname);
   m_svccombo.SetItemData(idx,lpGUID);
}

BOOL CJoinDlg::OnInitDialog()
{
   CDialog::OnInitDialog();

   if (DirectPlayEnumerate(JoinServiceEnumProc, this) == DP_OK)
   {
      m_svccombo.SetCurSel(0);
   }
   else
   {
      MessageBox (
         "DirectPlay Error: Unable to enumerate service providers.");
      EndDialog(FALSE);
   }

   return TRUE;  // return TRUE unless you set the focus to a control
```

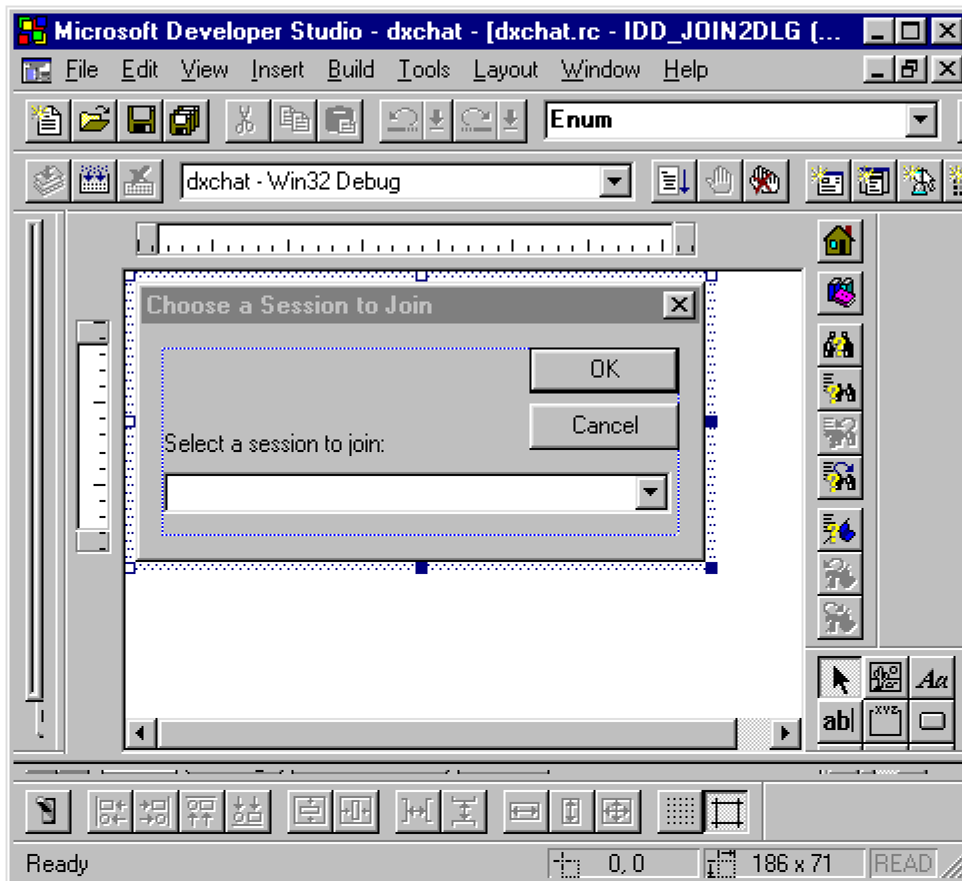
```
}
```

Add an **OnOK()** handler to get the GUID out of the combo box and store it in a member variable:

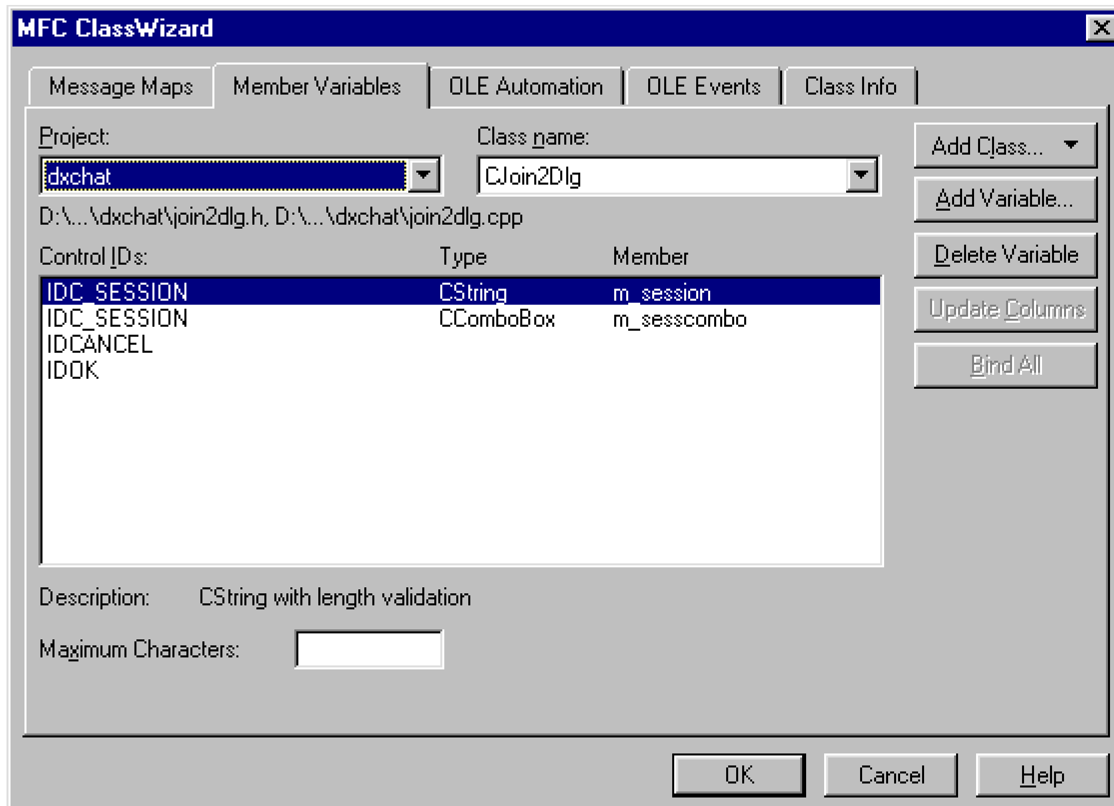
```
void CJoinDlg::OnOK()
{
   int idx = m_svccombo.GetCurSel();
   m_lpguid  = (LPGUID)m_svccombo.GetItemData(idx);

   CDialog::OnOK();
}
```

Step 6: The Join Session 2 Dialog

Add a dialog to finish the process of connecting to a session by enumerating the available sessions for the selected service provider and allowing the user to choose one. Make the dialog look like this:



Use ClassWizard to make a class for it, and to add member variables for its controls:



Pass a pointer to the document into the constructor. We'll need access to the **IDirectPlay** object in this dialog. Store it in a member in the dialog class.

Add an **OnInitDialog()** handler and, in it, enumerate the available sessions:

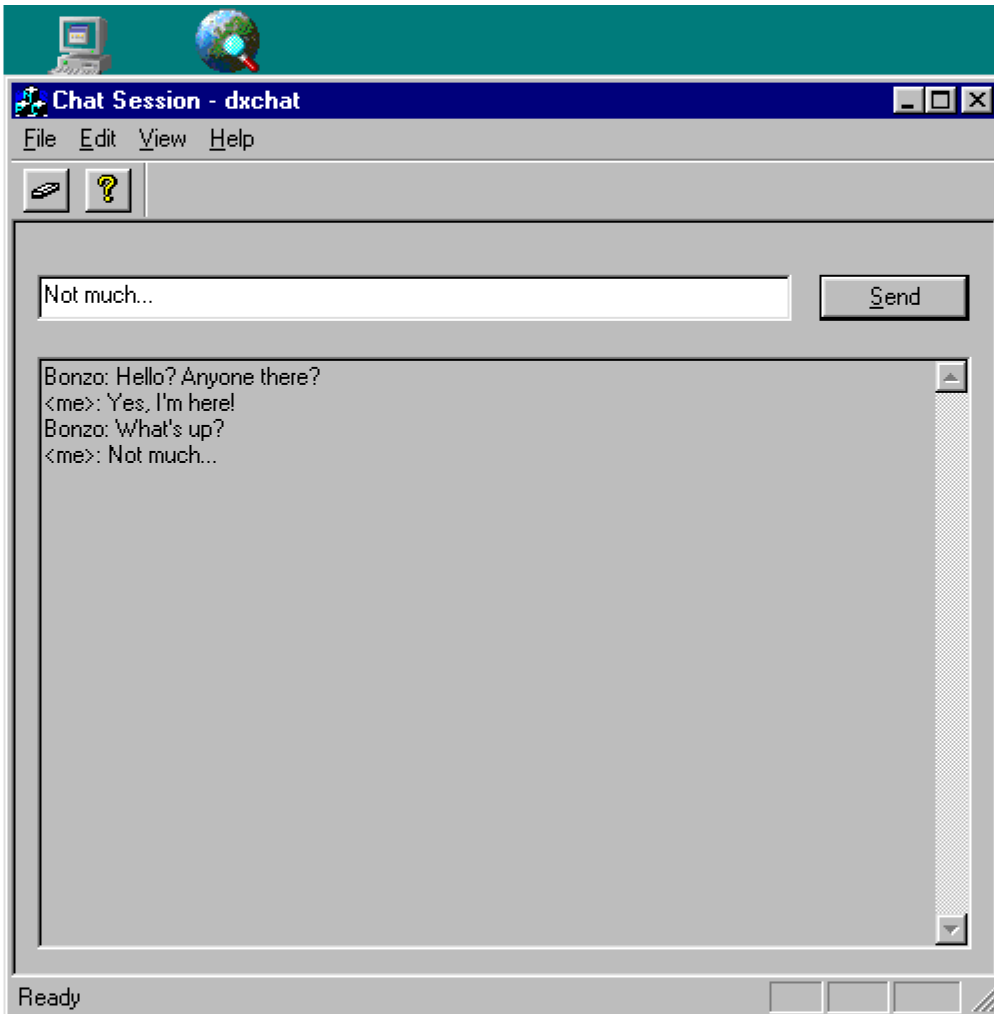
```
BOOL FAR PASCAL SessionEnumProc (LPDPSESSIONDESC lpDPDesc, LPVOID lpContext, LPDWORD
timeout, DWORD flags)
{
   ((CJoin2Dlg*)lpContext)->addToSessList(lpDPDesc->szSessionName,
      lpDPDesc->dwSession);

   return TRUE;
}

void CJoin2Dlg :: addToSessList (const char* sessname, DWORD sessid)
{
   int idx = m_sesscombo.AddString(sessname);
   m_sesscombo.SetItemData(idx,sessid);
}

BOOL CJoin2Dlg::OnInitDialog()
{
   CDialog::OnInitDialog();

   DPSESSIONDESC sdesc;

   memset(&sdesc, 0x00, sizeof(DPSESSIONDESC));

   sdesc.dwSize = sizeof(DPSESSIONDESC);
   sdesc.guidSession = CHAT_GUID;

   BeginWaitCursor();
```

```
   if (m_pDoc->m_pTheDPObject->
      EnumSessions(&sdesc, 5000, SessionEnumProc, this, NULL) == DP_OK)
   {
      m_sesscombo.SetCurSel(0);
      EndWaitCursor();
   }
   else
   {
      MessageBox (
         "DirectPlay Error: Unable to enumerate available sessions.");
      EndWaitCursor();
      EndDialog(FALSE);
   }

   return TRUE;  // return TRUE unless you set the focus to a control
}
```

Add an **`OnOK()`** handler to get the session ID out of the combo box and store it in a member variable:

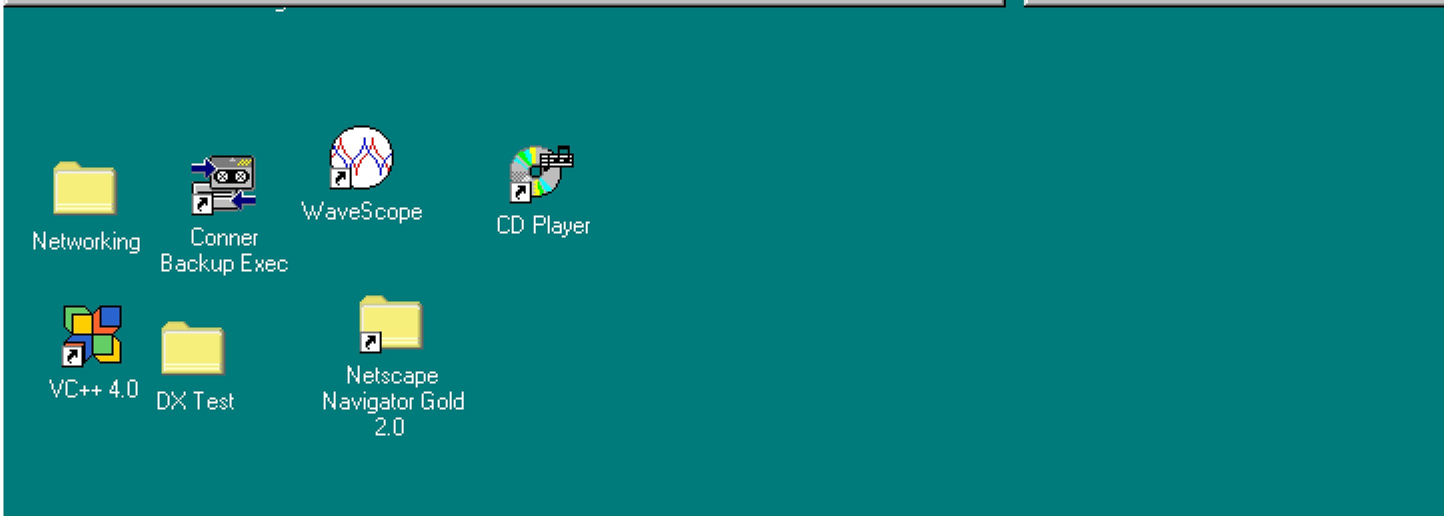
```
void CJoin2Dlg::OnOK()
{
   int idx = m_sesscombo.GetCurSel();
   m_sessionid  = m_sesscombo.GetItemData(idx);

   CDialog::OnOK();
}
```

That's it. The finished app looks like this:



Chat Session - dxchat
File Edit View Help
Not much...
Send
Bonzo: Hello? Anyone there?
<me>: Yes, I'm here!
Bonzo: What's up?
<me>: Not much...
Ready

Chat Session - dxchat
File Edit View Help
What's up?
<me>: Hello? Anyone there?
Andy: Yes, I'm here!
<me>: What's up?
Andy: Not much...
Ready



Networking
Conner Backup Exec
WaveScope
CD Player
VC++ 4.0
DX Test
Netscape Navigator Gold 2.0

Wrapping DirectX in C++

Clearly, from what we've just seen, DirectX can do everything we need to make WROXBlox! go, and a lot more. It's going to be a bit cumbersome to use, though, so there's a lot for us to do. It sure would be nice to have this stuff organized into objects, with much of the initialization and such hidden from us, wouldn't it? Well, since Microsoft didn't do it for us, let's get moving and do it ourselves. We'll make a class to handle all of the top-level game stuff (like the **`IDirectDraw`** and **`IDirectSound`** master objects), another class for sprites, and finally, one to handle individual wave audio sounds.

First, Some C++ Performance Issues

Compared to raw C, C++ has a bad rap as a performance killer. If you take a few precautions, however, C++ code can run pretty much as fast as C code. Here are a few pointers:

- Inline everything possible.
- Don't use exceptions.
- Avoid function call indirections:
- Don't use multiple inheritance.
- Avoid using virtual functions. If you must use them, inline them if you can. The compiler can sometimes make more efficient use of them.
- Avoid referencing objects through pointers. Whenever possible, make the actual object visible if you need to use it.

Of course, this is for speed. If speed isn't an issue, or more robust code is, design your classes more in line with normal OOP practices.

You'll notice that we'll follow these rules in the definition of the classes to follow. One glaring violation we'll perpetrate is to make the 'game slice' function in our app class a virtual function. When we get to it, we'll look at some alternatives. Making it a virtual function for our purposes here makes the code more readable and comprehensible, and it doesn't seem to greatly affect game performance for such a simple game.

CDirectXApp: A Game App Object

Let's get down to the business of building some C++ classes. We'll start with the top-level game object, **`CDirectXApp`**. We want this class to encapsulate all the nitty-gritty details of bringing up and shutting down DirectX, including creating the flipping surface and loading sprite bitmaps for use. It will hold all of our primary DirectX objects, like the one-and-only **`IDirectDraw`** and **`IDirectSound`** objects for an app, an **`IDirectDrawSurface`** for the primary flipping surface (with one back buffer), plus two off-screen **`IDirectDrawSurfaces`**: one for sprite data and another for the splash screen bitmap. We'll also need **`IDirectDrawPalettes`** corresponding to those surfaces. The **`CDirectXApp`** class should also hang onto pointers to the front and back buffers of the primary flipping surface for easy access. Also, we'll have this object create and hang onto the main application window. **`CDirectXApp`** should run the app's message loop and provide functionality for rendering sprites to the back buffer and for flipping the primary surface. We'll make the game slice a pure virtual function of this class. To use **`CDirectXApp`**, you'll create a descendant class that overrides the game slice function.

One sticky point here is that we'll need a window procedure for the main window, but we want to automate the management of that window as much as possible. A quick solution to this is to use extern linkage. You must supply the window proc function with the same name when you use **`CDirectXApp`**. There are other ways around this, but this is pretty straightforward, and introduces no additional indirection overhead to the app. We'll prototype the extern function like this:

```
extern long FAR PASCAL DXAppWndProc (HWND hWnd, UINT msg, WPARAM wParam,
   LPARAM lParam)
```

That's about all we'll need in **`CDirectXApp`**. While this object is rather rigid in its use of DirectX, it will nonetheless serve as a quite adequate foundation for games as simple as WROXBlox!, which don't need multiple back buffers or

other fancy features like that. Here's the class declaration for **CDirectXApp**:

```
class CDirectXApp
{

protected:

   // DirectX COM Objects:
   LPDIRECTSOUND         m_lpDSound;        // The IDirectSound object
   LPDIRECTDRAW          m_lpTheDDObject;   // The IDirectDraw object
   LPDIRECTDRAWSURFACE   m_lpDDSPrimary;    // DirectDraw primary surface
   LPDIRECTDRAWSURFACE   m_lpDDSBack;       // DirectDraw back surface
   LPDIRECTDRAWPALETTE   m_lpDDPal;         // Main DirectDraw palette
   LPDIRECTDRAWPALETTE   m_lpDDSplashPal;   // Splash panel DirectDraw
                                            // palette
   LPDIRECTDRAWSURFACE   m_lpDDSSprite;     // Offscreen sprite surface
   LPDIRECTDRAWSURFACE   m_lpDDSSplash;     // Offscreen splash panel
                                            // surface

   BOOL             m_bActive;      // is application active?

   HWND             m_hMainWnd;     // The app's main window

   CDirectXSprite   splash_sprite;  // A sprite for the splash bitmap

   HRESULT          m_hLastError;   // Last DirectDraw error that occurred

public:

   // DirectX COM Object Accessors

   LPDIRECTSOUND theDSoundObj ()
      {return m_pDSound;}

   void setTheDSoundObj (LPDIRECTSOUND pNewDS)
      {m_pDSound = pNewDS;}

   LPDIRECTDRAW theDDrawObj ()
      {return m_lpTheDDObject;}

   void setTheDDrawObj (LPDIRECTDRAW pNewDD)
      {m_lpTheDDObject = pNewDD;}

   LPDIRECTDRAWSURFACE thePrimarySurf ()
      {return m_lpDDSPrimary;}

   void setThePrimarySurf (LPDIRECTDRAWSURFACE pNewSurf)
      {m_lpDDSPrimary = pNewSurf;}

   LPDIRECTDRAWSURFACE theBackSurf ()
      {return m_lpDDSBack;}

   void setTheBackSurf (LPDIRECTDRAWSURFACE pNewSurf)
      {m_lpDDSBack = pNewSurf;}

   LPDIRECTDRAWPALETTE theDDrawPal ()
      {return m_lpDDPal;}

   void setTheDDrawPal (LPDIRECTDRAWPALETTE pNewPal)
      {m_lpDDPal = pNewPal;}

   LPDIRECTDRAWSURFACE theSpriteSurf ()
      {return m_lpDDSSprite;}

   void setTheSpriteSurf (LPDIRECTDRAWSURFACE pNewSurf)
      {m_lpDDSSprite = pNewSurf;}


   HWND gameWnd () {return m_hMainWnd;}
```

```
    BOOL isActive () {return m_bActive;}
    void  setActive (BOOL bAct = TRUE) {m_bActive = bAct;}


    // Start up and shut down the DirectDraw part of the app
    BOOL init (HINSTANCE hInstance, int nCmdShow);
    void shutdown();

    // Start up and shut down the DirectSound stuff
    BOOL initSound ();
    void shutdownSound ();

    // Functions to handle sprite and splash bitmaps
    BOOL loadSpritesFromBitmapRes ();
    BOOL loadSplashScreen ();
    void drawSplashScreen ();
    void selectSplashPalette ();
    void selectSpritePalette ();

    HRESULT restore ();

    // App message loop
    int go ();

    // Flip the primary surface
    void flip () {m_lpDDSPrimary->Flip( NULL, DDFLIP_WAIT );}

    // Draw a sprite on the back buffer
    void renderSprite (CDirectXSprite& sprite)
       { sprite.drawOn(m_lpDDSBack);}

    // Override in descendant class and put in game loop functionality
    virtual void gameSlice () = 0;


    CDirectXApp ();
    ~CDirectXApp ();

};
```

We've already covered what has to happen in the guts of most of these functions, so without further ado, here are the **CDirectXApp** member functions:

```
// Includes all the necessary Windows and GDK headers
#include "dxappclass.h"
#include "ddutil.h"


// Window proc - must be defined in calling app
extern long FAR PASCAL DXAppWndProc (HWND hWnd, UINT msg, WPARAM wParam,
   LPARAM lParam);

// App must have a BMP resource with this name
#define SPRITE_RES    "DXAPP_SPRITES"

// If app uses built-in splash panel functionality,
// it must have a BMP resource with this name
#define SPLASH_RES    "DXAPP_SPLASH"

// App must have an icon with this ID
#define ICON_RES    "DXAPP_ICON"


CDirectXApp :: CDirectXApp ()
{
   m_pDSound = NULL;
   m_hLastError = NULL;
}
```

```
BOOL CDirectXApp :: initSound ()
{
   if (DirectSoundCreate(NULL, &m_pDSound, NULL)  == DS_OK)
   {
      if (m_pDSound->SetCooperativeLevel(m_hMainWnd, DSSCL_NORMAL)
         == DS_OK)
         return TRUE;
      else
      {
         m_pDSound->Release();
         m_pDSound = NULL;
      }
   }

   return FALSE;
}


void CDirectXApp :: shutdownSound ()
{
   if (m_pDSound)
   {
      m_pDSound->Release();
      m_pDSound = NULL;
   }
}


BOOL CDirectXApp :: init (HINSTANCE hInstance, int nCmdShow)
{
   WNDCLASS            wc;
   char                buf[256];

   DDSURFACEDESC       ddsd;
   DDSCAPS             ddscaps;
   HRESULT             ddrval;

   // Register a window class for the game main window
   wc.style = CS_HREDRAW | CS_VREDRAW | CS_BYTEALIGNCLIENT;
   wc.lpfnWndProc = DXAppWndProc;
   wc.cbClsExtra = 0;
   wc.cbWndExtra = 0;
   wc.hInstance = hInstance;
   wc.hIcon = LoadIcon( hInstance, ICON_RES );
   wc.hCursor = LoadCursor( NULL, IDC_ARROW );
   wc.hbrBackground = NULL;
   wc.lpszMenuName = (LPSTR)NULL;
   wc.lpszClassName = "DirectXAppWnd";
   RegisterClass( &wc );


   // Create and show the main window
   m_hMainWnd = CreateWindowEx(
      WS_EX_TOPMOST,
      "DirectXAppWnd",
      "Generic DirectX Game",
      WS_POPUP,
      0, 0, 640, 480,
      NULL,
      NULL,
      hInstance,
      NULL );

   if( !m_hMainWnd )
   {
      MessageBox (NULL,
         "Unable to create main window.",
         "DirectX App Error",
          MB_OK);
      return FALSE;
   }
```

```
   ShowWindow( m_hMainWnd, nCmdShow );
   UpdateWindow( m_hMainWnd );

   // Initialize DirectDraw
   ddrval = DirectDrawCreate( NULL, &m_lpTheDDObject, NULL );
   if( ddrval == DD_OK )
   {
      // Get exclusive mode
      ddrval = m_lpTheDDObject->SetCooperativeLevel( m_hMainWnd,
         DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN );
      if(ddrval == DD_OK )
      {
         // Set video mode to 640x480x256
         ddrval = m_lpTheDDObject->SetDisplayMode( 640, 480, 8 );
         if( ddrval == DD_OK )
         {
            // Create the primary surface with 1 back buffer
            ddsd.dwSize = sizeof( ddsd );
            ddsd.dwFlags = DDSD_DDCAPS | DDSD_BACKBUFFERCOUNT;
            ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
               DDSCAPS_FLIP |
               DDSCAPS_COMPLEX;
            ddsd.dwBackBufferCount = 1;
            ddrval = m_lpTheDDObject->CreateSurface( &ddsd,
               &m_lpDDSPrimary, NULL );
            if( ddrval == DD_OK )
            {
               // Get a pointer to the back buffer
               ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
               ddrval = m_lpDDSPrimary->
                  GetAttachedSurface(&ddscaps, &m_lpDDSBack);
               if( ddrval == DD_OK )
               {
                  if (loadSpritesFromBitmapRes())
                  {
                     if (loadSplashScreen())
                        return TRUE;
                     else
                     {
                        MessageBox( NULL, "Error - unable to load splash"
                           " screen - probably not enough video memory on"
                           " your display adapter",
                           "DirectX App Error", MB_OK );
                        return FALSE;
                     }
                  }
                  else
                  {
                     MessageBox( NULL, "Error - unable to load sprites "
                        "- probably not enough video memory on your "
                        "display adapter",
                        "DirectX App Error", MB_OK );
                     return FALSE;
                  }
               }
            }
         }
      }
   }

   m_hLastError = ddrval;

   wsprintf(buf, "Direct Draw Init Failed (%08lx)\n", ddrval );
   MessageBox( m_hMainWnd, buf, "DirectX App Error", MB_OK );
   shutdown();
   DestroyWindow( m_hMainWnd );

   return FALSE;
}
```

```
BOOL CDirectXApp :: loadSplashScreen ()
{
   // create and set the palette
   m_lpDDSplashPal = DDLoadPalette(m_lpTheDDObject, SPLASH_RES);

   if (m_lpDDSplashPal)
      m_lpDDSPrimary->SetPalette(m_lpDDSplashPal);

   // Create the offscreen surface, by loading our bitmap.
   m_lpDDSSplash = DDLoadBitmap(m_lpTheDDObject, SPLASH_RES, 0, 0);

   if( m_lpDDSSplash == NULL )
   {
      shutdown();
      DestroyWindow( m_hMainWnd );
      return FALSE;
   }

   // We're not going to use it, but we have to set a color key for
   // the splash screen since we're going to use the sprite drawing
   // logic to display it
   DDSetColorKey(m_lpDDSSplash, RGB(0,0,0));

   // Set up the special splash screen sprite
   splash_sprite.setSource (m_lpDDSSplash,0,0,640,480);

   // Set back sprite palette
   if (m_lpDDPal)
      m_lpDDSPrimary->SetPalette(m_lpDDPal);

   return TRUE;
}


void CDirectXApp :: selectSplashPalette ()
{
   // Set splash palette
   if (m_lpDDSplashPal)
      m_lpDDSPrimary->SetPalette(m_lpDDSplashPal);
}


void CDirectXApp :: selectSpritePalette ()
{
   // Set sprite palette
   if (m_lpDDPal)
      m_lpDDSPrimary->SetPalette(m_lpDDPal);
}


void CDirectXApp :: drawSplashScreen ()
{
   renderSprite (splash_sprite);
}


BOOL CDirectXApp :: loadSpritesFromBitmapRes ()
{

   // create and set the palette
   m_lpDDPal = DDLoadPalette(m_lpTheDDObject, SPRITE_RES);

   if (m_lpDDPal)
      m_lpDDSPrimary->SetPalette(m_lpDDPal);

   // Create the offscreen surface, by loading our bitmap.
   m_lpDDSSprite = DDLoadBitmap(m_lpTheDDObject, SPRITE_RES, 0, 0);

   if( m_lpDDSSprite == NULL )
   {
```

```
      shutdown();
      DestroyWindow( m_hMainWnd );
      return FALSE;
   }

   // if we did not want to hard code the palette index (0xff)
   // we can also set the color key like so...
   DDSetColorKey(m_lpDDSSprite, RGB(0,0,0));

   return TRUE;
}


void CDirectXApp :: shutdown ()
{
   m_bActive = FALSE;

   if( m_lpTheDDObject != NULL )
   {
      if( m_lpDDSPrimary != NULL )
      {
         m_lpDDSPrimary->Release();
         m_lpDDSPrimary = NULL;
      }
      m_lpTheDDObject->Release();
      m_lpTheDDObject = NULL;
   }

   shutdownSound();
}


int CDirectXApp :: go()
{
   MSG      msg;

   // Run the message loop, calling gameSlice on idle
   while (1)
   {
      if (PeekMessage (&msg, NULL, 0, 0, PM_NOREMOVE))
      {
         if (!GetMessage(&msg, NULL, 0, 0))
            return msg.wParam;

         TranslateMessage(&msg);
         DispatchMessage(&msg);
      }
      else if (m_bActive)
         gameSlice();
   }

   return msg.wParam;
}
```

That's the **CDirectXApp** object. We've set a few rules along the way that we should recap. All **CDirectXApp**-based applications must:

- Subclass **CDirectXApp** and override the **gameSlice()** function.
- Define a window procedure for the main window and name it **DXAppWndProc**.
- Define a bitmap resource for the game sprites and call it **DXAPP_SPRITES**.
- Define a bitmap resource for the game splash screen and call it **DXAPP_SPLASH**.
- Define an icon resource for the app and call it **DXAPP_ICON**.

So, the resource file for the WROXBlox! app will have the following things in it:

```
// wbsprite.bmp contains the game sprites
DXAPP_SPRITES   BITMAP   wbsprite.bmp
```

```
// wbsplash.bmp contains the game splash screen
DXAPP_SPLASH    BITMAP   wbsplash.bmp

// wroxblox.ico is the game's icon
DXAPP_ICON      ICON     wroxblox.ico
```

CDirectXSprite: An Animated Sprite Object

Earlier in the chapter, we defined what a sprite was. Now let's decide what we want in our sprite object. Basically, we want the sprite object to manage a piece of an off-screen surface to be used for drawing an animated image on the screen. We need to be able to draw, animate, hide, show and move the sprite. The sprite needs to keep track of its own position and other states. We also want the sprite to be able to detect collisions between itself and other sprites, and with the edges of the screen.

Sprite Cel Animation

We'll animate our sprites by displaying a sequence of separate images for them, which, like the frames of a film, will make them look as if they're moving. To do this, for each animated sprite, we'll put a string of equal-sized images, or **cels**, right next to each other in the sprite bitmap. Then we'll tell the sprite that its piece of the sprite bitmap encompasses the whole string of cels, and we'll tell it how big one cel is. Then the sprite will draw just one cel for itself. When we tell it to animate, it will move to the next cel.

Sprite Motion

Each sprite must know its position on the screen, which makes moving it easy—just put it in a new position. It would be nicer, however, if we could do a little more than just that. We're going to give each sprite an X and Y velocity, and when we tell it to move, it will increment its X and Y position by those amounts. To make it more useful, we'll store the velocity in tenths of a pixel to give it finer resolution.

Sprite Collisions

There are all kinds of tricky ways to do detect collisions, but we're not going to delve into any of them here. For the sake of simplicity, we're going to go with the simplest possible: rectangle intersection. Basically, when two sprites collide, we'll look at the intersection of their bounding rectangles to determine what kind of collision occurred.

Sprite Aging

We need to be able to hide and show sprites, and it's sometimes useful to be able to have some kind of countdown until a sprite disappears. We'll call this **aging**. A good example of this is when a sprite is supposed to explode after it's been by a laser blast. The sprite must first switch over to the explosion sprite images, then, when the last cel of the explosion has been drawn, it should disappear. Setting its aging to equal the number of cels in the explosion sequence would accomplish this.

Rendering a Sprite on a Surface

All we need to do to draw the sprite is to `BltFast()` its current cel image to the back buffer with source transparency. When we were building the `CDirectXApp` object, there was a member in there called `renderSprite()`, which turned around and called a `drawOn()` function in `CDirectXSprite`. The `drawOn()` function will just do a `BltFast()` from the sprite's off-screen surface to the back buffer.

CDirectXSprite Declaration

Here's the class declaration for `CDirectXSprite`:

```
// Sprite velocity and internal position registers will be in tenths of a
// Pixel, but we could change that by putting something else here
#define VELOCITY_PRECISION   10

// Codes to indicate whether move() triggered an edge bounce
```

```
#define BOUNCED_TOP       0x0001
#define BOUNCED_BOTTOM    0x0002
#define BOUNCED_LEFT      0x0004
#define BOUNCED_RIGHT     0x0008


class CDirectXSprite
{
protected:

   // Source
   LPDIRECTDRAWSURFACE     lpDDSSource;       // Source surface
   int    nSrcX; // X position of first frame of sprite on source surface
   int    nSrcY; // Y position of first frame of sprite on source surface
   int    nSrcW; // Width of frames of sprite
   int    nSrcH; // Height of frames of sprite

   //Animation
   UINT   nNumFrames;    // Total number of frames that make up sprite
                         // - must all be on one row
   UINT   nCurFrame;     // Current frame to display on next flip

   // Display Position
   int      nDisplayX;   // Position on display surface to draw
                         // sprite, * 10 - ie, 6000 = display at 600
   int      nDisplayY;   // Position on display surface to draw
                         // sprite, * 10 - ie, 4000 = display at 400

   // Motion
   int      nXVelocity;  // Increment to add to pixel position on
                         // each move (in tenths)
   int      nYVelocity;  // Increment to add to pixel position on
                         //each move (in tenths)
   BOOL   bBounceTop;    // Does sprite bounce at this edge, or run off?
   BOOL   bBounceBottom; // Does sprite bounce at this edge, or run off?
   BOOL   bBounceLeft;   // Does sprite bounce at this edge, or run off?
   BOOL   bBounceRight;  // Does sprite bounce at this edge, or run off?

   // State
   int    nLife;   // -1 = "immortal", 0 = dead (don't process sprite),
                   // n = number of frames til dead

public:

   CDirectXSprite ()
   {
      lpDDSSource = 0;
      nSrcX = 0;
      nSrcY = 480;
      nSrcW = 64;
      nSrcH = 64;

      nNumFrames = 1;
      nCurFrame = 0;

      nDisplayX = 0;
      nDisplayY = 0;

      nXVelocity = 0;
      nYVelocity = 0;
      bBounceTop = FALSE;
      bBounceBottom = FALSE;
      bBounceLeft = FALSE;
      bBounceRight = FALSE;

      nLife = -1;
   }


   void setSource (LPDIRECTDRAWSURFACE lpSrc, UINT nX, UINT nY,
      UINT nWidth, UINT nHeight)
```

```
{
   lpDDSSource = lpSrc;
   nSrcX = nX;
   nSrcY = nY;
   nSrcW = nWidth;
   nSrcH = nHeight;
}

int spriteWidth ()  {return nSrcW;}
int spriteHeight () {return nSrcH;}


UINT numFrames () {return nNumFrames;}
void setNumFrames (UINT nNewFrames) {nNumFrames = nNewFrames;}

// 0-based frame index
UINT curFrame () {return nCurFrame;}
void setCurFrame (UINT nNewCur) {nCurFrame = nNewCur;}

void advanceFrame ()
{
   nCurFrame ++;

   // Loop back to beginning if necessary
   if (nCurFrame == nNumFrames)
      nCurFrame = 0;
}


// Values are in pixels
int displayX()   {return nDisplayX/VELOCITY_PRECISION;}
int displayY()   {return nDisplayY/VELOCITY_PRECISION;}
void setDisplayX(int nNewX) {nDisplayX = nNewX * VELOCITY_PRECISION;}
void setDisplayY(int nNewY) {nDisplayY = nNewY * VELOCITY_PRECISION;}

// Values are in velocity precision
int velocityX () {return nXVelocity;}
int velocityY () {return nYVelocity;}
void setVelocityX (int nNewVX) {nXVelocity = nNewVX;}
void setVelocityY (int nNewVY) {nYVelocity = nNewVY;}

BOOL bounceTop ()      {return bBounceTop;}
BOOL bounceBottom ()   {return bBounceBottom;}
BOOL bounceLeft   ()      {return bBounceLeft;}
BOOL bounceRight ()      {return bBounceRight;}
void setBounceTop (BOOL bNewBounce=TRUE)
   {bBounceTop = bNewBounce;}
void setBounceBottom (BOOL bNewBounce=TRUE)
   {bBounceBottom = bNewBounce;}
void setBounceLeft (BOOL bNewBounce=TRUE)
   {bBounceLeft = bNewBounce;}
void setBounceRight (BOOL bNewBounce=TRUE)
   {bBounceRight = bNewBounce;}

int life () {return nLife;}
void setLife (int nNewLife) {nLife = nNewLife;}
void age () {if (nLife > 0) nLife--;}


UINT move ();   // Update the position by adding the velocities,
                // and bouncing if necessary
                // Returns bounce edge code if bounce occurred

UINT collidesWith (CDirectXSprite& other_sprite);
// Returns bounce code if other sprite overlaps this one

// Render sprite on the given surface
// Usually, you'll use CDirectXApp::renderSprite() (which calls this)
HRESULT   drawOn (LPDIRECTDRAWSURFACE   lpSurf)
{
   RECT   rcRect;
```

```
        rcRect.top = nSrcY;
        rcRect.bottom = nSrcY + nSrcH;
        rcRect.left = nSrcX + (curFrame()*nSrcW);
        rcRect.right = nSrcX + nSrcW + (curFrame()*nSrcW);

        return lpSurf->BltFast( displayX(), displayY(), lpDDSSource,
           &rcRect, DDBLTFAST_SRCCOLORKEY|DDBLTFAST_WAIT );
     }
};
```

OK, now here are the **move()** and **collidesWith()** functions:

```
UINT CDirectXSprite :: collidesWith (CDirectXSprite& other_sprite)
{

   // Them
   RECT rThem = {other_sprite.displayX(),other_sprite.displayY(),
      other_sprite.displayX()+other_sprite.spriteWidth(),
      other_sprite.displayY()+other_sprite.spriteHeight()};

   // Us
   RECT rUs = {displayX(), displayY(), displayX()+spriteWidth(),
      displayY()+spriteHeight()};

   RECT isect;

   if (IntersectRect (&isect, &rThem, &rUs ))
   {
      // Collision occurred - set bounce codes
      // Returns BOUNCED_LEFT if bounce is mostly along a vert. edge
      // Returns BOUNCED_TOP if bounce is mostly along a horiz. edge

      if ( (isect.right - isect.left) > (isect.bottom - isect.top) )
         return BOUNCED_TOP;
      else
         return BOUNCED_LEFT;
   }

   // No bounce
   return 0;
}


UINT CDirectXSprite :: move ()
{
   UINT bounced = 0;

   // Update position
   nDisplayX += nXVelocity;
   nDisplayY += nYVelocity;


   // Check bounce

   if (bounceTop())
   {
      if (displayY() < 0)
      {
         // Invert Y velocity
         nYVelocity = -nYVelocity;

         // Reflect
         nDisplayY = -nDisplayY;

         bounced = BOUNCED_TOP;
      }
   }

   if (bounceBottom())
```

```
    {
       if ((displayY()+nSrcH) > 480)
       {
          // Invert Y velocity
          nYVelocity = -nYVelocity;

          // Back off
          setDisplayY(480-nSrcH);

          bounced = BOUNCED_BOTTOM;
       }
    }

    if (bounceLeft())
    {
       if (displayX() < 0)
       {
          // Invert X velocity
          nXVelocity = -nXVelocity;

          // Reflect
          nDisplayX = -nDisplayX;

          bounced = BOUNCED_LEFT;
       }
    }

    if (bounceRight())
    {
       if ((displayX()+nSrcW) > 640)
       {
          // Invert X velocity
          nXVelocity = -nXVelocity;

          // Back off
          setDisplayX(640-nSrcW);

          bounced = BOUNCED_RIGHT;
       }
    }

    return bounced;
}
```

CDirectXSound: A Sound Object

The last object we'll build is a wrapper around a wave audio sound. We'll use the **SndObj** stuff from `Dsutil.h` make this really easy. Here's the class declaration for our **CDirectXSound** class:

```
class CDirectXSound
{
protected:
   LPDIRECTSOUND   m_pDSound;   // "Parent" DirectSound object
   HSNDOBJ         m_hSnd;      // Sound handle

public:

   CDirectXSound ()
   {
      m_pDSound = NULL;
      m_hSnd = NULL;
   }

   BOOL init (LPDIRECTSOUND pDS, LPCSTR szName, int nConcurrent = 1);

   // pan=0 - centered pan=-10,000 - way left pan=+10,000 - way right
   BOOL play (int pan = 0, DWORD dwFlags = 0);

   void destroy ();
};
```

Here are the function definitions:

```
BOOL CDirectXSound :: init (LPDIRECTSOUND pDS, LPCSTR szName, int nConcurrent)
{
   // Remember "parent" IDirectSound
   m_pDSound = pDS;

   // Attempt to create
   m_hSnd = SndObjCreate (pDS, szName, nConcurrent);

   return (m_hSnd != NULL);
}


void CDirectXSound :: destroy ()
{
   // Clean up
   SndObjDestroy (m_hSnd);
   m_hSnd = NULL;
   m_pDSound = NULL;
}


BOOL CDirectXSound :: play (int pan, DWORD dwFlags)
{
   // Styled after DSUtil's SndObjPlay(), but with panning

   // Only try to play if we have a sound object
    if (m_hSnd != NULL)
   {
      // If looping, don't replay
      if ((!(dwFlags & DSBPLAY_LOOPING) || (m_hSnd->iAlloc == 1)))
      {
         // Get a buffer to play with
         IDirectSoundBuffer *pDSB = SndObjGetFreeBuffer(m_hSnd);

         if (pDSB != NULL)
         {
            // Set panning
            pDSB->SetPan(pan);

            // Play
            return (pDSB->Play( 0, 0, dwFlags) == DS_OK);
         }
      }
   }

    return FALSE;
}
```

Pretty simple.

Fitting It All Together

To use these classes, you'll need to do the things listed at the end of the section above on **`CDirectXApp`**, plus:

Declare a global **`CDirectXApp`** object for your app.
Build a **`WinMain()`** function.

It will typically look something like this:

```
// The game object - type is descendant class from CDirectXApp
CMyGame   theGame;

int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
   LPSTR lpCmdLine, int nCmdShow)
```

```
{
    // Initialize the game object
   if (!theGame.setupGame(hInstance, nCmdShow))
      return FALSE;

   // Message Loop
   int retval = theGame.go();

   // Going down - clean up the game object
   theGame.stopGame();

   return retval;
}
```

setupGame() and **stopGame()** are members of the descendant class that presumably turn around and call **CDirectXApp::init()** and **CDirectXApp::shutdown()** respectively. The **setupGame()** and **stopGame()** functions might also do other game-specific stuff, like setting up **CDirectXSprites** and **CDirectXSounds** for the game to use. The descendant app class can own **CDirectXSprites** and **CDirectXSounds** as members for use by the **gameSlice()** function.

WROXBlox! Application Architecture

Now we have the C++ tools to build the WROXBlox! Application, let's map the concepts we've just covered to the WROXBlox! project. All the pesky DirectX stuff is out of our way now; the rest is game logic.

> *As the rest of the code for WROXBlox! is game logic, we'll leave out the code here. Have a look on the CD if you're interested.*

Subclassing CDirectXApp: CWroxGame

Naturally, the first thing we'll want to do is make a subclass of **CDirectXApp** for the WROXBlox! Game. We'll call it **CWroxGame**.

There are basically three kinds of members that we're going to put in **CWroxGame**:

- Any **CDirectXSprite** objects the game needs.
- Any **CDirectXSound** objects the game needs.
- Any variables necessary for the game logic (the state, for instance).

We're also going to need to override the **gameSlice()** function, and we'll want to add some functions for initializing and destroying the game elements. Take a look back at the state diagram for WROXBlox! that we laid out earlier in the chapter. There's some stuff that we'll also want to add to **CWroxGame** to manage the game states.

The WROXBlox! DXAppWndProc() Function

We're relying on Windows messages for a few things, which are, of course, dealt with in the window procedure:

```
long FAR PASCAL   DXAppWndProc (HWND hWnd, UINT msg, WPARAM wParam,
   LPARAM lParam)
{
   switch( msg )
   {
      // Keep the cursor turned off
      case WM_SETCURSOR:
         SetCursor(NULL);
         return TRUE;

      case WM_KEYDOWN:
         switch( wParam )
         {
            // Exit game if either Esc or F12 is pressed
            case VK_ESCAPE:
```

```
            case VK_F12:
               PostMessage(hWnd, WM_CLOSE, 0, 0);
               break;

            // Otherwise, see if game has a use for it
            default:
               theGame.processKey(wParam);
               break;
         }
         break;

      // Timer is used to delay between game states
      case WM_TIMER:
         theGame.processTimer();
         break;


      case WM_DESTROY:
         // Going down - deactivate game
         theGame.setActive(FALSE);
         PostQuitMessage( 0 );
         break;
   }

   return DefWindowProc(hWnd, msg, wParam, lParam);
}
```

Summary

In this chapter, we've surveyed the basics of using the DirectX API, including DirectDraw, DirectSound, DirectPlay, and DirectInput. We wrote a complete game in C++, and an example program using the DirectPlay API.

Network Programming

Introduction

Windows NT is built from the ground up for networking, which means that it supports quite a number of protocol stacks out of the box. These protocol stacks include: NetBEUI, DLC (Data Link Control), NWLink (NetWare Link), TCP/IP, etc. However, a protocol stack is not the same as a programming interface; for example, NetBEUI and NetBIOS are not the same thing. NetBEUI is the protocol stack, while NetBIOS is the programming interface. In fact, the NetBIOS programming interface can be provided by protocols other than NetBEUI. When we talk about network programming, we are talking about writing distributed applications using Windows NT network programming interfaces. A distributed (or client-server, if you like the buzz word) application consists of two parts: a server, which provides some services, and a client, which handles user interaction. The two parts usually, but not necessarily, reside on different computers connected via a network. For them to communicate with each other, we need to use the network programming interfaces, which are sometimes also referred to as IPC (Inter-process Communication) mechanisms, that Windows NT provides.

> *At the time of writing, some of the functionality described in this chapter for named pipes and mailslots are specific to Windows NT. Unfortunately this is also true of the code. If you wish to use the code on Windows 95, feel free to modify the programs.*

IPC Mechanisms

Windows NT provides many IPC mechanisms: NetBIOS, Windows Sockets (TCP/IP), NetDDE, RPC (Remote Procedure Call), named pipes and mailslots. Let's look briefly at each one.

NetBIOS

NetBIOS first appeared with the IBM PC Network adapter card in August 1984. The IBM PC Network is IBM's first LAN. Since then, NetBIOS has become a standard programming interface in the PC environment for developing client-server applications. A NetBIOS client-server application can communicate over various protocols, e.g. NetBEUI protocol (NBF), NWLink NetBIOS (NWNBLink), and NetBIOS over TCP/IP (NetBT). Although other newer IPCs, e.g. RPC, named pipes, etc., have surpassed NetBIOS in flexibility, portability and ease of programming, NetBIOS is still provided for backward compatibility.

WinSock

The Windows Sockets API is an independent specification developed through cooperation of many software vendors, including Microsoft, NetManage, etc., to provide a standard for a network programming interface based on the popular Berkeley Software Distribution (BSD) sockets interface, developed at the University of California, Berkeley in the early 1980s. The intention is that Windows Sockets is used as the standard networking API for all Windows platforms. The continuing growth in the use of TCP/IP networking makes Windows Sockets a very popular IPC to use. We'll cover Windows Sockets in detail in the next chapter.

NetDDE

NetDDE is an extension of Windows' Dynamic Data Exchange (DDE) which provides information-sharing capabilities between applications running on two computers across the network.

RPC

Remote Procedure Call (RPC) was originally developed by Sun Microsystems. It has been adopted by the Open Software Foundation (OSF) as part of the Distributed Computing Environment (DCE) specification and is supported by most of the major computer vendors. Windows NT's RPC implementation is inter-operable with other DCE-based systems. The RPC mechanism allows a client application using a specially compiled 'stub' library to find a server which can execute a function remotely on its behalf and passes the function and data to the server's RPC Runtime. When the function is completed by the server, the data and results are sent back to the client application. The interesting part of RPC is that it can use other IPC mechanisms to establish communications between the computers on which the client and the server applications run. If the client and server are on the same computer, the Local Procedure Call (LPC) mechanism can be used instead, making RPC extremely flexible and portable.

Named Pipes and Mailslots

Named pipes and mailslots were first introduced with OS/2 and the Microsoft LAN Manager. Unlike other IPCs, they are implemented as file systems. As such, they share the same security and access control as file systems, such as NTFS, which is unique among the IPCs we've mentioned so far. Named pipes provide connection-oriented communication between applications running on the same computer, or on different computers connected by a network. Windows NT's implementation of mailslot provides a connection-less communication mechanism that supports broadcasts. They are provided for backward compatibility with existing Microsoft LAN Manager and IBM LAN Server applications and other applications such as the Microsoft SQL Server.

This chapter concentrates on writing client-server applications using named pipes, mailslots and NetBIOS. Named pipes and mailslots are selected because they're simple and easy to use due to their file-like API and their inter-operability with existing LAN Manager and LAN Server applications. NetBIOS is chosen to contrast the programming interface used by mailslots and named pipes.

Named Pipes

Named pipes provide a two-way connection-oriented communication channel between applications running either on the same machine or on different machines connected over a network. Windows NT's named pipe implementation is based on that of OS/2, with enhancements in asynchronous support and increased security. The most interesting enhancement is impersonation, which allows a server to change its security identity to that of the client's. For example, suppose a database server system uses named pipes to receive read/write requests from clients. When a request comes in, the database server program can change its identity to that of the client (i.e., impersonate the client) before attempting to perform the request. This allows better security control on the data in the database by denying clients with insufficient security clearances to access sensitive data. The named pipe API functions can be divided into server, client and general categories:

Category	Function	Description
Server	`CreateNamedPipe()`	Creates a server instance of a named pipe.
	`ConnectNamedPipe()`	Waits for a client connection.
	`DisconnectNamedPipe()`	Disconnects an instance of a named pipe from a client.
Client	`WaitNamedPipe()`	Waits for a server named pipe to become available.
	`CallNamedPipe()`	Connects, writes, reads and closes the named pipe.
	`TransactNamedPipe()`	Writes to and reads from a connected named pipe.
General	`GetNamedPipeHandleState()`	Retrieves current information about a named pipe.
	`GetNamedPipeInfo()`	Retrieves the named pipe characteristics.
	`PeekNamedPipe()`	Copies but leaves data intact in the named pipe.
	`SetNamedPipeHandleState()`	Sets read mode and blocking mode of a named pipe.

`ImpersonateNamedPipeClient()`	Assumes the identity of a client application.
`RevertToSelf()`	Resumes its own identity.

Opening a client-side named pipe, reading/writing and closing of a named pipe are performed using normal file operations: **`CreateFile()`**, **`ReadFile()`**, **`WriteFile()`** and **`CloseHandle()`** respectively.

Creating a Connection

In order to establish a connection using a named pipe, the server and the client side do things differently. The client side instigates a connection.

Server Side

The server takes the following steps:

1 The server creates an instance of a named pipe by calling **`CreateNamedPipe()`** with the desired characteristics, e.g. the maximum number of concurrent instances supported, input/output buffer size, etc. (more on the attributes later).

2 The server then waits for a client to connect to the named pipe by calling **`ConnectNamedPipe()`**.

3 Once the connection has been established, the client requests the server to perform one or more functions on its behalf.

4 The server breaks the connection by calling **`DisconnectNamedPipe()`**. After this, the server may either go back to Step 2 (i.e. wait for another connection), or carry on.

5 The server destroys the named pipe instance by calling **`CloseHandle()`**.

You should note that Windows 95 doesn't support server-side named pipe functions. In other words, Windows 95 applications can only be clients. Also, Windows 95 named pipes don't support overlapped I/O operations (to be discussed later).

A Wrapper for Named Pipes

Designing a wrapper for named pipes, or any IPC, is not easy. Depending on what you want to achieve, the resultant class or classes may look very different when designed by two individuals. For example, one may choose to encapsulate all named pipe functions in a base class. Another person may want the base class to encapsulate the common functions of different connection-oriented IPCs. If you're faced with different designs, it's always good to take a common denominator approach, which means that you don't provide all functions of an IPC in a class or classes like this. I'm designing the wrapper classes for use by named pipes, mailslots and NetBIOS, so I'm taking the latter approach, i.e. not including all named pipe functions in the wrapper classes. In fact, these classes only provide a minimal set of file-like operations, namely: **`Open()`**, **`Read()`**, **`Write()`** and **`Close()`**.

At first glance, these classes appear very limited in the tasks that they can do, because they don't provide all the available functions of any IPC. On closer examination, though, you'll see that, since C++ is an object-oriented language, if you need to, it's easy to derive a new class from the wrapper classes to provide the extra functions specific to a certain IPC . On the other hand, if the user just needs the four basic file-like operations as they write an application, these classes give him or her the potential to change the application's transport mechanism from one to another with minimal fuss, because no functions specific to a particular IPC have been used.

`CCommBase` is an abstract class, upon which the named pipe and mailslot classes are derived. The four major

methods provided by these classes are: **Open()**, **Read()**, **Write()** and **Close()**, which you're already familiar with. To use one of these classes for communication, you just need to **Open()** the IPC, use **Read()** and **Write()** to perform I\O and **Close()** the IPC when you're done. Initially, I was tempted to use the MFC **CFile** class as the base class to derive my mailslot and named pipe classes, but I looked into it and found that **CFile** contains a lot of member functions, such as **Seek()**, **Remove()**, **Rename()**, etc., which are meaningless for mailslots and named pipes. If I used **CFile** as base, I would have to override all these member functions to throw an exception, like the MFC socket classes do. In a word: messy.

Read on for a brief description of these classes.

CCommBase

This is the abstract base class. Its class definition is:

```
class CCommBase
{
protected:
   HANDLE m_Handle;
   BOOL m_InSession;
   BOOL m_Async;
   CString m_FileName;
   DWORD m_ErrorCode;
   OVERLAPPED m_ReadOl;
   OVERLAPPED m_WriteOl;

public:
   CCommBase();
   CCommBase(BOOL fUseOverlap);

   ~CCommBase();

   inline DWORD GetError() { return m_ErrorCode; };
   inline BOOL IsConnected() { return m_InSession; };

   virtual BOOL Open(const char* pszFileName, UINT nOpenFlags = 0) = 0;
   virtual UINT Read(void* pBuf, UINT nCount);
   virtual UINT Write(const void* pBuf, UINT nCount);
   virtual void Close();

};
```

It provides the implementation of the three basic file operations: **Read()**, **Write()** and **Close()**. These functions are all synchronous, i.e. they don't return control to the caller until the operation is completed, even if an overlapped operation is specified in the constructor. When **USE_OVERLAP** (defined in `Ccommba.h`) is specified, **Read()** and **Write()** use an event object in an **OVERLAPPED** structure to wait until the operation is complete. The reason for using **USE_OVERLAP** is that it allows concurrent read and write operations. If **USE_OVERLAP** is not specified, I/O operations are serialized. For example, if a **Read()** is issued in one thread before a **Write()** in another (without using the **USE_OVERLAP** mode), the **Write()** will not return until the **Read()** completes. If there's no incoming data, both operations block. This may not be what you would expect or want, especially if you have a lot of data to send but little to receive. All named pipe classes use **USE_OVERLAP**.

When **USE_OVERLAP** is specified, the constructor creates **OVERLAPPED** structures for subsequent **Read()** and **Write()** operations. Each **OVERLAPPED** structure contains an event object for synchronization. The **CCommbase** constructors are shown below:

```
CCommBase::CCommBase()
{
   //initialize class data members
   m_InSession = FALSE;
   m_Handle = INVALID_HANDLE_VALUE;
   m_ErrorCode = 0;
   m_Async = USE_NO_OVERLAP;
```

```
}

CCommBase::CCommBase(BOOL fUseOverlap)
{
   //initialize class data members
   m_InSession = FALSE;
   m_Handle = INVALID_HANDLE_VALUE;
   m_ErrorCode = 0;

   //create overlap structure and event objects
   //if overlapped i/o is to be used
   if ((m_Async = fUseOverlap) == USE_OVERLAP)
   {
      memset(&m_ReadOl, 0, sizeof(OVERLAPPED));
      memset(&m_WriteOl, 0, sizeof(OVERLAPPED));
      m_ReadOl.hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
      m_WriteOl.hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
   }
};
```

The implementation of **`Read()`** and **`Write()`** uses **`ReadFile()`** and **`WriteFile()`** respectively:

```
UINT CCommBase::Read(void* pBuf, UINT nCount)
{
   ULONG bytesRead;
   BOOL retCode;

   //read using either overlapped or non-overlapped i/o
   retCode = ReadFile(m_Handle, pBuf, nCount,&bytesRead,
            (m_Async)? &m_ReadOl: NULL);
   if (!retCode)
   {
      if ((m_ErrorCode = GetLastError()) == ERROR_IO_PENDING)
      {
         //wait till operation is complete: overlapped i/o only
         WaitForSingleObject(m_ReadOl.hEvent, INFINITE);
         //get the i/o results
         GetOverlappedResult (m_Handle, &m_ReadOl, &bytesRead, FALSE);
         return bytesRead;
      }

      return 0;
   }
   else
      return bytesRead;
};

UINT CCommBase::Write(const void* pBuf, UINT nCount)
{
   ULONG bytesWritten;
   char *buffer = (char *) pBuf;

   //write using either overlapped or non-overlapped i/o
   if (!WriteFile(m_Handle, pBuf, nCount, &bytesWritten,
         (m_Async)? &m_WriteOl: NULL))
   {
      if ((m_ErrorCode = GetLastError()) == ERROR_IO_PENDING)
      {
         //wait till operation is complete: overlapped i/o only
         WaitForSingleObject(m_WriteOl.hEvent, INFINITE);
         //get the i/o results
         GetOverlappedResult (m_Handle, &m_WriteOl, &bytesWritten, FALSE);
         return bytesWritten;
      }
      return 0;
   }
   else
      return bytesWritten;
};
```

When **USE_OVERLAP** is specified, **Read()** and **Write()** pass the address of their **OVERLAPPED** structure to **ReadFile()** and **WriteFile()** respectively to carry out asynchronous I/O. The code checks that the operation is pending and waits until it finishes by blocking at **WaitForSingleObject()** on the **OVERAPPED** structure's event object. When the event object is signaled, **GetOverlappedResult()** is used to retrieve the operation results.

CNamedPipeServer

CNamedPipeServer is derived from **CCommBase**. Its class definition is:

```
class CNamedPipeServer: public CCommBase
{
public:
   CNamedPipeServer();
   ~CNamedPipeServer();
   virtual BOOL Open(const char* pszFileName, UINT nOpenFlags = 0);
};
```

It implements the virtual member function **Open()**. During **Open()**, it creates a server named pipe and waits for a pipe connection by calling **CreateNamePipe()** and **ConnectNamedPipe()** respectively. It returns only when either an error occurs or when a connection is made.

```
BOOL CNamedPipeServer::Open(const char* pszFileName, UINT nOpenFlags)
{
   //check if a name has been specified
   if (!pszFileName)
      return FALSE;

   //create server-side named pipe
   m_Handle = CreateNamedPipe (pszFileName,
      (nOpenFlags)? nOpenFlags:
      PIPE_ACCESS_DUPLEX | FILE_FLAG_OVERLAPPED,
      PIPE_WAIT | PIPE_READMODE_MESSAGE | PIPE_TYPE_MESSAGE,
      PIPE_UNLIMITED_INSTANCES,
      SEND_BUF_SIZE,
      RECV_BUF_SIZE,
      NAMEDPIPE_TIME_OUT,
      NULL);

   //retrieve error code if any
   if (m_Handle == INVALID_HANDLE_VALUE)
   {
      m_ErrorCode = GetLastError();
      return FALSE;            // flag is not set here.
   };

   //save pipe name
   m_FileName = pszFileName;

   //prepare for overlapped i/o
   OVERLAPPED ol;
   memset(&ol, 0, sizeof(OVERLAPPED));
   ol.hEvent = CreateEvent(NULL, FALSE, TRUE, NULL);

   //wait for pipe connection
   BOOL fResult = TRUE;
   if (!ConnectNamedPipe(m_Handle, &ol))
   {
      if ((m_ErrorCode = GetLastError()) == ERROR_IO_PENDING)
      {
         //keep waiting
         WaitForSingleObject(ol.hEvent, INFINITE);

         //get result
         DWORD bytesRead;
         if (!(fResult = GetOverlappedResult(m_Handle, &ol,
```

```
                &bytesRead, FALSE)))
            {
                m_ErrorCode = GetLastError();
                fResult = FALSE;
            }
        }
    }
    else
    {
        m_InSession = TRUE;
    }

    CloseHandle(ol.hEvent);

    return fResult;
}
```

Creating the Named Pipe

The **CreateNamedPipe()** function creates an instance of a named pipe and returns a handle for subsequent pipe operations. A server application uses this function both to create a named pipe with the specified attributes and to create a new instance of an existing named pipe. Creating a named pipe requires a lot of information to be specified, so let's take a closer look at **CreateNamedPipe()**.

```
HANDLE CreateNamedPipe(
   LPCTSTR    lpName,          // address of pipe name
   DWORD   dwOpenMode,         // pipe open mode
   DWORD   dwPipeMode,         // pipe-specific modes
   DWORD   nMaxInstances,      // maximum number of instances
   DWORD   nOutBufferSize,     // output buffer size, in bytes
   DWORD   nInBufferSize,      // input buffer size, in bytes
   DWORD   nDefaultTimeOut,    // time-out time, in milliseconds
   LPSECURITY_ATTRIBUTES
      lpSecurityAttributes     // address of security attributes structure
   );
```

The parameters supplied to **CreateNamedPipe()** are as follows:

lpName is a pointer to a string containing the name of the pipe to be created. This name is in the form of:

```
\\.\pipe\pipename
```

A name must start with **\\.\pipe**, followed by an identifying name. The whole pipe name is limited to 256 characters and is case-insensitive. The name that you use can have any character other than a backlash (\), e.g.:

```
\\.\pipe\mynamedpipe
\\.\pipe\datapipe
```

The period (.) in the above example represents the name of the server machine. A server can create a named pipe instance using the period to represent the local machine. However, for a client to connect to a remote server, it must replace the period with the name of the server using either **CreateFile()** or **CallNamedPipe()**.

dwOpenMode is a flag that specifies the access mode defining the direction in which the information flows through a named pipe. The choices are:

Flag	Meaning
PIPE_ACCESS_INBOUND	The server reads and the client writes (half duplex).
PIPE_ACCESS_OUTBOUND	The server writes and the client reads (half duplex).
PIPE_ACCESS_DUPLEX	Read/write is allowed in both directions (full duplex).

You can also add the following characteristics to a named pipe:

Flag	Meaning
FILE_FLAG_WRITE_THROUGH	Disables buffering over a network. Doing so usually slows down the pipe.
FILE_FLAG_OVERLAPPED	Enables asynchronous read and write operations. For example, a write operation returns immediately, while the operation continues in the background. You can use a **WriteFile()** and specify an event object to be signaled on completion, or use **WriteFileEx()** to register a callback function which is called when the operation completes.

The next parameter, **dwPipeMode**, is a combination of flags which specify the read, write and wait modes of the pipe. The possible write flags are:

Flag	Meaning
PIPE_TYPE_BYTE	Specifies that the pipe should operate in byte mode, i.e. data is written to the pipe as a stream of bytes.
PIPE_TYPE_MESSAGE	Specifies that the pipe should operate in message mode, i.e. each write command sends just one, complete message. An invisible header specifying the length of the message is inserted in the beginning of the message and removed automatically once it's been read.

The read mode flags have the same meaning as write mode flags, but are for the read operation instead. Possible values are **PIPE_READMODE_BYTE** and **PIPE_READMODE_MESSAGE**.

The wait mode flags define whether or not a read operation from an empty pipe will cause the thread issuing it to block. Possible values are **PIPE_WAIT** and **PIPE_NOWAIT**. **PIPE_WAIT** is the default.

nMaxInstances defines the maximum instances of a named pipe. A **CreateNamedPipe()** function fails if it's called after the maximum number of instances have already been created. **PIPE_UNLIMITED_INSTANCES** specifies that there are no instance limitations.

nOutBufferSize and **nInBufferSize** define the initial buffer size for a pipe. If the buffer sizes are defined too small, performance degrades.

nDefaultTimeOut specifies the default timeout value in milliseconds for a client-side named pipe which calls the **WaitNamedPipe()** function using a timeout value of **NMPWAIT_USE_DEFAULT_WAIT**.

Finally, since named pipes are implemented as a file system, a **SECURITY_ATTRIBUTES** structure can be specified. We simply use **NULL** for this so that the default security descriptor is used.

Waiting for the Client

After you have created the named pipe and set up the **OVERLAPPED** structure, if you use **USE_OVERLAP**, **Open()** calls **ConnectNamedPipe()** to wait for an incoming connection. The **ConnectNamedPipe()** function enables a named pipe server to wait for a client process to connect to an instance of a named pipe. A client process connects by calling either the **CreateFile()** or **CallNamedPipe()** function.

```
BOOL ConnectNamedPipe(
   HANDLE   hNamedPipe,           // handle of named pipe to connect
   LPOVERLAPPED    lpOverlapped   // address of overlapped structure
);
```

Closing the Connection

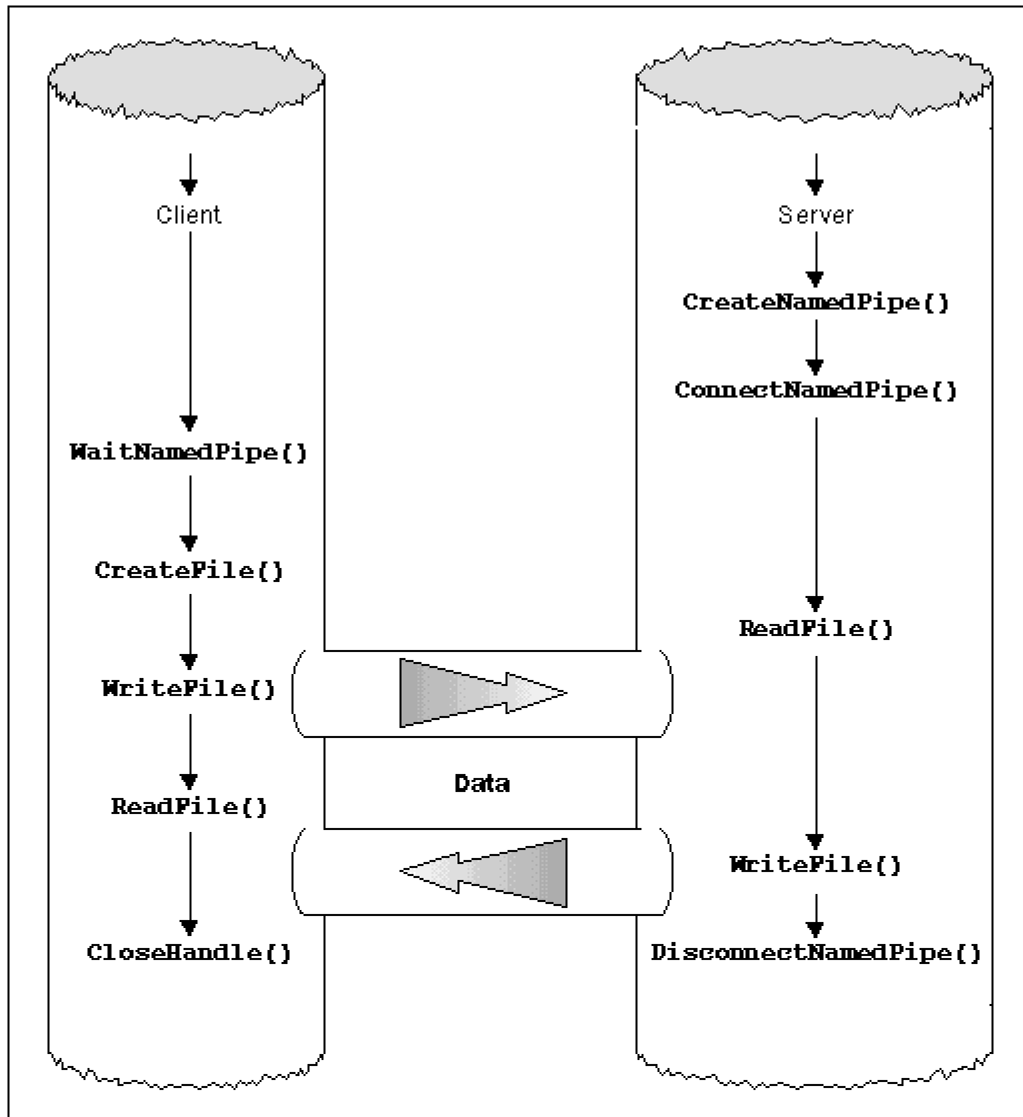
CNamedPipeServer doesn't use **DisconnectNamedPipe()** to force a connected client off. Instead, it uses **CloseHandle()**. The implication is that **CNamedPipeServer** cannot reuse the pipe handle for another connection, but has to create a new one for a new connection. This eliminates the house-keeping work involved in reusing a pipe handle, but introduces a slight overhead in creating a new instance of a named pipe every time a new connection is required.

Client Side

On the client side, the following steps are taken:

1. The client waits for the availability of a server named pipe instance by calling **WaitNamedPipe()**.To do this, the client must know the name of the server-side named pipe. If all the instances of the server-side named pipe are in use, the client either waits until an instance is available or times out, depending on the behavior specified in **WaitNamedPipe()**.
2. The client tries to connect to a named pipe by calling the file function, **CreateFile()**. Again, in order to do this, the client must know the name of the server-side named pipe.
3. The client makes one or more requests to the server when the connection is established.
4. After all processing has been completed, the client disconnects the named pipe and destroys the named pipe handle by calling **CloseHandle()**.

The interaction between the server and client-side applications is depicted in the following diagram:



CNamedPipeClient

CNamedPipeClient is derived from **CCommBase**. Its class definition is:

```
class CNamedPipeClient: public CCommBase
{
public:
   CNamedPipeClient();
   ~CNamedPipeClient();
   virtual BOOL Open(const char* pszFileName, UINT nOpenFlags = 0);
};
```

It implements the virtual member function **Open()**. During **Open()**, it waits for a server named pipe to become available by calling **WaitNamePipe()** before connecting to it using **CreateFile()**. **Open()** returns only when an error or timeout occurs, or when a connection is made.

```
BOOL CNamedPipeClient::Open(const char* pszFileName, UINT nOpenFlags)
{
```

```
    //check if a name has been specified
    if (!pszFileName)
       return FALSE;

    //wait for a server named pipe to become available
    if (!WaitNamedPipe(pszFileName, NAMEDPIPE_TIME_OUT))
    {
       m_ErrorCode = GetLastError();
       return FALSE;
    }

    //try to establish a pipe connection
    m_Handle = CreateFile (pszFileName,
             (nOpenFlags)? nOpenFlags:
             GENERIC_READ | GENERIC_WRITE,
             FILE_SHARE_WRITE ,
             NULL,
             OPEN_EXISTING,
             FILE_FLAG_OVERLAPPED,
             NULL);

    //retrieve error code if any
    if (m_Handle == INVALID_HANDLE_VALUE)
    {
       m_ErrorCode = GetLastError();
       return FALSE;
    };

    m_InSession = TRUE;
    m_FileName = pszFileName;

    return TRUE;
 };
```

Waiting on a Pipe

The **`WaitNamedPipe()`** function waits until either a timeout interval elapses or an instance of the specified server-side named pipe is available for connection. You can use **`NMPWAIT_USE_DEFAULT_WAIT`** as the **`dwTimeout`** value to use the default value defined for the pipe by **`CreateNamedPipe()`**.

```
BOOL WaitNamedPipe(
   LPCTSTR   lpNamedPipeName,    // address of name of pipe to wait for
   DWORD   dwTimeout             // time-out interval, in milliseconds
);
```

An Example of Using Named Pipes

Before we go through an example of using named pipes to build client-server applications we need to take a look at mailslots.

The reason for this is that named pipes are easy to use as long as you know the server's pipe name, but how do you get the name? You may have discovered by now that pipe names can be quite long and weird, so, as you'll see, combining them with mailslots makes them a little easier to use. The combined example shown later illustrates how these IPCs actually complement each other.

Mailslots

Unlike named pipes, which provide two-way connection-oriented communication between two parties, mailslots are designed to work in only one direction and are used in situations where many applications need to talk to the same application. Two classes of mailslots are supported by the Microsoft LAN Manager and IBM LAN Server: class 1 mailslots provide guaranteed delivery and class 2 mailslots provide a connection-less best-effort (i.e. delivery not guaranteed) delivery which supports broadcasts.

Windows NT only implements a subset of the Microsoft OS/2 LAN Manager mailslots API, i.e. class 2 mailslots. These are most useful for identifying other computers or services on a network, e.g. the Computer Browser service under Windows NT uses mailslots. We'll come back to using mailslots to discover services on the network later, but first, let's take a closer look at how mailslots are generally used.

When an application creates a mailslot, it receives a mailslot handle, which must be used when the application reads messages from the mailslot. The application that creates the mailslot is called the **mailslot server**. A mailslot exists until all server handles to it have been closed, or all server applications have terminated.

The more common server mailslot API functions include:

Function	Description
`CreateMailslot()`	Creates a mailslot and returns a mailslot handle.
`GetMailslotInfo()`	Retrieves the maximum message size, the mailslot size, the size of the next message in the mailslot, the number of messages in the mailslot and the amount of time a read operation can wait for a message.
`SetMailslotInfo()`	Changes the read timeout for a mailslot.

A **mailslot client** is an application that writes a message to a mailslot. Any application that knows the name of the mailslot can write messages into it. Messages are delivered on a first-come-first-served basis, i.e. without priority. In my opinion, the most important feature of mailslots is their ability to broadcast messages within a domain. If applications in a domain have each created a mailslot using similar names, every message that is broadcast using that name in the domain is received by each of the applications. Because one application can control both a server mailslot handle and the client handle retrieved when the mailslot is opened for a write operation, applications can easily implement a simple message-passing facility within a domain. Messages broadcast to a domain can be no larger than 400 bytes, although messages sent to an individual mailslot are limited only by the maximum message size specified by the creator of the mailslot (which can be unlimited). Because a mailslot is either send-only or receive-only, there's no need to use the `USE_OVERLAP` mode as in the named pipe classes.

A Wrapper for Mailslots

Many of the mailslot API functions are actually WIN32 file I/O functions, including `CreateFile()`, `ReadFile()`, `WriteFile()` and `CloseHandle()`. As you can see, these are, in fact, common with named pipes, which means that we'll derive the mailslot classes from `CCommBase`.

CMailSlotServer

`CMailSlotServer` is derived from `CCommBase`. Its class definition is:

```
class CMailSlotServer: public CCommBase
{
protected:
   DWORD m_Timeout;
```

```
public:
   CMailSlotServer();
   CMailSlotServer(DWORD timeout);
   ~CMailSlotServer();
   virtual BOOL Open(const char* pszFileName, UINT nOpenFlags = 0);
};
```

It implements the virtual member function **Open()**. During **Open()**, it creates a server mailslot by calling the **CreateMailslot()** API. It returns only when either an error occurs or a mailslot is created.

```
BOOL CMailSlotServer::Open(const char* pszFileName, UINT nOpenFlags)
{
   //check if a name has been specified
   if (!pszFileName)
      return FALSE;

   //create server mailslot
   m_Handle = CreateMailslot(pszFileName, BUF_SIZE, m_Timeout, NULL);

   //retrieve error if any
   if (m_Handle == INVALID_HANDLE_VALUE)
   {
      m_ErrorCode = GetLastError();
      return FALSE;
   };

   m_FileName = pszFileName;

   m_InSession = TRUE;
   return TRUE;
}
```

CreateMailslot()

The **CreateMailslot()** function creates a mailslot with the specified name and returns a handle that a mailslot server can use to perform operations on the mailslot. The mailslot is local to the computer that creates it. An error occurs if a mailslot with the specified name already exists.

```
HANDLE CreateMailslot(
   LPCTSTR lpName,           // address of string for mailslot name
   DWORD   nMaxMessageSize,  // maximum message size
   DWORD   lReadTimeout,     // milliseconds before read time-out
   LPSECURITY_ATTRIBUTES
      lpSecurityAttributes   // address of security structure
);
```

When an application creates a mailslot, **lpName** must specify a name. The mailslot name must have the following form:

```
\\.\mailslot\[path]\name
```

Mailslot names are case-insensitive. As with pipes, the name always starts with **\\.\mailslot**, followed by the name, optionally preceded by a path consisting of one or more pseudo-directory names, separated by backslashes. **nMaxMessageSize** is used to specify the maximum size, in bytes, of a message that can be written to the mailslot. A zero denotes a message of any size.

lReadTimeout specifies the amount of time, in milliseconds, that a read option will wait for a message to be written to the mailslot. Special values include zero, which denotes return immediately if there is no message, and **MAILSLOT_WAIT_FOREVER,** which denotes that the read operation should wait indefinitely.

lpSecurityAttributes specifies the security attributes to be used. A **NULL** can be used to denote default security where the handle is not inherited.

CMailSlotClient

CMailSlotClient is derived from **CCommBase**. Its class definition is:

```
class CMailSlotClient: public CCommBase
{
public:
   CMailSlotClient();
   ~CMailSlotClient();
   virtual BOOL Open(const char* pszFileName, UINT nOpenFlags = 0);
};
```

It implements the virtual member function **Open()**. During **Open()**, it opens the desired mailslot by calling **CreateFile()**. It returns when the operation completes or an error occurs.

```
BOOL CMailSlotClient::Open(const char* pszFileName, UINT nOpenFlags)
{
   //check if a name has been specified
   if (!pszFileName)
      return FALSE;

   //open a named mailslot
   m_Handle = CreateFile (pszFileName,
            (nOpenFlags)? nOpenFlags:
            GENERIC_READ | GENERIC_WRITE,
            FILE_SHARE_READ | FILE_SHARE_WRITE ,
            NULL,
            OPEN_EXISTING,
            FILE_ATTRIBUTE_NORMAL,
            NULL);

   //retrieve error if any
   if (m_Handle == INVALID_HANDLE_VALUE)
   {
      m_ErrorCode = GetLastError();
      return FALSE;
   };

   m_InSession = TRUE;
   m_FileName = pszFileName;

   return TRUE;
};
```

Writing Client-server Applications

Before we give you an example on how to use these mailslot and named pipe classes, we'll first take a general look at client-server applications. Unfortunately, the *term client-server* can mean different things to different people. Although people come up with a range of definitions, in general, it usually refers to a style of computing involving two logically related entities that cooperate over a network to accomplish a task. These two logically related entities are the **client** and the **server**.

The client, or the service requester, instigates communication to a server and asks for certain services to be performed by the server on its behalf. The server, or service provider, services the request and responds with the requested information. Normally, clients and servers communicate over a network, but there's no reason why they cannot be run on the same computer.

A client application usually provides a graphical user interface to the user, validates data entered by the user, sends requests to the server on the user's behalf and presents the response from the server to the user in the appropriate format.

A server application receives requests from the client, carries out the requests and sends the requested information back to the client. A server application is usually more complicated than a client application because it has to handle things like multiple clients, data integrity, access control, etc., which a client application doesn't need to worry about. A server application may not need a user interface at all, because that's usually handled by the client.

The reason that client-server computing has received so much attention recently comes from the industry trend towards **downsizing** (right-sizing) business applications to PCs and workstations. Downsizing refers to the process by which large mainframe and mini applications are broken up into smaller applications to run on one or more network servers. This trend has developed in response to the increased availability of inexpensive but powerful PC and workstation hardware. By converting existing large mainframe and mini applications into client-server applications, running on these networked PCs and workstations, downsizing brings with it both cost benefits (due to cheaper hardware and software development and maintenance) and the availability of computing power to the users in the form of desktop computers.

There are many kinds of client-server applications. One of the most common is a file server which accepts client requests for file records over a network. All network operating systems provide this service.

A database server accepts client SQL (Structured Query Language) requests and returns the data over the network. Instead of sending all data back to the client and letting it sort out the information itself, a database server processes the request and picks out only the information that the client requested. This saves network bandwidth and means that less powerful, inexpensive PCs or workstations can run client applications. Only the servers need to run on powerful hardware platforms.

To be able to talk to a server, a client must:

1. Find out where a particular service is being offered (there might be more than one server on the network).
2. Connect to the server.
3. Obtain the service.
4. Disconnect from the server when done.

Some applications, such as Telnet*,* do not go out and find all servers that they can connect to, but rather rely on the user to enter a valid address. This is because there are simply too many servers to report and trying to report them all usually involves broadcasting messages, which is a no-no for a wide-area network. Besides, broadcast messages don't usually pass through routers which connect various networks together. Consequently, Telnet relies on the user to specify either the domain name of the machine on which the server resides, for example,

```
mint.sydney.sterling.com
```

or an IP address in dotted decimal notation:

```
199.0.128.61
```

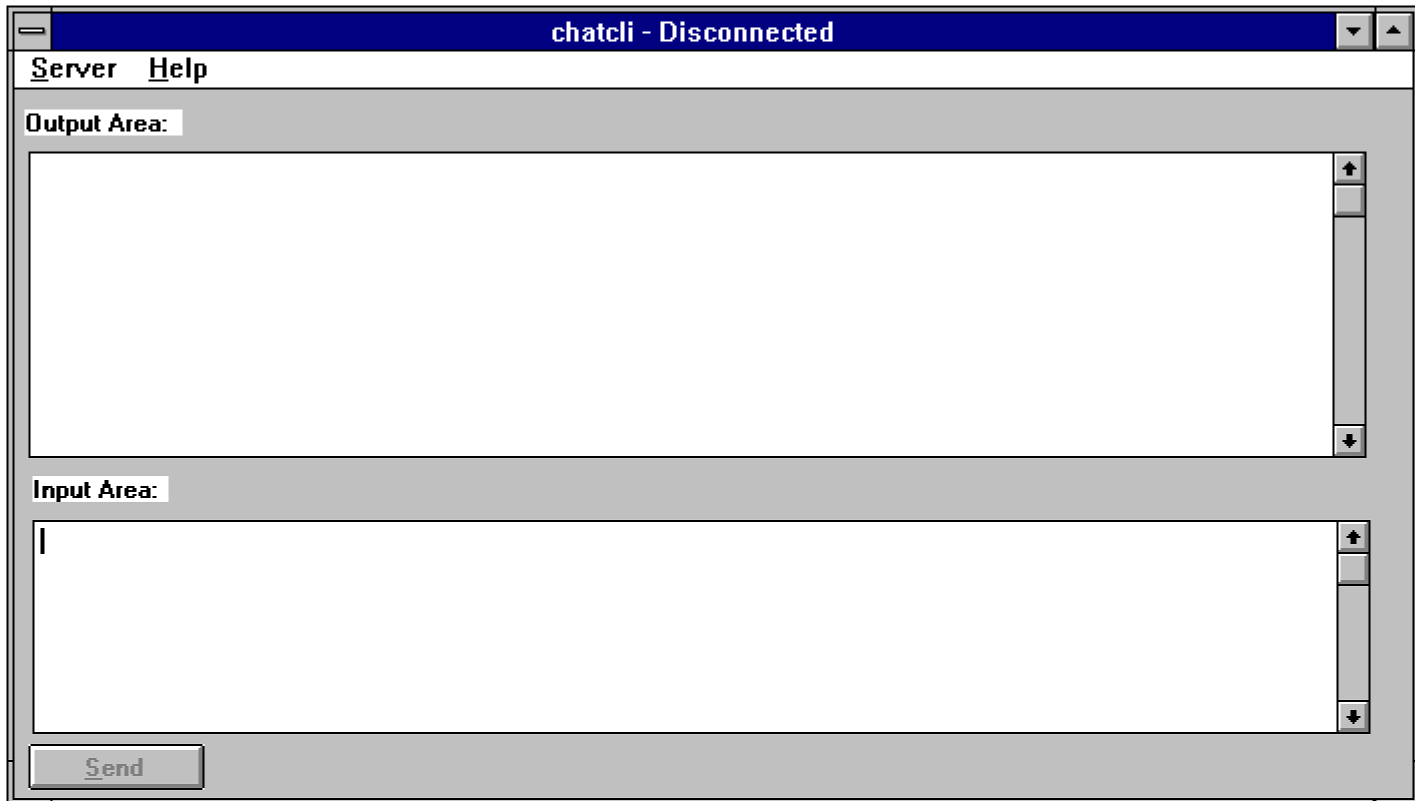
For a discussion of TCP/IP and WinSock, have a look at the following chapter.

For client-server applications running on small to medium size local area networks, finding the servers on the network makes the application more user-friendly and easier to use. Let us explore how we can do this by using an example.

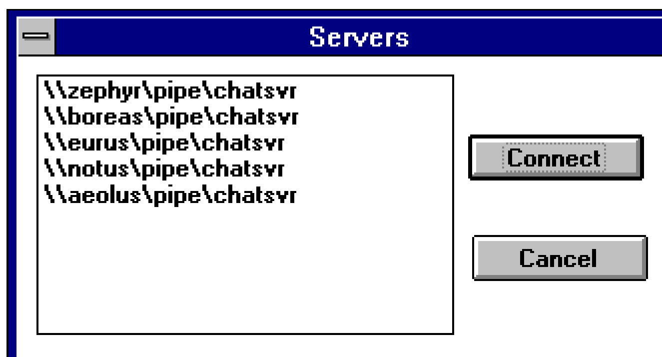
An Mailslot/Named Pipe Example

Our example is a multiparty chat application which consists of the server, `Chatsvr.exe`, and the client,

`Chatcli.exe`. The application allows users on different computers connected via a local area network to communicate via a client and server. It works in much the same way as the Windows NT Chat and IRC (Internet Relay Chat) programs. The server displays the number of client connections in its main window. You can see in the next screenshot that the client main window consists of two edit controls. The top edit control displays the messages received from other users and the bottom one allows the user to compose a message to send. The Send push button, just below the input edit control, is used to send the message. Once it has been sent, the input edit control is emptied to allow the user to enter a new message.



When the user starts `Chatcli.exe`, the Send button is disabled. It's enabled automatically once a connection to the server has been made. The user can view all the chat servers on the network by selecting the Server\Connect... menu. You can see the Connect dialog box below. The user can disconnect from a server by selecting the Server\Disconnect menu.



Making the Connection

In the section on named pipes, we discussed how simple it is for a client to connect to a server, providing that the client knows the name of the named pipe. But how do you find out the name of the pipe? A simple way is to let the user specify the server, but this isn't very user-friendly. The user has to remember a pipe name which is could be long, ugly and difficult to type in correctly. The alternative approach is to let the application do the work itself and provide a list of servers for the user to choose. The good thing about this is that it means you don't have to maintain a server location file. When a new server is added, the application automatically detects it and adds it to its list.

How do we do it? Simple; use mailslots and the directed broadcast capability that they bring. Let's say the pipe's name is,

```
\\.\pipe\abc
```

the server mailslot is,

```
\\.\mailslot\zephyr\abcserv
```

and each client creates a mailslot called,

```
\\.\mailslot\zephyr\abcclnt
```

(You'll see later that the name for a client is actually unimportant.)

When the client application starts, it broadcasts to all servers' mailslots a message that contains its mailslot name, so it broadcasts the message to:

```
\\*\mailslot\zephyr\abcserv
```

When it receives a message, each server sends its pipe name to the client's mailslot. The client receives the response and keeps a list of all the servers' pipe names. When a client wants to connect to a particular server, it just chooses one from list.

The client can repeat this process periodically to keep its server list up-to-date. Alternatively, the client may repeat this procedure whenever the user asks for the server list, from a menu or a toolbar, etc. Once a named pipe connection has been established, the client and server can then exchange whatever information they desire.

The Server

There are a number of different ways for you to organize the server application. You could use a single thread to handle all connections, a separate thread to handle each connection, a fixed number of worker threads, together with Windows NT I/O completion ports to manage multiple connection, or any variation on all of these. To help me illustrate the use of named pipes and mailslots, and because it's easy to understand, I'll use a separate thread to handle each connection.

Chatsvr is a single-document application created using the AppWizard.

In its constructor, **`CChatsvrDoc`** creates the mailslot handling thread **`MailThreadProc()`**, which is responsible for responding to broadcasts sent by chat clients to discover servers on the network. The thread first creates a **`CMailSlotServer`** object for receiving mail and a **`CMailSlotClient`** object for replying to the client whose name is contained in the broadcast message. The reply message is the name of the server named pipe which the client should use to make the server connection.

```
//Mailslot thread
//
UINT MailThreadProc(LPVOID pParam)
{
```

```
    char *name = (char *) pParam;
    char message[BUFFER_SIZE];
    char computer_name[NAME_LEN];
    char pipe_name[NAME_LEN];
    DWORD computer_name_len = sizeof(computer_name);
    CMailSlotServer mail_server;
    CMailSlotClient client;

    //create server mailslot to receive broadcasts from clients
    if (mail_server.Open(name))
    {
       GetComputerName(computer_name, &computer_name_len);

       //listen for broadcasts
       while (mail_server.Read(message, sizeof(message)))
       {
          //use content of the mail as return mailslot name
          if (client.Open(message))
          {
             //give the client the name of our
             //server named pipe
             wsprintf(pipe_name, PIPE_NAME, computer_name);
             client.Write(pipe_name, strlen(pipe_name));
          }
          else
             MessageBeep(0);

          client.Close();
       }
    }
    else
    {
       //probably already had another instance running so quit
       MessageBeep(0);
       AfxGetApp()->m_pMainWnd->SendMessage(WM_CLOSE);
    }

    return 0;
 }
```

The constructor of **CChatsvrDoc** also spawns a thread for listening to an incoming named pipe connection from a client.

```
 //Server Named pipe thread
 //there can be more than one at a time
 //
 UINT PipeThreadProc(LPVOID pParam)
 {
    CNamedPipeServer pipe;
    CNamedPipeServer *other_pipe;
    char alias[NAME_LEN];
    char message[BUFFER_SIZE];
    UINT bytes;

    //make sure we don't create too many threads
    //wait till the others have died
    WaitForSingleObject(hInstanceSem, INFINITE);

    //open the server named pipe
    wsprintf(message, PIPE_NAME, ".");
    if (!pipe.Open(message))
       RaiseException(1, 0, 0, NULL);

    //add pipe object to list while
    //guarding against concurrent list access
    WaitForSingleObject(hListSem, INFINITE);
    pipe_list.AddTail((CObject *) &pipe);

    //update client connection count
    PostMessage(hView, ID_MSG_CONNECT, NULL,
```

```
            (LPARAM) pipe_list.GetCount());

        ReleaseSemaphore(hListSem, 1, NULL);

        //clone itself to handle another instance of the named pipe
        AfxBeginThread(PipeThreadProc, NULL);

        //the first message received is always the name of the client
        if ((bytes = pipe.Read(alias, sizeof(alias))))
        {
           alias[bytes] = 0;
           //wait for more messages
           while ((bytes = pipe.Read(message, sizeof(message))))
           {
              //prevent concurrent access
              WaitForSingleObject(hListSem, INFINITE);

              //write to received message to all pipes
              //preceeded by the
              //name of the sender
                 for (POSITION pos = pipe_list.GetHeadPosition();
                 pos != NULL; )
                 {
                    other_pipe = (CNamedPipeServer *)
                    pipe_list.GetNext(pos);
                    other_pipe->Write(alias, strlen(alias));
                    other_pipe->Write(message, bytes);
                 }

              ReleaseSemaphore(hListSem, 1, NULL);
           }
        }

        //remove pipe object from list
        WaitForSingleObject(hListSem, INFINITE);
        pipe_list.RemoveAt(pipe_list.Find((CObject *) &pipe));

        //update client connection count
        PostMessage(hView, ID_MSG_CONNECT, NULL,
            (LPARAM) pipe_list.GetCount());

        ReleaseSemaphore(hListSem, 1, NULL);

        //enable another clone to run
        ReleaseSemaphore(hInstanceSem, 1, NULL);

        return 0;
    }
```

The permitted number of instances of the named pipe is governed by the initial count of the semaphore, **hInstanceSem**, which is initialized to **MAX_INSTANCE**. Since the first thing that the read thread does is to wait on this semaphore, it blocks as long as there are already **MAX_INSTANCE** running. When it gets past this semaphore, it creates an instance of the pipe by calling **CNamedPipeServer**'s **Open()** member function. When a connection is made, it saves the **CNamedPipeSever** object in an MFC **CObList** object. **pipe_list**, posts an **ID_MSG_CONNECT** message with the number of connections (number of **CNamedPipeServer** objects in the **pipe_list**) in the **lParam** to the **CChatsvrView** object and clones itself so that a new thread can take over its place to wait for another incoming connection.

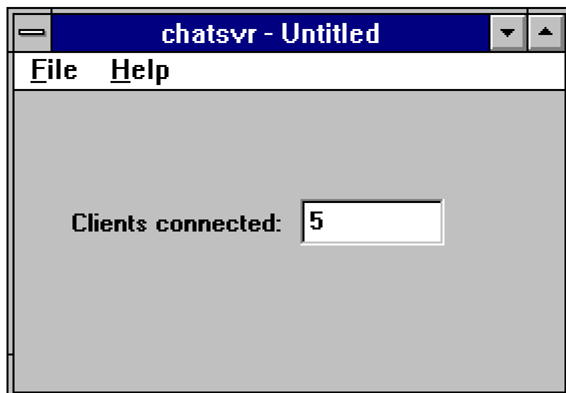
The read thread then does it's stuff. The first message it receives is always going to be the name of the client (you may consider this our chat server's protocol) which it promptly saves. It then continues to read from the pipe. Any message it receives is sent to all **CNamedPipeServer** objects in the **pipe_list**, preceded by the name of the sender that it saved earlier. This means that a message sent by any client will be sent to all connected clients with the sender's name preceding the message.

When a **Read()** fails, it means that the pipe has been broken. The read thread immediately removes its **CNamedPipeServer** object from the **pipe_list** and posts the **ID_MSG_CONNECT** message with the **lParam** set to

the current number of pipe connections to the **CChatsvrView** object. It then releases the semaphore, **hInstanceSem**, so that another waiting read thread may proceed, and terminates. As connections are made and broken, read threads are created and destroyed.

Since more than one read thread can access the **pipe_list**, it is protected from concurrent access by the semaphore, **hListSem**.

Instead of deriving **CChatsvrView** from **CView**, as most applications do, **CChatsvrView** is derived from **CFormView**. **CFormView** allows the main window to attach controls to it in much the same way as a **CDialog** object, i.e. you can use the resource editor to paint the controls. The controls attached include a static text label Clients connected: and a read-only edit field which displays the number of client connections. You can see the view window is below. As I said, **CChatsvrView** is informed of the number of clients currently connected to the server by **ID_MSG_CONNECT** messages.



The Client

The client application is `Chatcli.exe`, which is also generated using the AppWizard. The **CChatcliDoc** class provides communication with the chat server and manages the Server\Connect... and Server\Disconnect menu options.

Like **CChatsvrDoc, CChatcliDoc** creates in its constructor the mailslot-handling thread **MailThreadProc()**, which is responsible for finding chat servers on the network.

```
//Mailslot thread to find all servers
//
UINT MailThreadProc(LPVOID pParam)
{
   char computer_name[NAME_LEN];
   char name[NAME_LEN];
   DWORD computer_name_len = sizeof(computer_name);
   CMailSlotServer mymail(TIMEOUT);   //specify read timeout value
   char message[MAX_MSG_SIZE];
   char *buffer;

   //construct this application's mailslot name
   //use the thread Id to make the mailslot name unique
   //hence multiple instances of the application can be run
   //on the same machine for testing
   sprintf(name, MY_MAILSLOT_NAME, ".", GetCurrentThreadId());

   //create it for receiving incoming mail
   if (mymail.Open(name))
   {
      GetComputerName(computer_name, &computer_name_len);
   }
```

```
    else
    {
       MessageBeep(0);
       RaiseException(1, 0, 0, NULL);
    }

    //construct mailslot object for sending mail
    CMailSlotClient client;

    while (TRUE)
    {
       WaitForSingleObject(hEvent, INFINITE);
       if (fQuit)
          break;

       if (!client.Open(SVR_MAILSLOT_NAME))
       {
          RaiseException(1, 0, 0, NULL);
       }

#ifndef _NO_NETWORK
       sprintf(name, MY_MAILSLOT_NAME, computer_name,
          GetCurrentThreadId());
#endif

       //broadcast our mailslot name
       client.Write(name, strlen(name));
       client.Close();

       //wait for replies until TIMEOUT seconds past without
       //any reply
       int bytesRead;
       while ((bytesRead = mymail.Read(message, MAX_MSG_SIZE)))
       {
          buffer = new char[bytesRead + 1];
          memcpy(buffer, message, bytesRead);
          buffer[bytesRead] = 0;
          if (!PostMessage(hDialog, ID_MSG_SERVER, NULL,
             (LPARAM) buffer))
             delete [] buffer;
          MessageBeep(0);
       }

    }

    return 0;
}
```

The mailslot thread uses the broadcast capability of a mailslot to discover all the chat servers on the network. The thread first creates a **CMailSlotServer** object for receiving mail replies and then a **CMailSlotClient** object for broadcasting to all chat servers. In order for the discovery process to work, all chat servers must have a mailslot with the name:

```
\\computername\mailslot\zephyr\chatsvr
```

(Obviously, **computername** is different for different computers.) The mailslot thread broadcasts its **CMailSlotServer** name to all servers, sending the mail to,

```
\\*\mailslot\zephyr\chatsvr
```

and waits for replies in its server mailslot. When it receives a reply, the mailslot thread copies the reply to a buffer and posts a **ID_MSG_SERVER** message to the **CServer** object, which is responsible for displaying server names, with the **lParam** set to the buffer address. If there is no reply in ten seconds, it assumes that it's found all the servers on the network and goes back to waiting for request to find servers.

CChatcliDoc also handles the Server/Connect... and Server/Disconnect menu selections. When Server/Connect...

is selected, control is passed to **CChatcliDoc::OnFileConnect()** which displays a dialog box implemented using the **CServer** class. If a server has been selected on return from the dialog box (indicated by **IDOK**), **OnFileConnect()** tries to connect to the selected server's named pipe. If the connection is made, it sends the computer and user names through the pipe for user identification, spawns a read thread (**PipeReadThreadProc()**) to read from the pipe, then sends the user Windows message **ID_MSG_CONNECT** with the **lParam** set to **1** to the **CChatcliView** object to inform it of a connection, so that **CChatcliView** can enable the Send button. When the pipe thread receives a message over the pipe, it copies the message to a buffer and posts an **ID_MSG_RECEIVE** message to the **CChatcliView** object with **lParam** set to the buffer address.

```
//Pipe read thread
//
UINT PipeReadThreadProc(LPVOID pParam)
{
   //pipe object passed from CChatcliDoc
   CNamedPipeClient *pipe = (CNamedPipeClient *) pParam;
   char message[MAX_MSG_SIZE];
   char *buffer;

   //wait for input from named pipe
   int bytesRead;
   while ((bytesRead = pipe->Read(message, MAX_MSG_SIZE)))
   {
      //inform CChatcliView that data have been received
      buffer = new char[bytesRead + 1];
      memcpy(buffer, message, bytesRead);
      buffer[bytesRead] = 0;
      if (!PostMessage(hView, ID_MSG_RECEIVE, NULL, (LPARAM) buffer))
         delete [] buffer;
   }

   //named pipe broken: clean up
   PostMessage(hView, ID_MSG_CONNECT, NULL, 0);

   return 0;
}
```

When the user selects Server/Disconnect, **CChatcliDoc:OnFileDisconnect()** disconnects the named pipe and sends the user message **ID_MSG_CONNECT** with the **lParam** set to **0** so that **CChatcliView** can disable the Send button.

CChatcliDoc also manages enabling and disabling of the Server/Connect... and Server/Disconnect menus such that only one of them is enabled at a time, depending on the state of the server connection.

Like the server, **CChatcliView** is derived from **CFormView** to manage three controls in its main window: the multiline input/output edit controls and the Send button.

The Send button is enabled only when there is a connection to a server. When the user clicks on the Send button, **CChatcliView::OnButtonSend()** receives control. First of all, it checks whether there's text in the input edit control. If there's none, it displays an error message. If it finds some text, it retrieves the input edit control's text buffer and invokes the **CChatcliDoc::Send()** method to send the text to the chat server.

As we said earlier, **CChatcliDoc** and its spawned threads use two Windows user messages to inform **CChatcliView** of certain events.

When an **ID_MSG_CONNECT** message is received, **CChatcliView::OnConnect()** uses the **lParam** to determine whether the event is for a pipe connection (1) or a pipe disconnection (0). It enables or disables the Send button accordingly.

When an **ID_MSG_RECEIVE** message is received, **CChatcliView::OnReceive()** uses the **lParam** as a pointer to the received data and appends the data to the output edit control before freeing the buffer.

CServer is a **CDialog**-derived class to handle server selection. The dialog box consists of a list box for displaying the servers it finds, a Connect button and a Cancel button. When the user clicks the Connect button, **CServer** passes the selected server name back to **CChatcliDoc::OnFileConnect()** to make a connection. The user clicks the Cancel button, obviously, to exit from the dialog box without making a selection.

When the **CServer** object is created, an event object handle is passed to its constructor. It sets the event object to signal the mail thread to start looking for servers. Each **ID_MSG_SERVER** message contains a server pipe name which **CServer** puts into its list box. Consequently, the list box is updated as servers are discovered.

NetBIOS

NetBIOS was introduced by IBM with its first LAN in 1984. Since then, it has become a popular programming interface for writing LAN-based client-server applications. For example, the OS/2 Database Manager uses NetBIOS to communicate with its clients. NetBIOS provides both datagram and session support. **Datagrams** are short messages sent, but not necessarily delivered, across a network. In contrast, a **session** is a connection-oriented communication channel between two parties which guarantees that messages are delivered. In fact, you may consider datagram and session services as NetBIOS's equivalents to mailslots and named pipes, respectively.

Programming-wise, NetBIOS is very different from mailslots and named pipes, which are unique among NT's IPCs in that they provide a file-like interface. NetBIOS has only one API:

```
UCHAR Netbios(PNCB pNcb);
```

where **pNcb** is a pointer to the NetBIOS Control Block (**NCB**) whose definition is given below:

```
typedef struct _NCB {

   UCHAR   ncb_command;            /* command code      */
   UCHAR   ncb_retcode;            /* return code         */
   UCHAR   ncb_lsn;                /* local session number   */
   UCHAR   ncb_num;                /* number of our network name   */
   PUCHAR  ncb_buffer;             /* address of message buffer   */
   WORD    ncb_length;             /* size of message buffer   */
   UCHAR   ncb_callname[NCBNAMSZ];/* blank-padded name of remote*/
   UCHAR   ncb_name[NCBNAMSZ];     /* our blank-padded netname    */
   UCHAR   ncb_rto;                /* rcv timeout/retry count   */
   UCHAR   ncb_sto;                /* send timeout/sys timeout    */
   void    (CALLBACK *ncb_post)( struct _NCB * );/* POST routine address*/
   UCHAR   ncb_lana_num;           /* lana (adapter) number   */
   UCHAR   ncb_cmd_cplt;           /* 0xff => command pending   */
   UCHAR   ncb_reserve[10];        /* reserved, used by BIOS   */
   HANDLE  ncb_event;              /* HANDLE to Win32 event which*/
                                   /* will be set to the signaled*/
                                   /* state when an ASYNCH   */
                                   /* command completes      */
} NCB, *PNCB;
```

You must set up an **NCB** before you call WIN32's NetBIOS API.

ncb_command contains the NetBIOS command to be executed. You can execute a NetBIOS command either synchronously or asynchronously. A synchronous operation doesn't return control to the caller until it finishes, while an asynchronous one returns control immediately, even while the command is still executing. In 16-bit Windows, you'll almost never use NetBIOS in the synchronous mode, because the whole system freezes until NetBIOS completes the operation. This is not a problem with Win32 because of its multitasking capability. In fact, it's easier to use NetBIOS in synchronous rather than asynchronous mode. Asynchronous mode is specified by setting the most significant bit of **ncb_command**.

To get a good understanding of programming NetBIOS, you must really get to grips with the concept of names. There are two kinds of names: a group name and a unique name. A NetBIOS name is 16 bytes long and cannot start with a * (which has special meaning for certain commands) or a binary zero. You must register a name with NetBIOS before you start communicating with other computers.

Group names are non-unique, which means that several applications in a NetBIOS network can have the same group name. Group names are useful in connectionless communication where one application wants to talk to multiple applications with the same group name similar to the way we use mailslots in the network chat example. You shouldn't establish a (connection-oriented) session with a group name because there's no way to predict which application it connects to if there are a number of applications with the same group name running.

In contrast, a unique name, by definition, is unique in the network. You will not be able to create a unique name if that name has already been registered in the network. Once you have registered a unique name, other computers on the network can communicate solely with your application by using that name.

NetBIOS Commands

The more common NetBIOS commands are listed below by category.

Name Management Commands

Command	Meaning
NCBADDNAME	Add a unique name.
NCBADDGRNAME	Add a group name.
NCBDELNAME	Delete a name (either group or unique).

Datagram Commands

Command	Meaning
NCBDGRECV	Receive a datagram.
NCBDGRECVBC	Receive a broadcast datagram.
NCBDGSEND	Send a datagram.
NCBDGSENDBC	Send a datagram broadcast.

Session Commands

Command	Meaning
NCBCALL	Instigate a session.
NCBLISTEN	Wait for a session.
NCBRECV	Receive data over a specific session.
NCBRECVANY	Receive data over any session.
NCBSEND	Send data over a specific session.
NCBCHAINSEND	Send chained buffer over a specific session.
NCBSSTAT	Retrieve session status information.
NCBHANGUP	Terminate a session.

General Commands

Command	Meaning
NCBRESET	Clear all names in local name table,
NCBCANCEL	Cancel a previous command,

Using NetBIOS Commands

NetBIOS returns a name number in the **ncb_num** member of the **NCB** structure on successfully adding a name to the local name table using either a **NCBADDNAME** or a **NCBADDGRNAME** command. Once a name, either group or unique, has been registered with NetBIOS, an application can start communicating with another application.

For example, if an application wants to send a datagram , it has to set up a **NCBDGSEND** command with the **ncb_num** , **ncb_callname**, **ncb_buffer** and **ncb_length** fields set to its name number, the NetBIOS name of the application to which it wants to send the datagram, the pointer to the message to send and the length of the message respectively. The receive end sets up a **NCBDGRECV NCB** similarly to receive a datagram addressed to it.

To establish a session, the application instigating the session (client) uses the **NCBCALL** command while the receiver (server) issues a **NCBLISTEN** command. The server can use the special value, ***** ,as the first character in its **ncb_callname** field to specify that it accepts a session request from all names. Once the session has been established, a logical session name number is returned in the **ncb_lsn** field. From then on, all communication between the parties uses this logical session name number to set up commands like **NCBSEND**, **NCBRECV**, **NCBHANGUP**, etc.

Using asynchronous commands in 16-bit Windows requires setting up a post routine in the **ncb_cmd_cplt** field. The asynchronous command returns immediately. The post routine is called when the command finishes. This mechanism is still supported under Win32, but the preferred way to handle asynchronous commands is to use the new **ncb_event** field to specify an event object for NetBIOS to signal on completion. You can then use a **WaitForSingleObject()** call to wait for the event object to be signaled.

> **NCB is the traditional way to program NetBIOS applications. Windows NT actually allows you to program an application in WinSock (using Wsnetbs.h and Wsock32.lib) that uses the NetBEUI transport layer. However, you still need to know the basics of the transport protocol, such as addressing, broadcast capability, etc. to use WinSock with NetBEUI successfully.**

A Wrapper for NetBIOS

I want to provide the same C++ interface to NetBIOS as I have for named pipes and mailslots, so that applications written using these C++ network communication classes can easily switch from using mailslots and named pipes to NetBIOS. Another reason is to hide the complexity of the NetBIOS interface from the user. The user only programs using the four file-like class member functions: **Open()**, **Read()**, **Write()** and **Close()**. Consequently, the public member functions of the NetBIOS classes are exactly the same as those for mailslots and named pipes. However, there are new protected functions for carrying out certain common NetBIOS services.

CNBiosCommBase

CNBiosCommBase is the NetBIOS equivalent of **CCommBase** class for mailslots and named pipes. Like **CCommBase**, **CNBiosCommBase** is an abstract class which doesn't implement the public member function : **Open()**. Its class definition is:

```
//communication base class
class CNBiosCommBase
{
protected:
   static BOOL m_Initialized;
   UCHAR m_Num;
   UCHAR m_Lsn;
   UCHAR m_RecvCmd;
   UCHAR m_SendCmd;
   BOOL m_InSession;
   CString m_MyName;
```

```
      CString m_PartnerName;
      DWORD m_ErrorCode;
      DWORD m_Timeout;
      HANDLE m_TermEvent;
      CNcb *m_Recvncb;
      CNcb *m_Sendncb;

      UCHAR CreateGroupName(CString& name);
      UCHAR CreateUniqueName(CString& name);
      void NormalizeName(CString& target, CString source);
      void DeleteName(CString& name);
      void CancelCmd(CNcb& cancelncb);
      void Initialize();
      DWORD Wait(HANDLE hEvent, DWORD timeout);

   public:
      CNBiosCommBase();
      CNBiosCommBase(UCHAR sendCmd, UCHAR recvCmd,
         DWORD timeout = INFINITE, HANDLE hTermEvent = NULL);
      ~CNBiosCommBase();

      inline DWORD GetError() { return m_ErrorCode; };
      inline BOOL IsConnected() { return m_InSession; };

      virtual BOOL Open(const char* pszFileName,
         UINT nOpenFlags = 0) = 0;
      virtual UINT Read(void* pBuf, UINT nCount);
      virtual UINT Write(const void* pBuf, UINT nCount);
      virtual void Close();

   };
```

It implements the **Read()**, **Write()** and **Close()** functions and a number of basic NetBIOS services, such as adding/deleting group and unique names, limiting/padding a name to 16 bytes for use as a NetBIOS name, etc. All these basic NetBIOS helper functions are executed synchronously. The static data member **m_Initialized** is used in its constructor to detect the first instance of **CNBiosCommBase**'s derived class object such that the NetBIOS **NCBRESET** command is executed once and only once for a process in **CNBiosCommBase**'s constructor by calling the member function, **Initialize()**.

```
   //constructor
   CNBiosCommBase::CNBiosCommBase()
   {
      m_Timeout = INFINITE;
      m_Num = 0;
      m_Lsn = 0;
      m_RecvCmd = NCBRECV;
      m_SendCmd = NCBSEND;
      m_InSession = FALSE;
      m_ErrorCode = 0;
      m_TermEvent = NULL;
      m_Recvncb = new CNcb(CREATE_EVENT_OBJECT);
      m_Sendncb = new CNcb;
      Initialize();

   }

   //constructor with command and timeout specifications
   CNBiosCommBase::CNBiosCommBase(UCHAR sendCmd, UCHAR recvCmd,
      DWORD timeout, HANDLE hTermEvent)
   {
      m_Timeout = timeout;
      m_Num = 0;
      m_Lsn = 0;
      m_RecvCmd = recvCmd;
      m_SendCmd = sendCmd;
      m_InSession = FALSE;
      m_ErrorCode = 0;
      m_TermEvent = hTermEvent;
```

```
   m_Recvncb = new CNcb(CREATE_EVENT_OBJECT);
   m_Sendncb = new CNcb;
   Initialize();

}

void CNBiosCommBase::Initialize()
{
   //initialize class data members
   m_InSession = FALSE;
   m_Num = m_Lsn = 0;
   m_ErrorCode = 0;
   if (!m_Initialized)
   {
      CNcb ncb;

      ncb.m_ncb.ncb_command = NCBRESET;
      m_ErrorCode = Netbios(&ncb.m_ncb);

      m_Initialized = TRUE;
   }
}
```

The simple convenience class **CNcb** is used to initialize an **NCB** structure and provide an event object for synchronizing asynchronous NetBIOS commands. Using an event object for synchronization is better than using post routines, which use significantly more system resources. **CNcb**'s class definition and its implementation is:

```
#define LANA               0

//helper class for managing NetBios NCBs
class CNcb
{
public:
   BOOL m_CreateEvent;
   NCB   m_ncb;

   CNcb(BOOL fCreateEvent = NO_EVENT_OBJECT, UCHAR lana = LANA);
   ~CNcb();
};

//Initialze NCB and create event object if required
CNcb::CNcb(BOOL fCreateEvent, UCHAR lana)
{
   memset(&m_ncb, 0, sizeof(NCB));
   m_ncb.ncb_lana_num = lana;
   m_CreateEvent = fCreateEvent;
   if (m_CreateEvent)
      m_ncb.ncb_event = CreateEvent(NULL, TRUE, FALSE, NULL);
};

//get rid of the NCB and event object
CNcb::~CNcb()
{
   if (m_CreateEvent)
      CloseHandle(m_ncb.ncb_event);
};
```

We need the data member **m_CreateEvent** to determine in the destructor whether to delete the event object and because, when it's executing a NetBIOS command synchronously, the system uses an event object in the **ncb_event** field for synchronization. Although the system closes the event object handle before returning from the NetBIOS call, the **ncb_field** still contains some value. Also, all data member are public to facilitate access.

It should be noted that `CNcb` initializes `ncb_lana_num` to `0`. You may need to change LANA to 1 in order for these NetBIOS classes to work on your workstation if you have configured your workstation to use your second network adapter.

Only the asynchronous NetBIOS commands are used to implement the member functions, **Read()** and **Write()**. Let's look at the implementation of **Read()**:

```
//read a datagram or a message from a session
UINT CNBiosCommBase::Read(void* pBuf, UINT nCount)
{
   if (m_Num == 0)
      return 0;

   //receive a message
   m_Recvncb->m_ncb.ncb_command = m_RecvCmd | ASYNCH;
   m_Recvncb->m_ncb.ncb_num = m_Num;
   m_Recvncb->m_ncb.ncb_lsn = m_Lsn;
   m_Recvncb->m_ncb.ncb_buffer = (PUCHAR) pBuf;
   m_Recvncb->m_ncb.ncb_length = nCount;
   if (!Netbios(&m_Recvncb->m_ncb) &&
      (m_Recvncb->m_ncb.ncb_retcode == NRC_PENDING))
      Wait(m_Recvncb->m_ncb.ncb_event, m_Timeout);

   m_ErrorCode = m_Recvncb->m_ncb.ncb_retcode;
   if (m_Recvncb->m_ncb.ncb_retcode == NRC_PENDING)
   {
      //cancel previous command on timeout
      CancelCmd(*m_Recvncb);
      return 0;
   }
   else if (m_Recvncb->m_ncb.ncb_retcode)
      return 0;

   return m_Recvncb->m_ncb.ncb_length;
};
```

The first thing that **Read()** does is to check whether a name has been registered with NetBIOS. If a name has been registered, **m_Num** will contain a non-zero number. (We'll see later that a **CNBiosComm**-derived class object is responsible for registering names.) It then uses the **CNcb** object **m_Recvncb** created in its constructor to issue either **NCBDGRECV** or **NCBRECV** command, which is specified in the class data member **m_RecvCmd**. Unlike synchronous NetBIOS commands, which return the error code when the command is completed, an asynchronous command returns zero (no error) while still executing the command. **Read()** checks the **NCB** field **ncb_retcode** to decide what to do next. If the value is **NRC_PENDING**, **Read()** waits on the event object until it gets signaled on command completion or times out. **Read ()** saves the **ncb_retcode** in the data member **m_ErrorCode** for retrieval, using the **GetError()** member function. If the command is still incomplete on timeout, **Read()** cancels the operation by calling the protected member function **CancelCmd()**, which issues a **NCBCANCEL** command.

The **Write()** member function is implemented in a similar fashion, using the **NCBDGSEND** or **NCBSEND** command:

```
//send a datagram or a message over a session
UINT CNBiosCommBase::Write(const void* pBuf, UINT nCount)
{
   if (m_Num == 0)
      return 0;

   //send a message
   m_Sendncb->m_ncb.ncb_command = m_SendCmd;
   memcpy(m_Sendncb->m_ncb.ncb_callname,
      (const char *) m_PartnerName, NCBNAMSZ);
   m_Sendncb->m_ncb.ncb_num = m_Num;
   m_Sendncb->m_ncb.ncb_lsn = m_Lsn;
   m_Sendncb->m_ncb.ncb_buffer = (PUCHAR) pBuf;
   m_Sendncb->m_ncb.ncb_length = nCount;
   if (!Netbios(&m_Sendncb->m_ncb) &&
      (m_Sendncb->m_ncb.ncb_retcode == NRC_PENDING))
      Wait(m_Sendncb->m_ncb.ncb_event, m_Timeout);
```

```
    m_ErrorCode = m_Sendncb->m_ncb.ncb_retcode;
    if (m_Sendncb->m_ncb.ncb_retcode == NRC_PENDING)
    {
       //cancel previous command on timeout
       CancelCmd(*m_Sendncb);
       return 0;
    }
    else if (m_Sendncb->m_ncb.ncb_retcode)
       return 0;

    return nCount;
};
```

CNBiosSessionServer

CNBiosSessionServer is derived from **CNBiosCommBase**. Its class definition is given below:

```
class CNBiosSessionServer: public CNBiosCommBase
{
protected:
   static int m_Instances;
   CNcb *m_Openncb;

public:
   CNBiosSessionServer();
   CNBiosSessionServer(DWORD timeout, HANDLE hTermEvent = NULL);
   ~CNBiosSessionServer();
   virtual BOOL Open(const char* pszFileName,
      UINT nOpenFlags = 0);
};
```

It implements a virtual destructor and the virtual member function, **Open()**.

Like all **CNBiosCommBase**-derived NetBIOS classes, its constructor sets up the data members **m_RecvCmd** and **m_SendCmd**, so that **Read()** and **Write()** know which command to issue.

Before it carries on, **Open()** verifies that a NetBIOS name (local or remote) has been specified. It uses the NetBIOS **NCBADDNAME** command to create a unique name for the server. If it fails, it could mean that the name has already been created earlier. This may not be a problem because an application can use the same name to establish multiple sessions with different applications. In order to verify that, **Open()** executes a **NCBSSTAT** command to find out more about the sessions associated with the name and retrieves the name's **ncb_num** for use in a subsequent **NCBLISTEN** command to wait for an incoming session. An asynchronous **NCBLISTEN** is used so that the caller has the option to wait indefinitely, or wait till a timeout occurs.

```
//Open a NetBios session
BOOL CNBiosSessionServer::Open(const char* pszFileName, UINT nOpenFlags)
{
   //check if a name has been specified
   if (!pszFileName)
      return FALSE;

   //create NetBios name for server
   NormalizeName(m_MyName, pszFileName);
   if ((m_Num = CreateUniqueName(m_MyName)) == 0)
   {
      //name has been created
      CNcb ncb;
      SESSION_HEADER header;

      //retrieve NetBios name number
      ncb.m_ncb.ncb_command = NCBSSTAT;
      ncb.m_ncb.ncb_buffer = (PUCHAR) &header;
      ncb.m_ncb.ncb_length = sizeof(SESSION_HEADER);
      memcpy(ncb.m_ncb.ncb_name,
         (const char *) m_MyName, NCBNAMSZ);
```

```
      if ((m_ErrorCode = Netbios(&ncb.m_ncb)) == NRC_INCOMP)
         m_Num = header.sess_name;
      else
         return FALSE;
   }

   m_Instances++;
   //wait for incoming session
   m_Openncb->m_ncb.ncb_command = NCBLISTEN | ASYNCH;
   NormalizeName(m_PartnerName, "*");
   memcpy(m_Openncb->m_ncb.ncb_callname,
      (const char *) m_PartnerName, NCBNAMSZ);
   memcpy(m_Openncb->m_ncb.ncb_name,
      (const char *) m_MyName, NCBNAMSZ);
   if (!Netbios(&m_Openncb->m_ncb) &&
      (m_Openncb->m_ncb.ncb_retcode == NRC_PENDING))
      Wait(m_Openncb->m_ncb.ncb_event, m_Timeout);

   m_ErrorCode = m_Openncb->m_ncb.ncb_retcode;
   if (m_Openncb->m_ncb.ncb_retcode == NRC_PENDING)
   {
      //cancel previous command on timeout
      CancelCmd(*m_Openncb);
      return FALSE;
   }
   else if (m_Openncb->m_ncb.ncb_retcode)
      return FALSE;

   m_InSession = TRUE;
   m_Lsn = m_Openncb->m_ncb.ncb_lsn;
   return TRUE;

}
```

The only other thing that we need to cover is **CNBiosSessionServer**'s destructor. Since multiple sessions can be established using one name, the destructor checks that there are no more active or pending active sessions associated with the name before deleting it from the local NetBIOS name table The static data member **m_Instances** is used to keep track of the object instances.

CNBiosSessionClient

CNBiosSessionClient, once again, is derived from **CNBiosCommBase**.

```
class CNBiosSessionClient: public CNBiosCommBase
{
protected:
   CNcb *m_Openncb;

public:
   CNBiosSessionClient();
   CNBiosSessionClient(DWORD timeout, HANDLE hTermEvent = NULL);
   ~CNBiosSessionClient();
   virtual BOOL Open(const char* pszFileName,
      UINT nOpenFlags = 0);
};
```

Unlike **CNBiosSessionServer**, the constructor creates a unique name based on the current thread identifier. This is used for communicating with another application whose NetBIOS name is specified in **Open()**.

Open() uses the name specified as its remote partner's name for establishing a NetBIOS session.

```
//Open a NetBios session to a server
BOOL CNBiosSessionClient::Open(const char* pszFileName, UINT nOpenFlags)
{
   //check if a name has been specified
   if (!pszFileName)
```

```
        return FALSE;

    //establish session to remote partner name
    NormalizeName(m_PartnerName, pszFileName);
    m_Openncb->m_ncb.ncb_command = NCBCALL | ASYNCH;
    memcpy(m_Openncb->m_ncb.ncb_callname,
        (const char *) m_PartnerName, NCBNAMSZ);
    memcpy(m_Openncb->m_ncb.ncb_name,
        (const char *) m_MyName, NCBNAMSZ);
    if (!Netbios(&m_Openncb->m_ncb) &&
        (m_Openncb->m_ncb.ncb_retcode == NRC_PENDING))
        Wait(m_Openncb->m_ncb.ncb_event, m_Timeout);

    m_ErrorCode = m_Openncb->m_ncb.ncb_retcode;
    if (m_Openncb->m_ncb.ncb_retcode == NRC_PENDING)
    {
        //cancel previous command on timeout
        CancelCmd(*m_Openncb);
        return FALSE;
    }
    else if (m_Openncb->m_ncb.ncb_retcode)
        return FALSE;

    m_InSession = TRUE;
    m_Lsn = m_Openncb->m_ncb.ncb_lsn;
    return TRUE;

};
```

Since each instance of a **CNBiosSessionClient** object only establishes a single session with a remote application, it simply deletes its name from the NetBIOS name table in its destructor by calling the protected member function: **DeleteName()**.

CNBiosDatagramServer

CNBiosDatagramServer, unlike **CNBiosSessionServer**, uses the connectionless datagram services for communication which makes implementation relatively simple, compared to **CNBiosSessionServer**. Its class definition is given below:

```
class CNBiosDatagramServer: public CNBiosCommBase
{
public:
    CNBiosDatagramServer();
    CNBiosDatagramServer(DWORD timeout, HANDLE hTermEvent = NULL);
    ~CNBiosDatagramServer();
    virtual BOOL Open(const char* pszFileName,
        UINT nOpenFlags = 0);
};
```

Open() simply creates a NetBIOS group name for the name specified and records the **ncb_num** in **m_Num** for use by **Read()** and **Write()** (implemented in **CNBiosCommBase**).

```
//Establish NetBios name for server
BOOL CNBiosDatagramServer::Open(const char* pszFileName, UINT nOpenFlags)
{
    //check if a name has been specified
    if (!pszFileName)
        return FALSE;

    //create server mailslot
    NormalizeName(m_MyName, pszFileName);
    if ((m_Num = CreateGroupName(m_MyName)) == 0)
        return FALSE;

    return TRUE;
}
```

CNBiosDatagramClient

CNBiosDatagramClient also creates a unique name behind the scenes in its constructor, like **CNBiosSessionClient**, for communicating with the server whose name is specified in **Open()**. Its class definition is given below:

```
class CNBiosDatagramClient: public CNBiosCommBase
{
public:
   CNBiosDatagramClient();
   CNBiosDatagramClient(DWORD timeout, HANDLE hTermEvent = NULL);
   ~CNBiosDatagramClient();
   virtual BOOL Open(const char* pszFileName,
      UINT nOpenFlags = 0);
};
```

The **Open()** implementation is even simpler than that of **CNBiosDatagramServer**'s. All it does is to save the server's name in the **m_PartnerName** data member for use in subsequent **Read()**s and **Write()**s.

```
//establish remote partner name
BOOL CNBiosDatagramClient::Open(const char* pszFileName, UINT nOpenFlags)
{
   //check if a name has been specified
   if (!pszFileName)
      return FALSE;

   NormalizeName(m_PartnerName, pszFileName);

   return TRUE;
};
```

A NetBIOS Example

The network chat example also illustrates how NetBIOS classes are used, at the same time, demonstrating how easy it is to replace the transport layer of an client-server application by simply switching from using the mailslot and named pipe classes to using the NetBIOS classes. Aside from the class name changes in the program sources (which are unavoidable), the other changes are made in `Cchatsvrdoc.cpp` and `Cchatclidoc.cpp`, concerning the names used for pipes and mailslots. These changes are shown below.

CChatSvrDoc

From:

```
MAILSLOT_NAME   \\computername\mailslot\zephyr\chatsvr
PIPE_NAME       \\computername\pipe\chatsvr
```

to:

```
MAILSLOT_NAME   \\chatsvr
PIPE_NAME       \\computername\chatsvr
```

CChatcliDoc

From:

```
SVR_MAILSLOT_NAME   \\computername\mailslot\zephyr\chatsvr
MY_MAILSLOT_NAME    \\computername\mailslot\zephyr\chatcli\nnn
```

to:

```
SVR_MAILSLOT_NAME    \\chatsvr
MY_MAILSLOT_NAME     \\computername\nnn
```

where **nnn** is a unique number (the thread identifier is used for this purpose).

Note that NetBIOS names do not have to start with ****. I've just use the named pipe convention, where the first part of a remote pipe's name is **\\computername**, for NetBIOS names for connection-oriented sessions to minimize the changes required of the chat application. As you can see, I still use names like **MAILSLOT_NAME** and **PIPE_NAME** although they're really NetBIOS names.

The **chatsvr** and **chatcli** programs work in the same way as their counter-parts that used mailslots and named pipes. There are no appreciable differences in the appearance of the application to indicate whether it is using mailslots/named pipes or NetBIOS transport.

Choosing the IPC to Use

Which IPC you choose to use in an application depends on the needs of the application and the availability of the IPC on the platforms that the application intends to run on. For example, a named pipe can only be created on Windows NT, which precludes running the server application on Windows 95 or Windows 3.1. You should also take a good look at what the application needs to do. If an application needs to send small volumes of information from one end to many other ends (one-to-many relationship), you should use a connectionless IPC with broadcast or multicast capability, such as mailslots and NetBIOS datagrams. On the other hand, if an application delivers large volumes of information from one end to another end (one-to-one relationship), you should consider a connection-oriented IPC, such as named pipe or NetBIOS session. Although I only mention NetBIOS, mailslots and named pipes here because this chapter is on programming using these three IPCs, there's no reason why you should limit yourself to using them alone. Use whatever's suitable and available on the platform.

So, when you're choosing which IPC to use, you should ask yourself these questions:

Is the IPC available on all my target platforms?
Should I use connectionless or connection-oriented IPCs?
How easy or difficult is it to use the IPC?
What is its performance like?
How much system resources does the IPC consume?

Summary

This chapter has surveyed the various IPCs available on Windows NT for writing distributed (client-server) applications and discussed in detail how to use three IPCs: NetBIOS, mailslots and named pipes. We have taken a look at a simple but powerful approach to using NetBIOS, mailslots and named pipes in writing client-server applications, which utilizes NetBIOS and mailslots' broadcast capability for discovering services on a network and uses either a NetBIOS session or a named pipe for the actual exchange of information between the client and server. A number of simple C++ classes are developed to encapsulate these IPCs for use in writing client-server applications. They all use the file metaphor in which the major class member functions are the well-known **Open()**, **Read()**, **Write()** and **Close()** operations. A simple multi-party chat application is developed to illustrate the use of these classes in organizing a typical client-server application.

WinSock Internet Programming

Of all the emerging technologies of the digital age, nothing is more hyped than the so-called information highway. To most, the information highway is the Internet, a collection of computers connected together using standard protocols to exchange information. Many others see the Internet as just a first step in the construction of the superhighway. Whatever your view, there's no doubt that the Internet is quickly evolving and is changing the way we work and communicate with each other.

The explosive growth of the Internet in the past couple of years has spawned many new corporations, alliances and research programs. Announcements are made almost daily by software companies committing to build Internet connectivity into existing and future hardware and software products. New software programs are being developed to take advantage of the expansive connectivity made possible by the Internet.

In this chapter, you'll learn how to make your own Visual C++ applications Internet-capable by using the Windows Sockets Application Programming Interface (WinSock API). Specifically, we'll cover:

- The Socket programming model
- WinSock asynchronous extensions
- MFC Classes: **`CAsyncSocket`** and **`CSocket`**
- Building a Finger client application using **`CAsyncSocket`**
- Building a Home Automation client and server application using **`CSocket`**, **`CSocketFile`** and **`CArchive`**

A Brief History of WinSock

It's safe to say that the network protocol TCP/IP is largely responsible for the quick growth of the Internet. It's built on an open, layered architecture which makes it compatible with almost any lower-layer network infrastructure. Independence from the lower layers, the physical and data link layers, means that TCP/IP can run on any network medium, such as Ethernet, Token Ring, Serial Line, FDDI, etc. This flexibility and the open spirit in which it was developed has propelled TCP/IP into its current ubiquity.

TCP/IP quickly became the architecture of choice for developing network applications. The push began in UNIX environments, but it soon found its way into the DOS-based personal computer market. Inexpensive LANs, fast C and C++ compilers and the acceptance of Windows by the business community all paved the way for the quick entrance of TCP/IP into the PC world. Shortly after, many more PC software developers took interest in providing network services with their applications and different TCP/IP APIs began to surface. WinSock was one of them.

The **Windows Sockets API** (more commonly known as **WinSock**, or occasionally just **WSA**) is an extension of the original Sockets API. Sockets is an application programming interface that was developed in the early days of the Internet by the University of California at Berkeley for its version of UNIX. The API makes it easier to develop network applications because it shields the developer from the complexities of the underlying network. The API adds a layer of abstraction that allows developers to concentrate on the problem domain (the application or business logic) and not so much on the network peculiarities. It does so by generalizing the communication mechanism through the use of handles, in much the same way that file handles make it easier to work with disk files.

WinSock is an evolving, living document that describes network programming under Microsoft Windows. That is, WinSock is really just a specification that describes a network programming model. The specification is not owned by anyone. In fact, it was drafted through the cooperative efforts of a number of software vendors, all interested in having a standard for writing network applications under the Windows family of operating systems. Any vendor is free to implement the API specification, and many have. Each network vendor may provide its own version of the `Winsock.dll` (or `Wsock32.dll`) to implement the Windows Sockets API for that vendor's particular protocol stack. Microsoft began shipping its own version of the `Winsock.dll` as an add-on to Windows for Workgroups and is now distributing it as a standard component of Windows 95 and Windows NT.

The success of the WinSock API is due largely to the spirit in which it was born. The specification was drafted through the cooperative efforts of a number of software vendors, all interested in having a standard for writing network applications under the Windows family of operating systems. The WinSock Group, which began in 1991 as an informal gathering at Interop (a TCP/IP networking trade show), was organized to define a way for Windows-based applications to interact with TCP/IP. The current WinSock standard, Version 1.1, was released in January of 1993 and is largely built around TCP/IP, but certainly not restricted to it. In fact (although a great many implementations use TCP/IP), the specification is protocol-independent.

The informal gathering at Interop that led to the birth of the WinSock Standards Group was led by Martin Hall of Stardust Technologies. The WinSock Group continues to refine the specification and is currently completing Version 2, which includes extensions for additional network transports and new media.

For more information about the WinSock Group and the new specification, visit the Stardust web site at `http://www.stardust.com`.

WinSock Concepts

A **socket** is a software handle, similar to a file handle, that can be used to both send and receive data over a network. Like a file handle, a socket identifies a data structure which provides the information needed to orchestrate read and write operations. However, a socket represents an abstraction a bit more complex than a disk file. In other words, using a socket is not quite as easy and straightforward as using a file handle. Fortunately, MFC provides a set of classes (notably `CAsyncSocket` and `CSocket`) that reduce the complexities associated with using the bare-bones WinSock API. The MFC classes wrap socket handles, data structures and WinSock API functions into C++ objects. Thus, network programming can be approached from an object-oriented perspective, instantiating objects and calling methods. The exact sequence of steps (methods to be called) required to effectively use a socket depends on the type and role of the socket.

Types of Sockets

There are two types of sockets from which to choose: **datagram** and **stream**.

Datagram sockets allow data to be transferred over a network with a minimum of overhead. However, nothing is free. The reason for the low overhead is because datagram sockets use UDP as the network transport protocol. UDP guarantees that when the data is delivered it will be correct, but it doesn't guarantee delivery in all cases. The data may be lost, discarded, or arrive out of sequence. Datagram sockets are most often used when a small amount of data needs to be exchanged and the overhead of establishing a reliable stream connection is not warranted. For example, a datagram socket is perfect for simple systems that send status updates to other applications, without too much concern over whether or not they are heard.

Stream sockets are used to establish reliable, two-way connections to transfer streams of bytes. Again, nothing is free. The reliability is attained by relying on TCP as the network transport protocol. TCP ensures that data is delivered error-free by performing additional, sometimes time-consuming, handshaking with the receiver. Stream sockets are well suited for applications that require the error-free exchange of large amounts of data. Most Internet protocols, including the **HyperText Transfer Protocol** (HTTP), **File Transfer Protocol** (FTP) and many others rely on stream sockets.

The socket type is defined when the socket is created and cannot be changed later. The MFC C++ socket classes (discussed in more depth later this chapter) implement stream sockets by default, as shown by the second parameter of the `CAsyncSocket::Create()` member function listed below:

```
BOOL Create(UINT nSocketPort = 0, int nSocketType=SOCK_STREAM,
   long lEvent = FD_READ | FD_WRITE | FD_OOB | FD_ACCEPT |
   FD_CONNECT | FD_CLOSE, LPCTSTR lpszSocketAddress = NULL);
```

We'll take a closer look at all of the parameters later. First, let's look at how sockets are used and the roles and responsibilities of the programs wishing to exchange information across a network.

The Role of a Socket

It takes two to tango. Clients lead and servers follow. Servers arrive at the dance first and wait quietly along the wall, waiting for clients to call. Clients arrive later looking for a specific server and they always request a specific dance.

Socket programming is always approached from the perspective of clients and servers. In general, client sockets initiate connections and server sockets wait for and accept them. (This discussion assumes that we're using stream sockets, the most commonly used type of socket for Internet programming.) However, after the initial contact, either the client or the server is capable of sending the first packet of data. The actual 'dance steps' depend on the application or protocol. The Internet has some clearly defined protocols that establish the roles and responsibilities of the client and server sockets. Later in this chapter, we'll take a close look at the **Finger** protocol with a simple Internet application that is used to look for information about a user on a specific host.

When a socket is created, a program does *not* expressly indicate the role (client or server) of the socket, or how it will be used. Instead, the role is defined by the particular set and sequence of socket functions called. For example, to use a client socket, you simply create the socket, call the connect function to attempt to reach a server and establish a communication end-point, send and receive data, then close the socket. Server sockets, on the other hand, are used a bit differently. First, you create a socket (just like you create a client socket), bind it to a specific service or **port** and set it to quietly listen for clients trying to connect. Then, when a connection attempt is detected, you can accept the connection and create a new socket to delegate the responsibility of servicing the client. The newly created socket will perform the data exchange with the client, allowing the server socket to continue listening for connection requests from other clients.

IP Addresses and Port Numbers

So how does a client know how to reach a server? Well, the client must know two things: the **IP address** of the server's host computer and the specific service **port number**.

The IP address names the computer on which the server process is running, thus making it possible for a client to call on a specific 'dance partner'. Every computer on the Internet must have a unique address that distinguishes it from every other. Internet addresses are 32-bit numbers that are usually represented in a **dotted decimal notation**, where the 32-bit value is broken up into four 8-bit decimal chunks, separated by dots. This notation makes it easier for us carbon-based life forms to read and remember IP addresses. For example, the IP address 0xFF010A01 can be written as 255.1.10.1.

The port number identifies the specific service running on the host computer named by the IP address. Continuing with the analogy, the port identifies the 'dance step'. The Internet has some well-known dances that are assigned specific port numbers. So, for a client wishing to dance the File Transfer Protocol (FTP), it first identifies the dance partner (the host computer with a specific IP address), then the FTP port, 21. If the client fancies HyperText Transfer Protocol (HTTP) instead, it should try port 80. As an example, the following table lists a few of the well-known port numbers:

Port Number	**Protocol**
21	FTP (File Transfer Protocol)
23	Telnet
25	SMTP (Simple Mail Transport Protocol)
79	Finger
80	HTTP (HyperText Transfer Protocol)

You can find defines for many of the well known port numbers in `Winsock.h` as **`IPPORT_`***`servicename`*, so, for example, you can use **`IPPORT_TELNET`** and **`IPPORT_FINGER`** instead of **`23`** and **`79`**.

If you're developing a custom application, the port number you use should be in the range of 1,025 – 5,000. These values are set aside by the Internet Assigned Numbers Authority for user-defined services. Port numbers from 0 – 1,024 are reserved for well-known services such as HTTP.

Network Byte Order

We'll take a look at one final concept before we move on to examine the use of sockets and the MFC classes. When you're developing applications that are going to talk to the outside world, you need to keep in mind that it's a jungle out there. There are many different types of hardware platforms, each one a potential host or client to which we may want to connect and exchange data. And guess what—each has its own way of storing and manipulating multibyte data elements, such as integers and longs. This is a problem.

The most notable example is with the Intel and Motorola processors. Intel uses a **little-endian** byte ordering scheme which means that the least significant byte of a multibyte value is stored in memory before the most significant byte. Motorola uses a **big-endian** byte ordering scheme which means that the most significant byte of a multibyte value is stored in memory before the least significant byte. Seems crazy, but that's the way it is.

The bottom line is that you simply can't send multibyte data elements to another machine without some sort of translation, unless you're absolutely, without a doubt, 100% certain that the receiver machine uses the same byte ordering scheme. Of course, to ensure the portability of your code, you should never assume anything and take the extra effort to translate your data to network order before you send it, and back to host order once you've received it. To accomplish this is a simple matter of invoking what I call the 'Esperanto functions'.

The WinSock API provides four functions for converting numbers to and from network byte order. They are:

Function	Description
`htons()`	Takes a 16-bit (short) number in host byte order and returns a 16-bit number in network byte order.
`ntohs()`	Takes a 16-bit (short) number in network byte order and returns a 16-bit number in host byte order.
`htonl()`	Takes a 32-bit (long) number in host byte order and returns a 32-bit number in network byte order.
`ntohl()`	Takes a 32-bit (long) number in network byte order and returns a 32-bit number in host byte order.

You would need to use these functions when you're exchanging data which is to be interpreted by the network or by another machine. For example, the WinSock API function **`::connect()`** expects that the port number you pass to it is already in network byte order. However, if you're using the MFC implementation of this function on one of the socket classes, for example **`CAsyncSocket::Connect()`**, (and you should be), the port number will be in host byte order because the MFC function takes care of calling **`htons()`** to convert it for you. The line of code that does this (you'll find it in `Sockcore.cpp` in your MFC source directory) appears as:

```
sockAddr.sin_port = htons((u_short)nHostPort);
```

Once a socket is connected, none of this really applies if the application is simply exchanging ASCII characters as a stream of bytes. It really only applies to applications that are exchanging multibyte data elements, usually numeric, that need to be interpreted by both sides.

Let's Dance

We said earlier that, generally, clients initiate connections and servers wait for and accept them. This scenario really applies to just stream sockets, because they're connection-oriented. Given what we have learned about stream sockets, IP addresses and port numbers, let's take a look at the typical steps that you'll need to take to establish a

connection, send and receive data and close a connection.

To start, an MFC WinSock client application will instantiate an element of one of the socket classes, **CAsyncSocket** or **CSocket**. (We'll examine the difference between these classes later in the chapter.) Then it will call **Create()** on this MFC socket to create the underlying socket handle. A client announces its intentions to dance by calling the **Connect()** function on the socket (this wraps the WinSock API **::connect()** function). This is one of the prototypes for the MFC socket class member:

```
BOOL Connect(LPCTSTR lpszHostAddress, UINT nHostPort);
```

Notice that, to connect to a server, the client must specify the address of the host computer and the port number of the particular service in which it's interested. Keep in mind that this is the MFC implementation of the **::connect()** function. It does a lot of work for us that we would otherwise have to do ourselves, using the 'bare-bones' WinSock API. For example, the **lpszHostAddress** parameter allows us to specify either an IP address as a string in dotted decimal notation (e.g. **"204.148.170.2")** or a Domain Name such as **"www.wrox.com"**. If we pass an IP address, the function takes care of converting the string to a 32-bit number in network byte order. However, if we pass a Domain Name, the function will call the WinSock API function **::gethostbyname()** to lookup the IP address for us using DNS. Of course, the MFC **Connect()** function also takes care of converting the port number to network byte order.

A server doesn't call the **Connect()** function because it doesn't initiate connections. Instead, a server waits around listening for connection attempts and accepts them. To accomplish this, a raw WinSock API server application would first call the **::bind()** function to register a socket with a specific port number. Then it would call the **::listen()** function to wait for clients attempting to connect to that port. Of course, the MFC socket classes provide their own version of **::bind()** (as a member function imaginatively called **Bind()**) This is the prototype for **Bind()**:

```
BOOL Bind(UINT nSocketPort, LPCTSTR lpszSocketAddress = NULL);
```

Again, MFC does some work for us here. The function will convert the port number to network byte order and will convert the IP address from a string in dotted decimal notation to a 32-bit number in network byte order. If no IP address is specified, then the function will set the IP address to the WinSock constant **INADDR_ANY**. You don't usually need to specify an address, since using **INADDR_ANY** causes the WinSock to use an appropriate network interface address.

In fact, the **Bind()** function is rarely called directly by an MFC application because a socket is automatically bound to a port by the MFC implementation of the **Create()** function which calls the **Bind()** function internally.

Once a socket is bound to a port, an MFC server calls the **Listen()** member function to wait for clients attempting to connect. The **Listen()** function is declared as:

```
BOOL Listen(int nConnectionBacklog = 5);
```

The only parameter to this function is optional (it defaults to 5 if not specified) and defines the maximum length to which the queue of pending connections can grow (the valid range is from 1 to 5). When a listening socket detects a connection attempt (we'll see how it does this later), the MFC server should call the **Accept()** member function to create a new socket which will be used to service the connection. The listening socket is then free to continue listening for new connections, while the newly created socket manages the sending and receiving of data with the client. **Accept()** is declared as:

```
virtual BOOL Accept(CAsyncSocket& rConnectedSocket,
   SOCKADDR* lpSockAddr = NULL, int* lpSockAddrLen = NULL);
```

The **Accept()** member function is called on the instance of the listening socket. We pass to it a reference to a new socket that will be used to service the connection. The reference should point to a new, uninitialized socket, since the **Accept()** function will do the work of creating it and finalizing the connection with the client. The other two

optional parameters can be used to get the address information of the client attempting to connect. You don't usually need them because you can get the IP address of the client through the newly connected socket object that you pass into this function after the connection is finalized.

Once the connection has been established, the client and/or server is free to begin the dance by sending data. Exactly which side sends the first packet depends on the protocol, but, in most cases, the client leads. For example, with the Finger protocol (port 79), the client begins by sending a user ID (as a stream of characters followed by a carriage return and line feed) immediately after the connection is established. The server responds by sending back details about the user's account on the server's host computer then it closes the connection.

To send data over a connected socket, the client and/or server can use the MFC class member function **Send()**. This is the prototype:

```
virtual int Send(const void* lpBuf, int nBufLen, int nFlags = 0);
```

The **Send()** function will return the total number of characters sent or the value **SOCKET_ERROR** if an error occurs. Keep in mind that the total characters sent maybe less than the number indicated by **nBufLen**. In this case, the application must be prepared to try again with the remaining, unsent buffer contents (we'll discuss the mechanics of this later in the next chapter). If an error is detected, the application should call the **GetLastError()** method to get the particular error code. The MFC implementation of **Send()** does absolutely nothing more than to call the raw WinSock **::send()** function, passing it the parameters untouched.

Once the data has been sent across the connected socket, it's up to the receiving end to take the trouble to read it. A client and/or server can read the contents of a socket's input buffer by calling the **Receive()** class member:

```
int Receive(void* lpBuf, int nBufLen, int nFlags);
```

The receive function will read as many characters as are currently available up to the number specified by the parameter **nBufLen**. If there are more characters to read than can fit into **lpBuf**, the application should be prepared to call **Receive()** again to read the remaining characters (we'll cover the mechanics of this later in the chapter). Like the **Send()** function, the MFC implementation of **Receive()** does nothing more than to call the WinSock API version. If an error occurs, the function will return **SOCKET_ERROR**. The application should call **GetLastError()** to get the exact error code.

From this point, the client and the server may continue to send and receive data back and forth until one side decides that the dance is over and it's time to close the connection. To close a connection, an application should call the **Close()** member function of the connected socket. The prototype is:

```
virtual void Close();
```

Calling the **Close()** function frees all resources associated with the socket, including any queued data remaining in the send or receive buffers. An error will result if any further socket functions are called after the socket is closed.

The table below recaps the function calls necessary and the sequence of the calls:

Client	**Server**
	CAsyncSocket::Create(...) Create a socket & bind the socket to a port.
	CAsyncSocket::Listen() Listen for connections.
CAsyncSocket::Create() Create a socket.	
CAsyncSocket::Connect(...) Connect to an IP address and port.	
	CAsyncSocket::Accept(...) Accept the connection and create a

new socket to handle send and receive.

CAsyncSocket::Send(...)

Send data.

CAsyncSocket::Receive(...)

Receive data.

CAsyncSocket::Receive(...)

Receive data.

CAsyncSocket::Send(...)

Send data.

CAsyncSocket::Close()

Close.

The Asynchronous Model

WinSock was designed to take advantage of the message-based architecture of Windows. Most of the extensions to the original Berkeley sockets model were made to accommodate the cooperative multitasking Windows 3.x operating system. For example, socket functions that can potentially take a while to complete can be made to return immediately before completing (i.e. they can be executed asynchronously) so that other Windows programs, including the calling program, can continue to execute and dispatch messages. When the socket function finally does complete, the WinSock DLL will post a message to the application, signaling the status of the socket. Under this model, network programming becomes event-driven, asynchronous and fits well into the Windows environment.

This is a departure from the traditional programming model employed by a large number of existing UNIX-based sockets applications. Most of these rely on the preemptive nature of the operating system to effectively manage multitasking. That is, when a socket function is called that takes a while to complete, the calling application is blocked from further execution until the function completes. However, other applications continue to run because the preemptive operating system is able to switch tasks. This architecture makes it easy to develop applications that are sequential in nature, and so fit well into the traditional character-based, command-line UNIX environment.

Windows NT and Windows 95 are also preemptive multitasking operating systems. This means that you can write socket programs that block waiting for socket functions to complete without keeping the whole system tied up. However, keep in mind that the thread calling a blocking function is suspended until the blocking task completes. If the thread is the only thread of the process, or it's the main user interface thread, all user input is halted until the function completes.

One way to avoid this is to use multiple threads, perhaps worker threads, to handle the blocking calls. This is fine if the program is designed exclusively for Windows NT or Windows 95, but if you plan to deploy the application for Windows 3.x (which doesn't support independent threads), you should use the extended asynchronous WinSock API. In fact, we recommend that you use the asynchronous model whenever possible, regardless of the target platform. It makes for more portable code and doesn't complicate your application logic with the unnecessary management of multiple threads.

Having said that, it turns out that, to get things done, there are many times you may want to use both the asynchronous approach *and* threads. This is especially true of server applications that are handling requests from multiple clients. For example, an HTTP server would most likely use the asynchronous approach to detect connection attempts and to read client requests. It would then create worker threads to send replies (usually files that are quite large) so that the main thread can remain responsive to further connection requests from other clients.

WinSock Messages

So now we've seen that WinSock can use an asynchronous model and we know the steps necessary to establish a connection, send and receive data and close the connection. But how does all this work in the message-based architecture of Windows? How does a server know that a client is trying to connect? How does a server know that a client has sent data? How does a client know that a server has replied?

The answer to all of these questions is that the WinSock DLL sends a message to the application whenever a network event occurs. Well, not just any network event, just the ones in which the application is interested.

The important thing to note here is that by registering an interest in this particular event, the program is informing the WinSock DLL that it wants the corresponding function to be made non-blocking. This means that future calls made to that function will most likely return immediately with the value **SOCKET_ERROR**. However, this doesn't mean that an actual error occurred. To determine whether an error really did occur, the application should call the member **GetLastError()** of the socket class. If the error code returned is **WSAEWOULDBLOCK**, WinSock is simply indicating that the connection didn't complete immediately; that it would have blocked and is still in progress. The DLL will send a message to the application later, when the operation completes.

To register an interest in a network event, an MFC application calls **AsyncSelect()** on their socket object. Behind the scenes, this makes use of the WinSock API function **::WSAAsyncSelect()**. The function is declared as:

```
int WSAAsyncSelect(SOCKET hSocket, HWND hWnd, u_int wMsg,
   long lEvents);
```

It's useful to examine this bare-bones WinSock API prototype because it illustrates the message-based architecture that we've been talking about. The MFC version hides some of this from us, so we'll look at that next.

This function is about the most important function in the WinSock API. It's used to request that the WinSock DLL should send a message to the window identified by **hWnd** whenever network events specified by the value in **lEvents** occur for the socket **hSocket**. The message that is to be sent is specified by the parameter **wMsg**. The value of **lEvent** is created by performing a bit-wise OR operation on any of the constants listed in the following table.

Event	Meaning
FD_ACCEPT	Send a message whenever a client is attempting to connect to a listening socket. The application can call the **Accept()** function without it blocking.
FD_CLOSE	Send a message when the socket has been closed.
FD_CONNECT	Send a message when a connection attempt is completed.
FD_OOB	Send a message when out-of-band data is available to be read.
FD_READ	Send a message when there is data available to be read.
FD_WRITE	Send a message when a socket is ready for writing.

Each call to **::WSAAsyncSelect()** overrides the setting of the last call. To cancel all event notifications, call the function with **lEvent** set to zero (this doesn't cancel events already posted to the message queue, only the delivery of future events). When an event does occur, the DLL will send the message, **wMsg**, to the window indicated by **hWnd**. The **wParam** parameter sent as part of the message will contain the socket handle; the **lParam** parameter's low-word will contain the event (one of the **FD_** values listed above) and the high-word will contain an error code if applicable.

A single message is sent for each event that occurs. For example, if there's data to be read from a socket and **FD_READ** notifications have been enabled for that socket, the DLL will post a single **FD_READ** event to the indicated window. The application must then call the appropriate receive function (based on type of socket) to re-enable further notification for this event. If after calling the receive function there's still more data to read, the DLL will

post another **FD_READ** event to the window's message queue. Otherwise, no more **FD_READ** events will be generated until more data arrives. Write notifications work in a similar fashion. For example, when data is sent to a stream socket, the **Send()** function will attempt to push through as many bytes as it can in one shot. If less than the total amount is sent, the application should wait for the DLL to post an **FD_WRITE** event before it attempts to resend the remaining bytes. The DLL will continue to post additional **FD_WRITE** events when the socket is capable of handling another attempt.

The two exceptions to the model just presented are the **FD_CLOSE** and **FD_CONNECT** events. These two events only occur once for a given socket. The **FD_CLOSE** event notifies a window that a socket has been closed. The **FD_CONNECT** event notifies a window that a socket is now connected. All of the other events can occur several times over the life of a socket.

Let's take a closer look at each of these events, including the curious **FD_OOB** event, by examining the MFC implementation of the **::WSAAsyncSelect()** function. We'll start by presenting the MFC socket classes, beginning with the class **CAsyncSocket**.

MFC Socket Classes

As we've mentioned, MFC includes two classes that embody the WinSock API. **CAsyncSocket** is the socket base class which encapsulates the asynchronous architecture that we've been discussing. **CSocket** is a synchronous, easy-to-use class derived from **CAsyncSocket**. There are also additional classes that work with **CAsyncSocket** and **CSocket** to present a more object-oriented interface to socket programming. We'll look at each of the classes throughout this chapter, beginning with the base class **CAsyncSocket**.

CAsyncSocket

CAsyncSocket hides most of the complexities of dealing with the WinSock API from the developer. It makes socket programming easier by encapsulating socket handles, data structures and WinSock API functions into a C++ class derived from **CObject**. Probably the neatest thing about **CAsyncSocket** is that it collaborates with a 'secret' class, **CSocketWnd**, which redirects event notifications back to **CAsyncSocket**. Therefore, instead of handling event notifications through the message map of a window in your application, you simply override virtual members of the **CAsyncSocket** class. The **CSocketWnd** window, which is created as a hidden window by MFC, calls these virtual functions each time a network event occurs. The functions correspond directly to the socket events listed earlier. For example, to handle the event **FD_CONNECT**, you override the class member **OnConnect()**.

To accomplish this callback scheme, you need to register your interest in the various events discussed above. Instead of using the raw **::WSAAsyncSelect()** function, you should use a wrapper provided by the **CAsyncSocket** class. The member function is named **AsyncSelect()** and its prototype is:

```
BOOL AsyncSelect(long lEvents = FD_READ | FD_WRITE | FD_OOB |
   FD_ACCEPT | FD_CONNECT | FD_CLOSE);
```

Notice the absence of the parameters **hSocket**, **hWnd** and **wMsg** that appeared in the bare-bones WinSock API. The **hSocket** parameter isn't needed because the **CAsyncSocket** object keeps track of that information internally in the attribute **m_hSocket**. The **hWnd** and **wMsg** parameters are also not needed because messages are always directed to the hidden **CSocketWnd** window and the message is always **WM_SOCKET_NOTIFY**. Really, the only thing that the MFC function does is call the WinSock API version with the appropriate parameters added (it also provides for 'thread safety' by tracking the handle to the hidden window across thread contexts). So, the only parameter that you must specifically supply is the value of **lEvents**. Remember that the value **lEvents** is the result of a bit-wise OR operation and it indicates the network events that we're interested in receiving.

When an event occurs, the hidden window will call the appropriate virtual function of the **CAsyncSelect**-derived object to handle that event. To add functionality, you must derive your own socket classes from either **CAsyncSocket** or **CSocket**. As a matter of fact, the default versions of the virtual callback functions do absolutely nothing! They are simply place holders and you should override the functions for the events in which you are

interested. Let's take a closer look at these functions and their use. They are listed in the table below with their corresponding event identifiers.

Function	Event Identifier
`OnAccept()`	`FD_ACCEPT`
`OnClose()`	`FD_CLOSE`
`OnConnect()`	`FD_CONNECT`
`OnOutOfBandData()`	`FD_OOB`
`OnReceive()`	`FD_READ`
`OnSend()`	`FD_WRITE`

OnAccept()

The **`OnAccept()`** function is called by the framework to notify a listening socket that another application is requesting a connection to the IP address and port number to which the socket is listening. The **`OnAccept()`** function is only valid for a stream socket that is bound and listening to a specific port. It indicates that the application should call the **`Accept()`** function to complete the connection and to re-enable the event for other connection attempts. Calling **`Accept()`** in response to this event will probably succeed immediately.

```
void CMyListeningSocket::OnAccept(int nErrorCode)
{
   if (nErrorCode == 0)
   {
      CMyConnectionSocket *pSocket = new CMyConnectionSocket;

      if (Accept(*pSocket))
      {
           ...
      }
      else
      {
         delete pSocket;
      }
   }
}
```

The example above illustrates the basics of establishing a connection with a remote application. This is from the perspective of the server, since **`OnAccept()`** is only valid for listening sockets. First, the program should check the value of **`nErrorCode`** to make sure that everything's OK. The only possible error code is **`WSAENETDOWN`**, indicating that the network subsystem failed. Next, the application creates a new socket to serve as the connection end-point. The new socket will be responsible for handling the send and receive operations with the connected client, so that the listening socket is free to continue listening for other connection attempts. Finally, the **`Accept()`** function is called to complete the connection. Upon a successful return, the new socket will be connected to the client and will inherit all of the properties of the listening socket. The application should then be prepared to handle send and receive operations, using the newly created socket and should destroy it when done.

OnClose()

The **`OnClose()`** function is called by the framework when the remote host terminates a connection. It is only valid for a stream socket that was previously successfully connected to a remote socket by calling either the **`Connect()`** or **`Accept()`** function.

Be careful, though. The **`OnClose()`** notification simply means that the remote socket has finished sending data. It doesn't mean that you have finished reading, so, before you destroy the closed socket, you should always call the **`Receive()`** function first to check for any unread buffered data. Also, keep in mind that either side can close a connection by calling the **`Close()`** member function of **`CAsyncSelect`**. The destructor method of **`CAsyncSocket`** will automatically call the **`Close()`** member function.

```
void CMySocket::OnClose(int nErrorCode)
{
   if (nErrorCode == 0)
   {
      // Fake an OnRead() event to force app to check receive buffer
      OnRead(0);
      . . .
      delete this;
   }
}
```

This example illustrates one possible way of handling the **OnClose()** event. It calls the **OnRead()** function, which tricks the application into trying to read the receive buffer one last time. Then, depending on the application, perhaps it could display the data to the user. (Usually, the application will have to cache the data in an internal buffer as it is received. The Finger application that we'll be developing later in this chapter uses a simple **CString** object to store incoming bytes.) Finally, since it's no longer needed, the socket is destroyed. Clearly what action you decide to take will be application-specific.

OnConnect()

The **OnConnect()** function is called by the framework to notify a client that a previous call to the **Connect()** function has completed. It can only occur once in the connection-life of a socket and is useful for one-time initialization procedures. Be sure to examine the **nErrorCode** parameter to check whether the connection was indeed successful.

```
void CMySocket::OnConnect(int nErrorCode)
{
   if (nErrorCode == 0)
   {
      m_timeConnected = CTime::GetCurrentTime();
   }
   else
   {
      ::AfxMessageBox("Connect failed!");
      delete this;
   }
}
```

The example above illustrates one possible use of the **OnConnect()** function. The socket first checks the value of **nErrorCode** to verify that the connection was successful. If it was, it updates an internal attribute of the derived class **CMySocket** to keep track of when it actually connected. This might be useful for an application that keeps track of connection statistics. If the connection fails, it simply displays a message and then destroys itself.

> **Note that OnConnect() and OnSend() apply only to CAsyncSocket. In CSocket, these methods are never called because the calls to Connect() and Send() block; they don't return until they complete.**

OnOutOfBandData()

The **OnOutOfBandData()** function is called by the framework to notify an application that there is 'urgent data' available to be read. Conceptually, the WinSock API uses **out of band data** to simulate a second independent channel for two connected stream sockets to exchange high-priority data. It's similar to an **OnRecieve()** message, but it informs the application that it should call the **Receive()** function with the **MSG_OOB** flag set to read the urgent data (the value of which is always application-specific). The documentation for the **CAsyncSocket** class suggests that you avoid using out of band data in your applications. There is some confusion about the specification under WinSock 1.1, so differences exist between implementations. The MFC documentation suggests that if you need this functionality, you should create a second socket for passing urgent or high priority data.

OnReceive()

The **OnReceive()** function is called by the framework to notify the socket that there is data in the buffer that can be retrieved by calling the **Receive()** member function. It indicates that calling **Receive()** will probably succeed immediately. If any data remains in the buffer after calling **Receive()**, the DLL will post another **FD_READ** event which will trigger **OnReceive()** again.

```
void CMySocket::OnReceive(int nErrorCode)
{
   TCHAR tmp[512 + 1];
   int nCount;

   switch (nCount = Receive(tmp, 512, 0))
   {
      case SOCKET_ERROR:
         // Call GetLastError() to determine the cause
         . . .
         break;

      case 0:
         // The other end gracefully closed the socket.
         break;

      default:
         tmp[nCount] = '\0';
         m_strRecvBuffer += tmp;
         break;
   }
}
```

This example demonstrates one possible implementation of the **OnReceive()** function. The function calls **Receive()** to read up to 512 characters (if there are more than 512 waiting to be read, the DLL will post another **FD_READ** event, triggering the **OnReceive()** function again). If an error is reported by the **Receive()** function, **GetLastError()** is called to determine the exact error code. The application should check the return code and handle it appropriately. If **Receive()** returns zero, this indicates that the connection has been closed by the connected socket. If the **Receive()** function returns a value greater than zero, the characters read are appended to the end of a **CString** object, **m_strReceive**.

> *Warning: This is not necessarily a good implementation, since a socket can receive any kind of data, and lots of it. Unless you're certain that your application will only be receiving small string values, I would recommend against using a* **CString** *object as a receive buffer.*

OnSend()

The **OnSend()** function is called by the framework to notify an application that it can send data by calling the **Send()** function. It indicates that calling **Send()** will probably succeed immediately. With each call to the **Send()** function, the WinSock DLL will try to push as many bytes through as it can in one shot. It will then post another **FD_WRITE** event when it is ready for more, triggering **OnSend()** again. Your application must keep track of how much was sent with each call to the **Send()** function and be prepared to try again if the DLL can't send it all.

```
void CMySocket::OnSend(int nErrorCode)
{
   // Make sure there is something to send
   if (m_strSendBuffer.IsEmpty())
      return;

   //  Try to send it all!
   int nSent = Send(m_strSendBuffer, m_strSendBuffer.GetLength());

   if (nSent == SOCKET_ERROR)
   {
      //  See what happened
      switch (GetLastError())
```

```
         {
            // These are OK, we'll try again later.
            case WSAEWOULDBLOCK:
            case WSAEINPROGRESS:
               return ;

            default:
               // Yikes!
               break;
         }
      }
      else
      {
         // Remove what was sent from the send buffer
         m_strSendBuffer = m_strSendBuffer.Mid(nSent);
      }
   }
```

The example above illustrates one possible implementation of the **OnSend()** function. It assumes that there is a member variable named **m_strSendBuffer** of type **CString** which contains the data to be sent. First, the function checks to see whether there is, in fact, anything to send by calling the **IsEmpty()** method of the string. If there is something to send, it attempts to send the entire string all at once and checks the return code to see if it was successful. If the return code is **SOCKET_ERROR**, it calls the **GetLastError()** function to get the exact error code. Error codes of **WSAEWOULDBLOCK** and **WSAEINPROGRESS** are acceptable in this example because the DLL will post another **FD_WRITE** event when it can. Other errors should be handled appropriately. If the value of **nSent** is something other than **SOCKET_ERROR**, the application removes the sent characters from the string. Remember that **nSent** may be less than the total to send. If it is, only the sent characters are removed from the send string and the WinSock DLL will post another **FD_WRITE** event when it can.

> *Just as a reminder,* **OnConnect()** *and* **OnSend()** *apply only to* **CAsyncSocket**. *In* **CSocket**, *these methods are never called because the calls to* **Connect()** *and* **Send()** *don't return until they complete.*

The **CAsyncSocket** class has an additional **virtual** function, called **ConnectHelper()**, that is of some interest. It's not exactly a callback function, though; it's a **protected virtual** function that you can override to complete the connection process. It's called by **CAsyncSocket::Connect()** as the final step before actually attempting to connect. This makes it easy to override the actual connection code without overriding the first part of **Connect()** which is responsible for converting a dotted decimal IP address or a domain name into a valid network address. The **CSocket** class overrides this function to pump messages waiting for the connection to complete before returning, making the connection function synchronous. Another possible reason for overriding the function is presented below:

```
BOOL CMySocket::ConnectHelper(const SOCKADDR* lpSockAddr,
                              int nSockAddrLen)
{
   // Set the status code and notify of event
   SetStatus(CMySocket::statusConnecting);
   BroadcastEvent(CMySocket::eventConnecting);

   // Register an interest in Connection Event. This will
   // make this socket asynchronous.
   AsyncSelect(FD_CONNECT);

   // Do the super class method. If it returns true then we
   // were connected immediately.
   if (CAsyncSocket::ConnectHelper(lpSockAddr, nSockAddrLen))
   {
      // Execute the logic already coded to handle a connection.
      OnConnect(0);
      return TRUE;
   }
   else
```

```
        {
            // Return OK if connection is deferred.
            return (GetLastError() == WSAEWOULDBLOCK);
        }
    }
```

The function begins by calling the **SetStatus()** and **BroadcastEvent()** functions. These are not **CAsyncSocket** functions; they are methods added to the derived class, **CMySocket**. The parameter passed to **SetStatus()** is an enumerated value that is stored internally by the object to keep track of its current state. Here, the status is set to **statusConnecting** to indicate that a connection attempt is in progress. **BroadcastEvent()** is used to send a message to any window that is interested in getting status updates for the socket. Of course, the event can be any value, not just the standard WinSock events, because this is not part of MFC, but is application-specific. Here, the enumerated value **eventConnecting** (which doesn't correspond to any of the standard WinSock events) is being sent. **BroadcastEvent()** doesn't have to send messages; it could call some function of another object, perhaps a **CDocument** object, or update a status bar instead.

Next, the **ConnectHelper()** calls **AsyncSelect()** to register an interest in future connection attempts. This automatically sets up the socket to be non-blocking and asynchronous. This means that a future call to the WinSock API **::connect()** function would probably return immediately with an 'error', indicating that the function would have blocked. This is exactly what we want to happen. In fact, the very next statement calls the **CAsyncSocket::ConnectHelper()** function, which simply turns around and calls the WinSock API **::connect()** function. We check the return code and if it returns successfully (not likely, as we've just explained), the connection was established. In this case, we call the **OnConnect()** function because we want to perform the same logic that would have been executed had the function completed asynchronously. If the call to the **CAsyncSocket::ConnectHelper()** function returns with an error, we check to see if the error code is **WSAEWOULDBLOCK**, which is not really an error at all. In this case, the **OnConnect()** function will be called by the framework when the connection is completed.

ConnectHelper() simply gives us a little more control over the entire process. We'll be using it in the Finger application that we develop later in this chapter. When we build Finger, we'll be developing a **CAsyncSocket**-derived class that implements the **SetStatus()** and **BroadcastEvent()** functions described above, but, before we do that, let's take a look at the other MFC socket class, **CSocket**.

CSocket

CSocket is derived from **CAsyncSocket** and its goal is to provide an easy-to-use object that deals with sockets in a synchronous fashion. It collaborates with a couple of other classes to make sending and receiving data to and from a socket much the same as writing and reading data to and from a disk file. This makes it easier to use than the **CAsyncSocket** model, but in some cases, it's a little less flexible.

CSocket works with the MFC class **CSocketFile** to send and receive socket data over a network. Once a **CSocket** is associated with a **CSocketFile**, you can call the **CSocketFile** class members to read and write directly to the socket as if it were a file. In fact, you can even associate a **CArchive** object with **CSocketFile** to support the serialization of objects to and from a socket. Of course, for this to work, both the sending and receiving side of the connection must speak MFC. In other words, it doesn't make sense to serialize an object to a socket if the receiver isn't also an MFC application that can de-serialize the object. This fact makes **CArchive** unusable when you're writing well-known Internet applications like Finger and Telnet. The protocols for these applications simply do not allow for the storage schemes used by **CArchive**. However, you can still use **CSocketFile** as a stand-alone file object with any Internet applications, since it sends to and receives from a socket using a simple stream of characters.

CSocket differs from **CAsyncSocket** in that it's designed to be mostly synchronous. In other words, **CSocket** functions generally do not return to the caller until their operations have completed. For example, the **ConnectHelper()** function, which is overridden in **Csocket**, implements a message pump, looping until it intercepts the **FD_CONNECT** message. When the message is received, the function returns to the caller with a value indicating the success of the operation. To the application, this means two things. The first is that when the

Connect() function is called, it won't return until the socket is in fact connected (assuming there are no errors). The second is that the **OnConnect()** callback function will never be called, since the **FD_CONNECT** message is intercepted by the message loop in **ConnectHelper()**. Thus, the **Connect()** function is a quasi-synchronous operation that makes sure Windows messages are dispatched while it waits for the one message in which it's interested that indicates the connect has completed.

The other synchronous **CSocket** function is **Send()**. It also implements a message pump that is used when it's unable to send all of the requested data at once. It loops, peeking for the **FD_WRITE** message and doesn't return until all of the data has been sent. This is mostly done to accommodate the use of **CSocketFile** and **CArchive**. These two classes really make socket programming much easier, especially when you need to send C++ objects back and forth between MFC applications. As an example, the following snippet of code shows the steps necessary to send an object, **pSomeObject**, to a socket named **pSomeSocket**:

```
CSocketFile file(pSomeSocket);
CArchive ar(&file, CArchive::store);

pSomeObject->Serialize(ar);
```

Simple! On the receiving side, it's the same process except that the archive should be created with the **CArchive::load** flag set instead. Keep in mind that the **OnSend()** callback of the sending socket is never called because the message loop in **SendChunk()** (a helper for **Send()**) intercepts the **FD_WRITE** event. However, on the receiving side, the **OnReceive()** *will* be called when data arrives to the socket.

To illustrate the use of **CSocket**, **CSocketFile** and **CArchive**, we're going to build both the client and the server for a truly interesting application (well, *I* think so) at the end of this chapter. The application will allow you to control the appliances in your home from anywhere in the world! Yes, we'll build a server application that accepts commands to turn lights on and off, control dimmer switches and even open and close drapes. We'll also build a client application that will connect to our 'magic' server and send it the commands to control your home. Don't believe me? It's actually pretty simple and it makes for a good example of the power of using the MFC serialization feature with sockets.

But before we build the **CSocket** sample application, it makes sense to look a complete application that uses **CAsyncSocket**. This will give us a better understanding of the foundation upon which **CSocket** is built. So let's begin now by constructing an Internet Finger client application.

Building a Finger Client

It's time to roll up our sleeves and get down to programming our first real Internet application. We'll start with a simple Finger client program that will allow us to look up information about users in the databases of host computers all over the world (well, the ones attached to the Internet anyway). Why begin with Finger? Well, it's a relatively simple protocol that illustrates almost all of the elements of a complete Internet application. (I say *almost* because we'll be approaching it strictly from a client's perspective. Server functions like **`Listen()`** and **`Accept()`** are not used.) For this reason, and because it's fairly easy to implement with a small amount of coding, Finger has become the 'Hello, World' of socket programming.

> *We have supplied a simple Finger server on the CD with the book so that you can easily test the client. We won't be discussing the server in this chapter, but full source code is provided. It uses similar techniques to the home automation server that you'll see later in the chapter.*

Let's begin with an example of the expected input and output for Finger. For example, to find out who **`usrname`** is, anyone with a Finger client program could issue this command:

```
finger usrname@somecompany.com
```

Finger is most often used in this command-line format because it was born on UNIX machines, but, of course, we'll be adding a nice Windows interface to the program with edit boxes and command buttons. Anyway, the result is that the server responds with something like this:

```
Login name: usrname              In real life: User Name
Directory: /usr/users/usrname            Shell: /bin/sh
Last login Wed Jan  3 15:21 on ttyp6 from 206.2.188.172
No Plan.
```

It's not terribly interesting, but it's exciting when you see it work for the first time! The specific information that is returned is up to the server, but the RFC suggests that the server should return the user's full name, address or office location and telephone number.

> *RFC stands for Request For Comments. RFCs are documents describing the behavior and requirements for various Internet-related protocols. Since the behaviors described by RFCs are works in progress, a single protocol such as Finger may have many RFCs through the course of its development. Each RFC receives a number with larger numbers representing more recent RFCs than smaller numbers. Currently Finger is described by RFC 1288.*

Most servers will include a user's logon name, the last time that user logged on or checked mail, and the contents of a plan file if one exists (a plan file is a simple text file that can contain anything the user wants to say to everyone that Fingers).

The Finger application, like the majority of Internet programs, follows the client/server model. A client begins by attempting to establish a connection to the user's host computer. Remember that a client must indicate both the address of the host computer and a port number in order to establish a connection. In the example above, the address of the host computer is identified by the domain name **`somecompany.com`**, which must be converted to an IP address before it can be used. The port number is always 79, as shown in the table earlier in the chapter. Next, assuming the connection was successful, the client will send an ASCII command to the server followed by a CR/LF pair. The command is usually just the user's logon name (**`usrname`** in the example above).

The server responds by sending back the requested information. Again, the data may be in any format, so I would advise that you don't try too hard to parse it. About the only thing that you can count on is that each line will be separated by a CR/LF pair. That's about it. When the server is done, it terminates the connection by closing the socket. The client should then read the socket and display the information to the user. Simple!

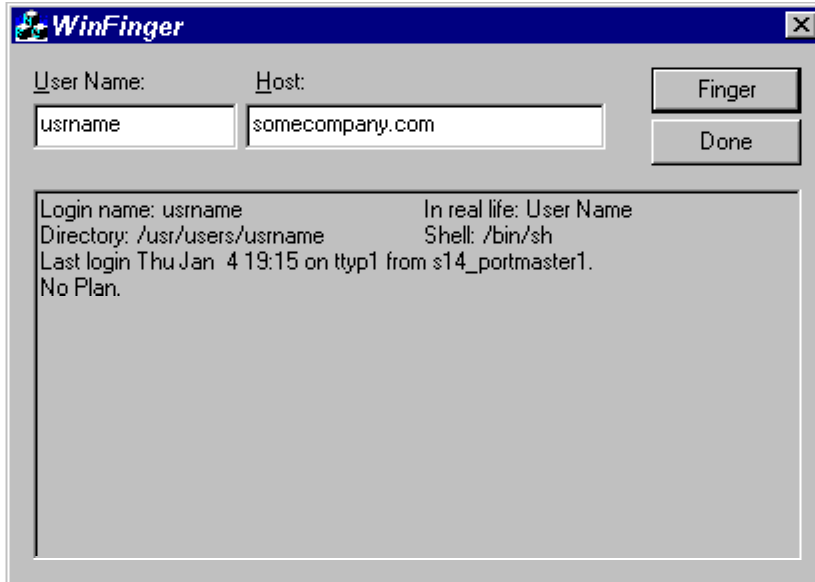
Finger Client	**Finger Server**
The client creates a socket with **`Create()`**.	The server socket is created, bound to port and listening.
The client attempts a connection with **`Connect()`** using the Finger port (79).	
	The server accepts the connection.
In **`OnConnect()`**, the client registers an interest in other events using **`AsyncSelect()`**.	
In **`OnSend()`**, the client sends Finger a command by calling **`Send()`**.	
	The server receives a command and sends back the requested information.
In **`OnReceive()`**, the client receives the results of Finger by calling **`Receive()`**(**`OnReceive()`** may be called a number of times until all the information is received.)	Once the information's been sent, the server closes the connection.
...	
`OnClose()` notifies the client that the connection is closed	

Constructing the Interface

Now that we understand the basics of what we need to do, let's take a few minutes here to discuss the design goals for the application's user interface. The main goal is to keep it simple, both for the user and the application developer if we can help it.

From the user's perspective, there needs to be a way to enter the name or IP address of a host computer, a way to enter the logon name of a user to Finger and a way to submit the Finger request. The user of our application will want to be able to enter this information quickly, just like they can when they use the command-line version, so we shouldn't hamper them with unnecessary menus and dialog boxes. The application does only one thing, so there really is no need for too many bells and whistles.

From the developer's perspective, there needs to be a way to get the same information from the user and a way to display the results. Again, there's no need to complicate the application and we could probably live without an extended document-view architecture. In fact, I think a simple dialog-based interface will do just fine. Here, you can see what the final version should look like:



You can find this application on the CD in the `WinFinger` directory. To create this application, run MFC AppWizard (exe) and make sure that you select the radio button labeled Dialog based on Step 1. Also, be sure to check the Windows Sockets check box under the question, Would you like to include WOSA support? on Step 2 of the AppWizard. You can leave all the other options at their default settings.

The next step is to edit the dialog template that was created for us. Add three edit boxes to the dialog as illustrated above; one for the host name, one for the user name and one for the response. The response edit field should have the Multi-line style bit checked. Also, change the OK button so that it reads Finger and has the identifier **IDC_FINGER**. This will serve as our submit button. You should also set the Cancel button's label to read Done.

Now activate ClassWizard and add a member variable to the dialog class of type **CString** for each of the edit boxes. I'll use **m_strUser**, **m_strHost** and **m_strResponse** throughout this example. Finally, add a member function named **OnFinger()** to handle the **BN_CLICKED** message for the button **IDC_FINGER**.

Compile your program and we're ready to start coding! We will begin by creating a new socket class.

The Finger Socket Class

For this program, we'll use inheritance to construct a new class, **CFingerSocket**, derived from **CAsyncSocket**. We use **CAsyncSocket** here to clearly illustrate the use of **AsyncSelect()** and the callback functions that make the entire application asynchronous. We've also included a **ConnectHelper()** function in this class to show how to override and extend the connection function.

The **CFingerSocket** class will:

- Handle the sending of the user id in response to an **OnSend()** callback.
- Handle the receiving of the host results in response to an **OnReceive()** callback.
- Keep track of its current state.
- Broadcast important events to the WinFinger application window.

CFingerSocket will encapsulate all of the logic of the Finger protocol. To use it, an application simply creates an instance of the object, sets the user ID that is to be sent, then it connects it to the remote host. **CFingerSocket** will

do the rest. Let's take a look at the header file, `FingerSocket.h`.

If you take a look at the section towards the beginning, labeled **`// Definitions`**, you'll notice two items. The first is a compiler directive checking to see whether **`WM_SOCKETEVENT`** has been previously defined. **`WM_SOCKETEVENT`** is the message that will be sent to your window when something interesting happens (e.g. the socket is connected).

```
#ifndef WM_SOCKETEVENT
#error You must define 'WM_SOCKETEVENT' as a WM_USER message.
#endif
```

The value of **`WM_SOCKETEVENT`** is not defined here; it's left up to the developer to choose the appropriate **`WM_USER`** value to assign to it. The benefit of not hard-coding it here is that you can give it any value that makes sense to the particular application that uses this class. Thus, you can safely use the message identifier without it conflicting with another user-defined message in your application. We don't have any other user-defined messages in this application but I always try to code with reuse in mind.

The bottom line is that you should have a declaration something like the one below in your application header file, or better yet, in the `StdAfx.h` header file. If you don't, you'll get a compiler error reminding you to declare it.

```
   #define WM_SOCKETEVENT (WM_USER + 1)
```

Next, you'll notice a very strange definition for **`ON_WM_SOCKETEVENT()`**. This definition is a new message map signature for the **`WM_SOCKETEVENT`** message that we've just discussed:

```
#ifndef ON_WM_SOCKETEVENT
#define ON_WM_SOCKETEVENT() \
   { WM_SOCKETEVENT, 0, 0, 0, AfxSig_vwwh, \
      (AFX_PMSG)(AFX_PMSGW)(void (AFX_MSG_CALL CWnd::*)(UINT, UINT,
SOCKET))OnSocketEvent },
#endif
```

Note that message map signatures are considered part of the implementation of MFC, so are subject to change.

This is just a small convenience for writing the message handler. The message signature will take care of splitting the **`wParam`** and **`lParam`** values of the message into the appropriate components defined for the message. To use it, first add the definition to your window's message map, something like we did in `WinFingerDlg.cpp`:

```
BEGIN_MESSAGE_MAP(CWinFingerDlg, CDialog)
//{{AFX_MSG_MAP(CWinFingerDlg)
//}}AFX_MSG_MAP
   ON_WM_SOCKETEVENT()
END_MESSAGE_MAP()
```

You'll probably want to add it outside of the ClassWizard area, as I've done here. ClassWizard will not be able to help you with this message map entry.

The next thing to do is to declare your handler function. You must call it **`OnSocketEvent()`** (because that is the name included in the message signature) and you should declare it as:

```
afx_msg void OnSocketEvent(UINT nEvent, UINT nErrorCode, SOCKET hSocket);
```

We'll discuss how this function gets invoked shortly, but before that let's take a look at the rest of the header file which defines the interface for **`CFingerSocket`** and is shown below:

```
class CFingerSocket : public CAsyncSocket
{
   DECLARE_DYNAMIC (CFingerSocket);
public:
   // Socket Status Codes
```

```
    enum SocketStatus
    {
       // enumeration values not shown (see code on CD)
    };

    // Socket Events
    enum SocketEvent
    {
       // enumeration values not shown (see code on CD)
    };

// Constructors
public:
    CFingerSocket(LPCTSTR lpszUser, CWnd *pWndNotify = NULL);

// Attributes
public:
    // Accessors
    UINT GetStatus()                        { return m_nStatus; }
    void SetStatus(UINT nStatus)            { m_nStatus = nStatus; }
    void GetText(CString &str)              { str = m_strRecvBuffer; }

protected:
    CWnd* m_pWndNotify;         // This is who we notify
    SocketStatus m_nStatus;     // The socket status
    CString m_strSendBuffer;    // Our simple send Buffer
    CString m_strRecvBuffer;    // Out simple receive Buffer

// Operations
public:

// Overridables
public:
    // Socket Methods
    virtual BOOL ConnectHelper(const SOCKADDR* lpSockAddr,
                               int nSockAddrLen);
    virtual void OnClose(int nErrorCode);
    virtual void OnConnect(int nErrorCode);
    virtual void OnReceive(int nErrorCode);
    virtual void OnSend(int nErrorCode);

    // Other Methods
    virtual void BroadcastEvent(SocketEvent nEvent, UINT nErrorCode = 0);

// Implementation
public:
    virtual ~CFingerSocket();
};
```

The constructor for the **CFingerSocket** class takes two parameters: a string of characters and a pointer to a window. The string of characters, **lpszUser**, is obviously the logon name of the user which we are going to Finger. The other parameter, **pWndNotify**, is a pointer to a window to which we want all **WM_SOCKETEVENT** messages for this socket sent. Thus, when you instantiate a **CFingerSocket** object, the object will store the user's logon name and it will be sent soon after the socket is connected. The socket will also store a pointer to the window and it will broadcast **WM_SOCKETEVENT** messages to it as things move along. As you might have guessed, the function **BroadcastEvent()** is the function responsible for sending the messages and the events that it can broadcast are declared as enumerated values under **enum SocketEvent**.

CFingerSocket also keeps track of its current status. As events are triggered and the socket moves from one state to another, the member variable **m_nStatus** will be assigned one of the values in the enumeration **enum SocketStatus**. You can call the accessor method **GetStatus()** at any time to query the current status of the socket. There isn't a whole lot of use for this, except perhaps to check the status of a socket before performing another operation on that socket. For example, suppose the user clicks on the Finger button of our application while the socket is waiting to be connected. The socket status member variable would tell us **statusConnecting** and you could then either cancel the outstanding request or ask the user to wait until it finishes.

Next, take a look at the attributes **m_strSendBuffer** and **m_strRecvBuffer**. As their names imply, they are both **CString** objects and they are used to store data that is to be sent and data that is received. The send buffer is assigned the value **lpszUser** in the constructor. Later, when the **OnSend()** callback function is called, **CFingerSocket** will send whatever it finds in **m_strSendBuffer** and it will empty the buffer as it goes. As data is received, **CFingerSocket** will append the received characters to **m_strRecvBuffer**. The **GetText()** accessor method can be called at any time to get the contents of the receive buffer.

A word of caution. Using a **CString** object as a receive buffer can be dangerous. You never know what the host socket will return in response to your Finger request, or how much data is going to be returned. (The RFC states that if no user ID is sent, the server should return a list of all users either logged on or that have accounts. This could be a very large amount of data!) Normally, you should be OK because the protocol specifically states that returned data must be ASCII with CR/LF pairs separating each line. However, you never know when you're going to come across a misbehaved Finger server. So, while using a **CString** is not the greatest for a commercial implementation, it is used here because it clearly illustrates the mechanics of **OnReceive()** without complicating the code with extraneous buffering logic. On the other hand, using a **CString** as the send buffer is perfectly fine here because we are completely controlling its contents.

The last thing to notice at the bottom of the header file is the overrides of the **CAsyncSocket** virtual functions. Notice that not all of them are overridden, just the ones that make sense to our application. We'll be taking a look at the code for each one shortly, but first, let's begin by examining the constructor for our class. It appears as:

```
CFingerSocket::CFingerSocket(LPCTSTR lpszUser, CWnd* pWndNotify)
{
   // Initialize the status code
   m_nStatus = CFingerSocket::statusReady;

   // Save the window to notify
   m_pWndNotify = pWndNotify;
   m_strSendBuffer = lpszUser;

   // Protocol requirement
   m_strSendBuffer += "\r\n";
}
```

Nothing too surprising here. The socket status is initialized, a pointer to the window to notify is saved, and the logon name to Finger is added to the send buffer. Also, a CR/LF pair are added to the end of the send buffer so that the host can determine when we are done sending the command. This is a protocol requirement and if you forget to send it, your client will sit there waiting forever for the host to send its reply. Of course, the server will be waiting for you to finish sending the command!

If you look back to the header file at the prototype of the constructor, you'll notice that there's no default value for the parameter **lpszUser**. You must supply a value when you create an instance of the class. The value will be added to the send buffer so that it's ready to be sent as soon as the socket is connected. **CFingerSocket** is thus able to manage the requirements of the protocol without the application needing to get involved in the handling of read and write notifications. However, even though **CFingerSocket** is a self-contained and self-managing object, it still should be able to at least notify the application of its progress. That's the purpose of the optional parameter **pWndNotify**, which is a pointer to a window that will receive event notifications through its message map. That is, as interesting things happen to the socket (such as when it receives data, sends data, etc.) the socket will broadcast the events to the window. It does so by calling its own **BroadcastEvent()** method, shown below:

```
void CFingerSocket::BroadcastEvent(SocketEvent nEvent, UINT nErrorCode)
{
   if (m_pWndNotify)
   {
      // Send the event
      m_pWndNotify->SendMessage(WM_SOCKETEVENT,
         MAKEWPARAM(nEvent, nErrorCode),
         (LPARAM) m_hSocket);
   }
}
```

The function first checks to see if there is, in fact, 'anyone listening'. If there's a window to send a message to, the function sends the **WM_SOCKETEVENT** message. Remember, this is the user-defined message that you have to declare and that triggers the **OnSocketEvent()** handler. The event and the error code are passed to your handler in the **wParam** parameter. The **lParam** will contain the handle of the socket sending the message. It's best to use the provided **ON_WM_SOCKETEVENT()** message map signature defined in the header file, since it will take care of splitting the parameters out for you. By the way, it's easy for you to change **BroadcastEvent()** so that it calls a function rather than sending a message. For example, you may want it to call a function of a **CDocument** object. Also keep in mind that it really isn't needed to create a simple Finger client, but we'll use it in our WinFinger application to keep the user informed of the progress by updating the caption of the window as the process moves along.

So when is this function called and what calls it? Take a look at the **ConnectHelper()** function which has the following implementation:

```
BOOL CFingerSocket::ConnectHelper(const SOCKADDR* lpSockAddr,
                                  int nSockAddrLen)
{
   // Set the status code and notify of event
   SetStatus(CFingerSocket::statusConnecting);
   BroadcastEvent(CFingerSocket::eventConnecting);

   // Pay attention to connect notification
   AsyncSelect(FD_CONNECT);

   // Attempt the connect
   if (CAsyncSocket::ConnectHelper(lpSockAddr, nSockAddrLen))
   {
      // Connected Now!
      OnConnect(0);
      return TRUE;
   }
   else
   {
      // Return OK if connection is deferred.
      return (GetLastError() == WSAEWOULDBLOCK);
   }
}
```

Remember that **ConnectHelper()** is a virtual method of **CAsyncSocket** that is called by the framework as part of the **Connect()** function. We override it here because we want more control over how the socket connects to a remote host. The first thing we do is set the socket status and then broadcast the **WM_SOCKET** message to whatever window is interested. Remember that the connection may take a while to complete, so the status is set to reflect an 'in progress' value.

Next, we call the **AsyncSelect()** function to make the operation asynchronous. This tells the WinSock DLL not to block waiting for the operation to complete. Instead, it should post an **FD_CONNECT** message to our socket (actually, to the **CSocketWnd** window associated with our socket) when it is done. All other events are disabled for now. They will be re-enabled as needed, as we'll see later.

Next, we call the **ConnectHelper()** function of our super class, **CAsyncSocket**. If you peek at the MFC source code for the function, you'll see that it does nothing more than to call the WinSock API **::connect()** function, so we check the return code to see whether the connect was successful. If it was, it indicates that our socket was able to complete the connection right away. However, this is very unlikely, since we told the API not to block waiting for the connection, but it *is* possible (in which case no **FD_CONNECT** would be posted to the socket), so we code for it by calling our **OnConnect()** handler directly. We have some code in **OnConnect()** that we want to execute whether the call completes synchronously or asynchronously, so we make sure that the function gets called. Now, if the call to the **CAsyncSocket**'s **ConnectHelper()** is *not* successful (the more likely scenario), we check the last error to see whether it's the value **WSAEWOULDBLOCK**. This is OK and it's what we expect. It simply means that the connection is still in progress. Any other error is a problem and our function will return the Boolean value **FALSE**.

Next, let's take a look at our callback function, **OnConnect()**, which will be called by the framework when the connection attempt is finally completed. Note that I said *completed* and not *established*. The function will be called as a result of either a successful or unsuccessful connection attempt.

```
void CFingerSocket::OnConnect(int nErrorCode)
{
   // Check to see if the connection was OK
   if (nErrorCode == 0)
   {
      // Report the new status
      SetStatus (CFingerSocket::statusConnected);
      BroadcastEvent (CFingerSocket::eventConnect);

      // Pay attention to other events
      AsyncSelect(FD_READ | FD_WRITE | FD_CLOSE);
   }
   else
   {
      // Report the problem
      SetStatus(CFingerSocket::statusReady);
      BroadcastEvent(CFingerSocket::eventConnect, nErrorCode);
   }
}
```

So first thing's first. We check the value of the parameter **nErrorCode**. If it's anything other than zero, the connection failed. In this case, we simply reset our socket status back to the original value and broadcast the result back to the notification window. On the other hand, if the connection is OK, we change our status code to **statusConnected** and broadcast the news to the notification window. Then, and most importantly, we ask the WinSock API to send us notification messages whenever the socket receives data, sends data, or is closed. If you forget to do this, your socket will be deaf, dumb and blind!

Once the socket has been connected, the very next notification the WinSock API will send is **FD_WRITE**, indicating that the socket is ready to accept data to send. This event will trigger our **OnSend()** callback function which is implemented as:

```
void CFingerSocket::OnSend(int nErrorCode)
{
   if (m_strSendBuffer.IsEmpty())
   {
      // Notify whoever is interested then get out
      BroadcastEvent(CFingerSocket::eventSend);
      return;
   }

   // Try to send what's in the buffer
   int nSent = Send(m_strSendBuffer, m_strSendBuffer.GetLength());

   if (nSent == SOCKET_ERROR)
   {
      UINT nErrorCode = GetLastError();

      //  See what happened
      switch (nErrorCode)
      {
         // These are OK and expected!
         case WSAEWOULDBLOCK:
         case WSAEINPROGRESS:
            return;

         default:
            // Report the error.
            BroadcastEvent(CFingerSocket::eventError, nErrorCode);
            break;
      }
   }
```

```
    else
    {
       // Remove what was sent from the send buffer
       m_strSendBuffer = m_strSendBuffer.Mid(nSent);

       // Notify whoever is interested.
       BroadcastEvent(CFingerSocket::eventSend);
    }
 }
```

The **OnSend()** function will be called whenever the socket is capable of accepting data to be written to the connected socket. It's called once, almost immediately after a successful connection, but is not called again until after you call the **Send()** function. **Send()** is called an **enabling** function because calling it enables further **OnSend()** notifications. In other words, when you call **Send()**, the API will try to write as much as it can to the socket. Later, when it's ready to handle another write attempt, it will post another **FD_WRITE** event, triggering the **OnSend()** function again.

Our implementation first checks the contents of **m_strSendBuffer** to see if there is in fact anything to send. Remember that this member variable is where the logon name of the user to Finger is stored by the constructor. The first time **OnSend()** is called, the function **IsEmpty()** should return **FALSE**. Therefore, the **CAsyncSocket Send()** method will be called to try to send the entire logon name all at once. (There shouldn't be any problem with that!) We then check to see how many characters were actually sent, or if an error occurred. The sent characters are removed from the send buffer so that the next time **OnSend()** is called (and it will be called again after each call to **Send()**) only the remaining, unsent portion is transmitted. If an error occurs, we check to see if it's one of the acceptable errors listed in the **switch** statement. Other errors are broadcast to the notification window.

After the host receives the information we sent, it responds by sending back the information about our user. The **OnReceive()** function of our socket will be called each time data arrives and can be successfully read (maybe). Our implementation appears below:

```
void CFingerSocket::OnReceive(int nErrorCode)
{
   TCHAR tmp[BUFSIZE + 1];
   int nCount;

   switch (nCount = Receive(tmp, BUFSIZE, 0))
   {
      case SOCKET_ERROR:
         // Report it!
         BroadcastEvent(CFingerSocket::eventError, GetLastError());
         break;

      case 0:
         // The server gracefully closed the socket.
         OnClose(0);
         break;

      default:
         // Add the text to the receive buffer
         tmp[nCount] = '\0';
         m_strRecvBuffer += tmp;
         break;
   }

   // Tell whoever cares of the the event
   BroadcastEvent(CFingerSocket::eventReceive);
}
```

The first thing we do is call the **Receive()** function to read up to **BUFSIZE** bytes at a time (**BUFSIZE** is defined as 512 for the purposes of this example). This is a completely arbitrary number. The **Receive()** function will do what it can to read the requested number of characters. It may actually read less, but never more, than this number. If there are fewer than 512 characters to read, it will return immediately having read only what was available. In other words, it doesn't wait for that many to be available before returning. If there are more than 512, it will read up to 512

and the WinSock API will post another **FD_READ** event, triggering **OnReceive()** once again. Note that **Receive()**, like **Send()**, is an enabling function. This means that **OnReceive()** will be called once when data first arrives. It doesn't get triggered again until after you call **Receive()** to read the buffer contents, and only then if there's more data to be read.

> *There's another approach to receiving data that may be useful. Rather than hard-coding the size of the temporary buffer used to receive the data, some applications make a call to the* **CAsyncSocket** *member function* **IOCtl()** *to first determine the total number of bytes queued for the application to receive. For example:*

```
DWORD dwBytes;
pSocket->IOCtl(FIONREAD, &dwBytes);
```

> *Then, the application can allocate the exact buffer size to receive the data. This avoids extra calls to* **OnReceive()** *that result when the buffer is too small. This might make your application a little snappier, but is usually not necessary.*

As the data arrives, our function slaps a string-terminator at the end and then appends it to the **CString** receive buffer. If an error is encountered, it's broadcast to the notification window. Also, if the total number of characters read during one call to **Receive()** is zero, this indicates that the server has closed the connection.

The last method of **CFingerSocket** is the **OnClose()** handler. **OnClose()** is called when either side, the server or the client, closes the connection. Our implementation appears below:

```
void CFingerSocket::OnClose(int nErrorCode)
{
   // Update our status code
   SetStatus(CFingerSocket::statusReady);
   BroadcastEvent(CFingerSocket::eventDisconnect);
}
```

As you can see, it does nothing more than to reset its internal status code and then broadcast the event to the notification window. For our application, the server is the one the closes the connection. It does so after spilling its guts about the user we are Fingering. Therefore, this event is of great importance to our application. We'll use this event as an indication that it's time to display the contents of the receive buffer to our user. Let's shift gears now and focus our attention on using the **CFingerSocket** class within our application.

The CWinFingerDlg Class

WinFinger is a dialog-based application. Our main window, **CWinFingerDlg**, manages the use of the socket class just described by creating a new socket each time the user presses the Finger button. It also responds to the broadcast messages sent by the socket by displaying the socket progress and by displaying the host response when it arrives. **CWinFingerDlg** has one member variable to keep track of the socket object, declared as a pointer to a Finger socket, as shown below:

```
CFingerSocket* m_pSocket;
```

The window's constructor sets this pointer to **NULL** and the handler for the **BN_CLICKED** event of the Finger button does the work of creating the socket. The handler, **OnFinger()**, is implemented as:

```
void CWinFingerDlg::OnFinger()
{
   // Update the MFC member variables that are connected
   // to the controls
   UpdateData(TRUE);

   if (m_strHost.IsEmpty())
   {
      // The loopback address
      m_strHost = "127.0.0.1" ;
```

```
   }

   // Reset the response value
   m_strResponse.Empty();
   UpdateData(FALSE);

   // Delete the socket it it exists
   if (m_pSocket)
   {
      delete m_pSocket;
   }

   // Create a new socket to use for this finger request
   // Set it up to send notifications to this window
   m_pSocket = new CFingerSocket(m_strUser, this);

   if (m_pSocket->Create())
   {
      // Try to connect.  This will most likely return right away
      // and then send 'Connecting' event since it would otherwise block.
      if (!m_pSocket->Connect(m_strHost, 79))
      {
         delete m_pSocket;
         m_pSocket = NULL;

         // The connect failed!
         AfxMessageBox(IDP_FINGER_CONNECT_FAILED);
      }
   }
   else
   {
      delete m_pSocket;
      m_pSocket = NULL;

      // The Create failed!  Most unusual!
      AfxMessageBox(IDP_SOCKET_CREATE_FAILED);
   }
}
```

The first thing the function does is to call the **`UpdateData()`** function to perform the dialog's field exchange routine. When **`UpdateData()`** is called with the parameter value **`TRUE`**, it will copy the contents of the edit boxes into the associated member variables. Next, the function checks the contents of the member variable **`m_strHost`** to see whether the user has entered a host address. If not, we want our application to connect to a Finger server running on the user's machine. We accomplish this by setting the host address to the magic IP address 127.0.0.1. This IP address is known as a **loopback** address. It allows your machine to talk to itself as if it were both the client and the server. Moving on, we clear the contents of the field exchange variable **`m_strResponse`** and then update the display. Now we're finally ready to deal with the socket object.

If the socket object already exists, we delete it. Really, we should check its status to see whether there's a Finger already in progress. You could call the **`GetStatus()`** function, report the status to the user, then let the user make the decision about canceling the operation. Alternatively, you could disable the Finger button until the first Finger is complete.

In either case, the next step is to instantiate a new **`CFingerSocket`** object. We do so by passing the user logon name (**`m_strUser`**) and the window to notify (**`this`**) to the constructor. Then we create the socket and try to connect to the host using port 79 (the well-known Finger port). If the connect is successful, we're on our way! But remember, just because the connect returns **`TRUE`** doesn't mean that we're actually connected. Most likely, it means that the connection is in progress and that we'll receive a notification later when it finishes.

Let's take a look at our dialog's notification handler now. It's implemented as:

```
void CWinFingerDlg::OnSocketEvent (UINT nEvent, UINT nErrorCode, SOCKET hSocket)
{
   CString string;
```

```
    // Get the socket that sent this message
    CFingerSocket* pSocket = (CFingerSocket*)
                                    CFingerSocket::FromHandle(hSocket);
    ASSERT (pSocket != NULL);

    // Check the event code
    switch (nEvent)
    {
       case CFingerSocket::eventConnecting:
          string.LoadString(IDS_WF_CONNECTING);
          break;

       case CFingerSocket::eventConnect:
          if (nErrorCode == 0)
          {
             string.LoadString(IDS_WF_CONNECTED);
          }
          else
          {
             string.LoadString(IDS_NO_CONNECT);
             AfxMessageBox(string + m_strHost);
             string.LoadString(IDS_WF_DEFAULT);

             delete m_pSocket;
             m_pSocket = NULL;
          }
          break;

       case CFingerSocket::eventReceive:
          string.LoadString(IDS_WF_RECEIVING);
          break;

       case CFingerSocket::eventSend:
          string.LoadString(IDS_WF_SENDING);
          break;

       case CFingerSocket::eventDisconnect:
          string.LoadString(IDS_WF_DISCONNECTED);
          m_pSocket->GetText(m_strResponse);
          UpdateData(FALSE);

          delete m_pSocket;
          m_pSocket = NULL;
          break;

       case CFingerSocket::eventError:
          GetWindowText(string);
          break;

    }

    SetWindowText(string);
 }
```

The first thing we do is get a pointer to the socket that sent this message by calling **`FromHandle()`** to lookup the MFC object that is attached to the socket handle. This is completely unnecessary in this application because we've rigged it so that there's only one socket 'alive' at any one given point in time, that socket being **`m_pSocket`**. However, it illustrates the correct use of the **`OnSocketEvent()`** handler in a multi-socket application.

Next, we check to see which event occurred. Most are just progress messages and we respond by simply updating the caption of the window so that the user is kept informed of the progress. In response to the connect event, we check the value of **`nErrorCode`** to see whether it was successful. If it wasn't, we inform the user and delete the socket. The event of real importance to the application is the disconnect event, **`eventDisconnect`**. We respond to it by displaying the contents of the receive buffer to the user. Remember that this event is sent by the **`OnClose()`** method of the socket, indicating that the server has finished sending data and has closed the socket. This is the ideal time to display the results by updating our dialog's field exchange member variable and then performing the screen

update.

That's it! Try out the application by running the program that's included on the CD-ROM in the back of this book. You don't even need access to the Internet to test the example, since we've also supplied a Finger server. I encourage you to study the code for both the server and client and try to make a few improvements. For example, you could think about using something other than a `CString` for the receive buffer or try adding a timer to see how long a request takes and to cancel long-running requests. You can learn a lot from this simple application.

For our next project, let's leave our asynchronous socket class behind and take a look at an application that uses `CSocket` instead.

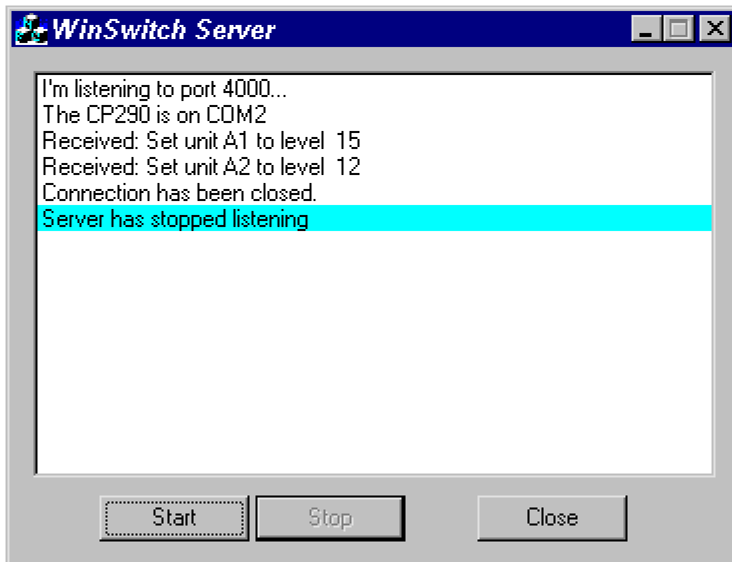
Building a Home Automation Application

The Finger client we built in the previous section uses the MFC class **`CAsyncSocket`** exclusively. **`CAsyncSocket`** is the socket base class that comes very close to coding at the 'bare-bones' WinSock API level. It provides for some abstraction above the API, but not much. **`CSocket`**, on the other hand, is a subclass of **`CAsyncSocket`** that makes socket programming even easier. It works with a couple of other classes which allow you to uses a socket like you would a file. To illustrate the use of **`CSocket`**, **`CSocketFile`** and **`CArchive`**, let's build an application to automate our homes!

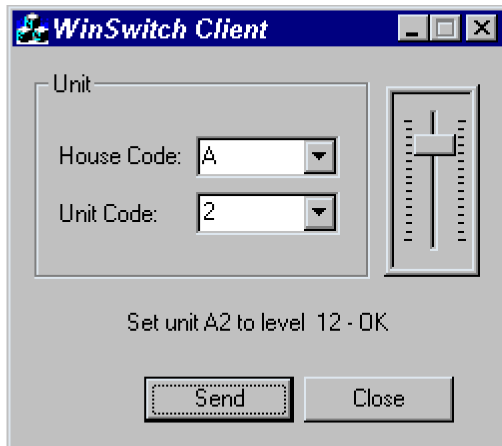
The application that we'll be constructing, `WinSwitch`, is divided into two programs. The first is a dialog-based server program that can send commands across the wiring in your house to turn appliances on and off, using simple X-10 technology. All you need is a standard serial port, a CP-290 controller and one or more X-10 modules, although you don't need them to compile and run the application.

> **For more information about Home Automation, you can visit the web page `http://www.hometeam.com`**

Here, you can see the interface for our server program:



As you can see, the server creates a socket that listens on port 4000. It also opens the serial port COM2 to try to find the CP-290 controller and then it waits patiently for clients trying to connect. When a client does connect, the server will create a new socket to communicate with the client and then it waits for the client to send a command. The commands come in the form of serialized objects. The client constructs the objects and sends them to the server through a socket with an associated **`CArchive`**. The interface for the client looks like this:



The WinSwitch Client program allows the end-user to select an X-10 module and a dimmer setting and it sends the information to the server program that does all the work of communicating with the CP-290 controller. Each module is identified by a unique House Code and Unit Code, such as A1 or C8. The client program provides two **CComboBox** controls to identify the module and a **CSliderCtrl** to set the dimmer level. When the user presses the Send button, the program instantiates a **CX10Unit** object with the module information and serializes it to the socket archive. When the server receives the object, it sends the information to the CP-290 which does all the real work. The server then sends back a message to the client in the form of a **CString** object. The client will display this message to the user when it receives it.

Let's take a look at the details!

The WinSwitch Server

The server application, WinSwitch Server, is a simple dialog-based application, since there's really no need for much of a user interface; it's a network-based server after all. To create this application, we used MFC AppWizard (exe) to create a Dialog based application and, of course, we selected Windows Sockets support from Step 2 of the AppWizard. The remaining options were left at their default settings. Next, we added the list box control shown and the Start and Stop buttons to the dialog resource.

Let's take a look at the handler for the **BN_CLICKED** event of the Start button:

```
void CWinSwitchSDlg::OnStart()
{
   // Try to listen to the port
   m_pLSocket = new CListenSocket(this);

   if (m_pLSocket->Create(WINSWITCH_PORT))
   {
      if (m_pLSocket->Listen())
      {
         // Display Startup message
         CString strMsg;
         strMsg.Format(_T("I'm listening to port %d..."), WINSWITCH_PORT);

         m_list.ResetContent();
         Display(strMsg);

         // Enable/Disable buttons
         m_btnStart.EnableWindow(FALSE);
         m_btnStop.EnableWindow(TRUE);
      }
   }
   else
```

```
    {
       delete m_pLSocket;
       m_pLSocket = NULL;
    }

    // Try to "start" the CP290 on COM2
    if (m_CP290.Open(2))
    {
       Display(_T("The CP290 is on COM2")) ;
    }
    else
    {
       Display(_T(" -> Could not find CP290 on COM2"));
    }

}
```

The first thing the function does is create a **CListenSocket** socket, **m_pLSocket**, to listen for clients wishing to connect to the WinSwitch port. After creating the listening socket, it displays a start-up message (**Display()** is a simple method that adds a string to the list box), disables the Start button and enables the Stop button. Finally, the function tries to initialize the CP-290 controller by calling its **Open()** method.

m_CP290 is a member variable of type **CCP290** which provides a high-level object for dealing with the serial port and sending commands to the CP-290 controller. The **Open()** method simply opens the serial port passed as a parameter, COM2 in this example, at 600 baud as required by the controller. I won't bore you with the details of the class **CCP290** here. The complete source code is available on the CD-ROM provided with this book. One thing, though, is to be sure that the symbol **VC_EXTRALEAN** is not defined in `StdAfx.h`. If it is, the communications functions needed to access the serial port won't be available. Let's move on and look at how the listening socket connects.

The listening socket is created as an instance variable of the class **CWinSwitchSDlg**, the main window, and it will receive **OnAccept()** notifications when a client tries to connect. Notice that the constructor of **CListenSocket** is passed the **this** pointer for the dialog. The socket will use it to keep a pointer to the window and forward events to it as they are received. Actually, the only event that it is really interested in is **OnAccept()** which is implemented in **CListenSocket** as:

```
void CListenSocket::OnAccept(int nErrorCode)
{
   // Tell the Server Window
   if (m_pWnd)
   {
      m_pWnd->OnSocketAccept(this);
   }
}
```

Pretty simple! This is the only function declared in the class **CListenSocket** and it just turns around and calls the **OnSocketAccept()** function of the main window (**m_pWnd** is the **CWinSwitchSDlg** window passed as the **this** pointer to the socket constructor).

CWinSwitchSDlg::OnSocketAccept() is implemented as:

```
void CWinSwitchSDlg::OnSocketAccept(CListenSocket* pLSocket)
{
   // Called by the listening socket when a client
   // tries to connect
   CClientSocket* pCSocket = new CClientSocket(this);

   if (pLSocket->Accept(*pCSocket))
   {
      // Initalize the connection socket (files and archives)
      pCSocket->Initialize();

      // Add the new socket to our list of active clients
```

```
            m_clients.AddTail(pCSocket);

            // Find out who we're talking to
            CString strAddr;
            UINT nPort;

            pCSocket->GetPeerName(strAddr, nPort);

            // Display IP address of client
            Display(_T("Accepted a connection from ") + strAddr);
        }
        else
        {
            // This would be strange
            Display(_T("Accept Failed!"));
            delete pCSocket;
        }
    }
```

First, the function creates another socket of type **CClientSocket**, which will be used to manage the connection. Just like the listening socket, the new socket is created with the **this** pointer passed to its constructor. This will ensure that the server receives notifications as they occur on the connected socket. Next, the **Accept()** function of the listening socket is called with a reference to our new socket passed as a parameter. The **Accept()** function will complete the connection with the client application providing our new socket as the end-point. Then, the **Initialize()** function of the new socket is called (we'll take a look at it in a minute) and the new socket is added to a **CObList** list, **m_clients**, just to keep track of it. A message is displayed to anyone bored enough to be sitting around watching our server run. Notice the use of the function **GetPeerName()**. **GetPeerName()** will tell us the IP address and port number of the client machine. This is interesting information and is displayed in the list box.

CClientSocket is the most interesting class in this entire application. It's used by both the server and the client programs to send and receive objects using the really neat **CSocketFile** and **CArchive** classes. Let's take a look at the definition of **CClientSocket**:

```
class CClientSocket : public CSocket
{
    DECLARE_DYNAMIC(CClientSocket);

// Construction
public:
    CClientSocket(CWinSwitchSDlg* pWnd);

    void Initialize();

// Unit Functions
public:
    void SendUnit(CX10Unit& unit);
    void ReceiveUnit(CX10Unit& unit);
    void SendMsg(LPCTSTR lpszMessage);
    void ReceiveMsg(CString& strMsg);

// Attributes
protected:
    CWinSwitchSDlg* m_pWnd;

    CSocketFile* m_pSocketFile;
    CArchive* m_pArchiveIn;
    CArchive* m_pArchiveOut;

// Overridable callbacks
protected:
    virtual void OnReceive(int nErrorCode);
    virtual void OnClose(int nErrorCode);

 // Implementation
public:
    virtual ~CClientSocket();
```

```
};
```

The first thing to notice is that this class is derived from **CSocket**. **CSocket** is the synchronous MFC-defined class that makes it possible to use sockets like files. **CClientSocket** will take advantage of this wonderful fact by associating itself with a **CArchive** object and serializing the objects it's asked to send and receive. Take a look at the **protected** attributes of the class and you'll see a pointer to a **CSocketFile** object and two pointers to **CArchive** objects. They are created by the function **Initialize()** and used by the member functions **SendUnit()**, **ReceiveUnit()**, **SendMsg()** and **ReceiveMsg()**. The function **Initialize()** appears below. Remember that it's called by the main window once the client socket has been created by the **OnSocketAccept()** method.

```
void CClientSocket::Initialize()
{
   m_pSocketFile = new CSocketFile(this);

   m_pArchiveIn = new CArchive(m_pSocketFile, CArchive::load);
   m_pArchiveOut = new CArchive(m_pSocketFile, CArchive::store);

}
```

The function creates a new **CSocketFile** and passes **this** to the constructor. The constructor will attach the socket to the file so that read and write operations called on the file are directed to the socket. Also, two archives (one for loading and one for storing), are created and attached to the socket file. This sets the stage for serializing objects directly to the socket! To illustrate, let's now look at the send and receive functions of our client socket class.

A server can send a textual message to a client by calling the **SendMsg()** function. It's implemented as:

```
void CClientSocket::SendMsg(LPCTSTR lpszMsg)
{
   CString strMsg(lpszMsg);

   *m_pArchiveOut << strMsg;
   m_pArchiveOut->Flush();
}
```

The function just creates a **CString** object and stores it to the archive. Also, it calls the **Flush()** function to force the buffer contents out to the socket immediately. Too easy! You can bet that the **ReceiveMsg()** function is just as easy:

```
void CClientSocket::ReceiveMsg(CString& strMsg)
{
   *m_pArchiveIn >> strMsg;
}
```

The other two send and receive functions are used to serialize **CX10Unit** objects to and from a socket. The class **CX10Unit** encapsulates the specifics about an X-10 module and it implements a standard **Serialize()** function that can be used with our socket's send and receive functions:

```
void CClientSocket::ReceiveUnit(CX10Unit& unit)
{
   unit.Serialize(*m_pArchiveIn);
}

void CClientSocket::SendUnit(CX10Unit& unit)
{
   unit.Serialize(*m_pArchiveOut);
   m_pArchiveOut->Flush();
}
```

So, the only question is, when are these functions called? To answer that, let's jump back to the **CWinServerSDlg** class (our server's main window) and look at the **OnSocketReceive()** function. This function is called by the

CClientSocket whenever it has received data that needs to be read.

```
void CWinSwitchSDlg::OnSocketReceive(CClientSocket* pCSocket)
{
   // Need to build a unit object to receove the data
   CX10Unit unit;
   pCSocket->ReceiveUnit(unit);

   // Display the thing
   m_list.SetCurSel(m_list.AddString(_T("Received: ") +
                                  unit.Description()));

   if (m_CP290.IsOpen())
   {
      // Send the unit information to the CP-290 controller
      m_CP290.SetUnit(unit);

      // Send message back to client
      pCSocket->SendMsg(unit.Description() + _T(" - OK"));
   }
   else
   {
      // Tell the client the bad news.
      pCSocket->SendMsg(_T("CP-290 is not available!"));
   }
}
```

This server function creates a local variable of type **CX10Unit** and then calls the socket's **ReceiveUnit()** function, described earlier. We display a message on the list box and then check to see whether the CP-290 controller is available to handle the request. If it is, the function passes the unit information to the controller and then sends back a textual message to the WinSwitch Client indicating the request has been passed on to the controller. If the controller is not available, a different message is sent back to the client.

And that's it! The server really does nothing more beyond passing the request off to the controller. So let's take a look now at the client application to see how it establishes a connection with the server, sends the unit information and receives messages.

The WinSwitch Client

The WinSwitch Client has one purpose: to send commands to the server program. As written, it's a simple application that allows the user to specify the address of an X-10 module, set a dimmer level using a slider control and send the request off to the WinSwitch Server. Also, it can accept textual messages sent by the server and display them to the end-user. This program uses the same **CClientSocket** class employed by the server program (with a slight modification to accommodate the new main window) and it illustrates the use of **CSocket** in a client application. When we're attempting to connect to the server, we'll see that there's a difference between using **CSocket** over **CAsyncSocket**. Let's jump right in…

The WinSwitch Client program, like the server program, is a dialog-based, AppWizard-produced application with WOSA support. We added combo box and slider controls to the dialog template, as well as a Send button. Let's begin by looking at the code for the **OnSend()** handler:

```
void CWinSwitchCDlg::OnSend()
{
   // See if we need to connect first
   if (!m_pSocket)
   {
      // Try to connect. If we can't, bail out.
      // DoConnect will display error message
      if (!DoConnect())
         return;
   }

   // We are connected. Do field exchange to
```

```
    // update the member variables.
    UpdateData(TRUE);

    // Create an X-10 Unit Object
    CX10Unit unit(m_strHouseCode.GetAt(0),
                  ::atoi(m_strUnitCode),
                  15 - m_slider.GetPos());

    // Send it!
    m_pSocket->SendUnit(unit);
}
```

To start, the function checks to see whether a socket has already been created. There's only one socket object used in the program and it remains open until either the server shuts down or the user presses the Close button. So, if the socket is already created, the function moves along and calls the dialog's **`UpdateData()`** function to transfer the values the use enters from the controls to the associated member variables. Then the function constructs a new **`CX10Unit`** object with the values selected by the user. Finally, the socket's **`SendUnit()`** function is called to serialize the object to the awaiting server.

One thing to point out here is the statement:

```
15 - m_slider.GetPos()
```

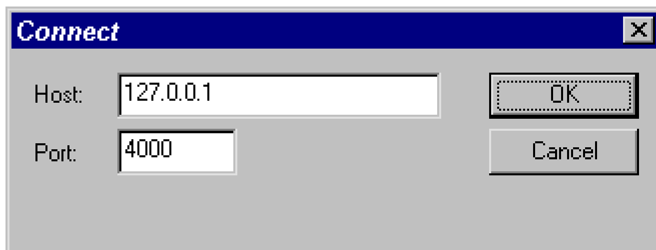
This will get the position of the slider control and invert the value so that up is down and down is up. In other words, after applying this formula, moving the slider thumb north increases the value and moving the slider thumb south decreases the value. This works because there was a previous call to the **`SetRange(0,15)`** in **`OnInitDialog()`**.

If the socket is not yet connected when this function is called, the dialog's **`DoConnect()`** method is invoked. **`DoConnect()`** will handle the dirty work of asking the user for the IP address (or domain name) of the server and the appropriate port number to use. Keep in mind that we're using the synchronous class **`CSocket`** in this program. Therefore, the connect can be handled in this way (since it doesn't return to the caller until it has completed). Had we been using the asynchronous class **`CAsyncSocket`**, we couldn't have written to the socket in **`OnSend()`**. We would have to wait until we received the **`OnConnect()`** notification. Of course, we would not be able to use an archive either! Let's look at **`DoConnect()`** next:

```
BOOL CWinSwitchCDlg::DoConnect()
{
    // Display the connect dialog
    CConnectDlg dlg(this);

    // Set default values
    dlg.m_strHost = _T("127.0.0.1");
    dlg.m_nPort = WINSWITCH_PORT;

    if (dlg.DoModal() == IDOK)
    {
        // Create the new socket
        m_pSocket = new CClientSocket(this);

        if (m_pSocket->Create())
        {
            // Try to connect using the user-entered values
            if (m_pSocket->Connect(dlg.m_strHost, dlg.m_nPort))
            {
                // Build CSocketFile and CArchive objects
                m_pSocket->Initialize();
                return TRUE;
            }
            else
            {
                // Connect Failed!
                MessageBox(_T("Could not connect to ") + dlg.m_strHost);
            }
        }
```

```
        else
        {
            // Create Failed - very unusual!
            MessageBox(_T("Could not Create() a new socket!"));
        }
    }

    // Not Conencted!
    if (m_pSocket)
        delete m_pSocket;

    m_pSocket = NULL;

    return FALSE;
}
```

As you can see, the function creates a modal dialog to get the address and port number of the server from the user. Notice the default value of **"127.0.0.1"** is set for the IP address. Remember from our discussion earlier of the Finger application that this special address is a loopback address that makes it possible to test the application without using a network. Of course, to be truly useful, the application should be used over a network and the address should default to a value read from the registry.



If the user presses the OK button, the **DoConnect()** function will try to create a socket and connect to the address and port entered by the user. Remember that **Connect()** will not return until the operation is complete (although it does pump messages in the mean time). If it's successful, the socket's initialization function is called to create the **CSocketFile** and **CArchive** objects. Control will then return to the main window's handler for the Send button and the unit information will be transmitted.

Lastly, let's examine the code that handles the receipt of textual messages from the server. Recall that the server application sends back messages after it receives and process the unit information sent by the client. The code below is all it takes to read the message and display it to the user.

```
void CWinSwitchCDlg::OnSocketReceive(CClientSocket *pSocket)
{
    pSocket->ReceiveMsg(m_strMsg);

    UpdateData(FALSE);
}
```

That's it! You can now manage the appliances in your home or office from just about anywhere. This little application actually has a lot of potential, but you might want to try a few improvements. For example, it would be nice if the client could request a list of modules from the server, sorted by name, so that the user doesn't have to specify a cryptic address. I'll leave that exercise up to you. Using **CSocket**, **CSocketFile** and **CArchive**, it's actually very simple to implement. You can serialize anything! Objects can be disassembled at one end and reassembled at the other.

Summary

The Internet is exploding! People are thinking of new applications everyday to take advantage of the fact that it's everywhere and it's inexpensive. With tools like Visual C++ and the classes provided by MFC, implementing those

new ideas is getting easier.

In this chapter, we introduced the WinSock Application Programming Interface. We focused on its asynchronous extensions, designed to take advantage of Windows' message-based architecture. We talked about the different types of sockets, datagram and stream, and how they can be used in a client/server environment. We also discussed important concepts, such as network byte ordering, IP addresses and port numbers within the context of the MFC socket classes. Finally, we developed two complete applications, demonstrating the use of all of the MFC socket classes including **CAsyncSocket**, **CSocket**, **CSocketWindow**, **CSocketFile** and **CArchive**.

Intranet Programming with WinINet

Internet-enabled Applications

Previous chapters have illustrated how Windows sockets provide a transport-independent way of programming network-based applications. Even though we had a great deal of control over how the application uses the network, we also had to do a significant amount of work to implement even a very simple network application. In a real production environment, the problem becomes even more complex. We would have had to take a hard look at other details, including access authentication, security and network failure scenarios. The problem becomes very complex very quickly if the same application designer has to consider the application requirements as well as the networking details. This is especially true if the target network is a huge, heterogeneous, unfriendly network such as the Internet.

Despite the fanfare associated with the Internet and the **Information Highway**, analysts and experts agree that **intranets** will become productive and profitable long before the actual Internet. An intranet is an isolated network, connecting computers within a corporation using existing Internet-based technologies. For example, a particular company may have an intranet consisting of a World Wide Web server at each of their departmental sites, giving information on the department's activities.

By isolating the corporate intranet from the Internet outside, a company can enjoy the benefits provided by Internet technologies without suffering the drawbacks. No worries about external security break-in attempts, leakage of proprietary secrets, etc.

Ironically, when we set about implementing real business solutions on the intranet, using existing Internet technologies, we discover just how limited these technologies really are. In particular, the building blocks that we can utilize are a handful of protocols, mostly designed in the 1970s, having changed very little since. The basic set of application level protocols includes:

File Transfer Protocol (FTP)
Gopher Protocol
Network News Transfer Protocol (NNTP)
Simple Mail Transfer Protocol (SMTP)
Hypertext Transfer Protocol (HTTP)
Telnet and Remote Login (rlogin)
Simple Network Management Protocol (SNMP)
Bootstrapping Protocol (BOOTP)
Domain Name System (DNS)
Network File System (NFS) and Remote Procedure Calls (RPC)
X Protocol
Finger
Whois
Archie
WAIS
Veronica
Ping and Traceroute

While sufficient for satisfying the information consumption demands of the general public on the Internet, these protocols fall short on delivering reliable, easy-to-use yet flexible services needed for business-oriented applications on the Intranet.

To solve more and more complex business problems with Internet-based technology, new protocols and easier-to-use development tools and programming interfaces need to be introduced. Network-based software has to be made easier to program. Only then can the designer concentrate on the complex business problem at hand, and not toil over the details of networking transport programming.

In a relentless race to become the dominant force in the future of the Internet, Microsoft and its competitors are scrambling to define flexible yet easy to use software foundations upon which to build new Internet/intranet-enabled applications. This is an exciting time. Both the independent software developer and the end user will benefit from this intense competition.

In this chapter, we'll examine the design and implementation of an intranet application. It solves a common problem found in corporate MIS: the need to distribute and collect information in a timely and accurate manner across the enterprise. We'll look at a versatile new operating system extension from Microsoft which we can use to great advantage: the Win32 **`Wininet.dll`** library, referred to as **WinINet**. Instead of working out a new protocol from scratch, we'll leverage WinINet and combine the standard FTP and Gopher protocols into a useful custom application protocol. We'll examine the API in detail and discuss the operation of the two protocols.

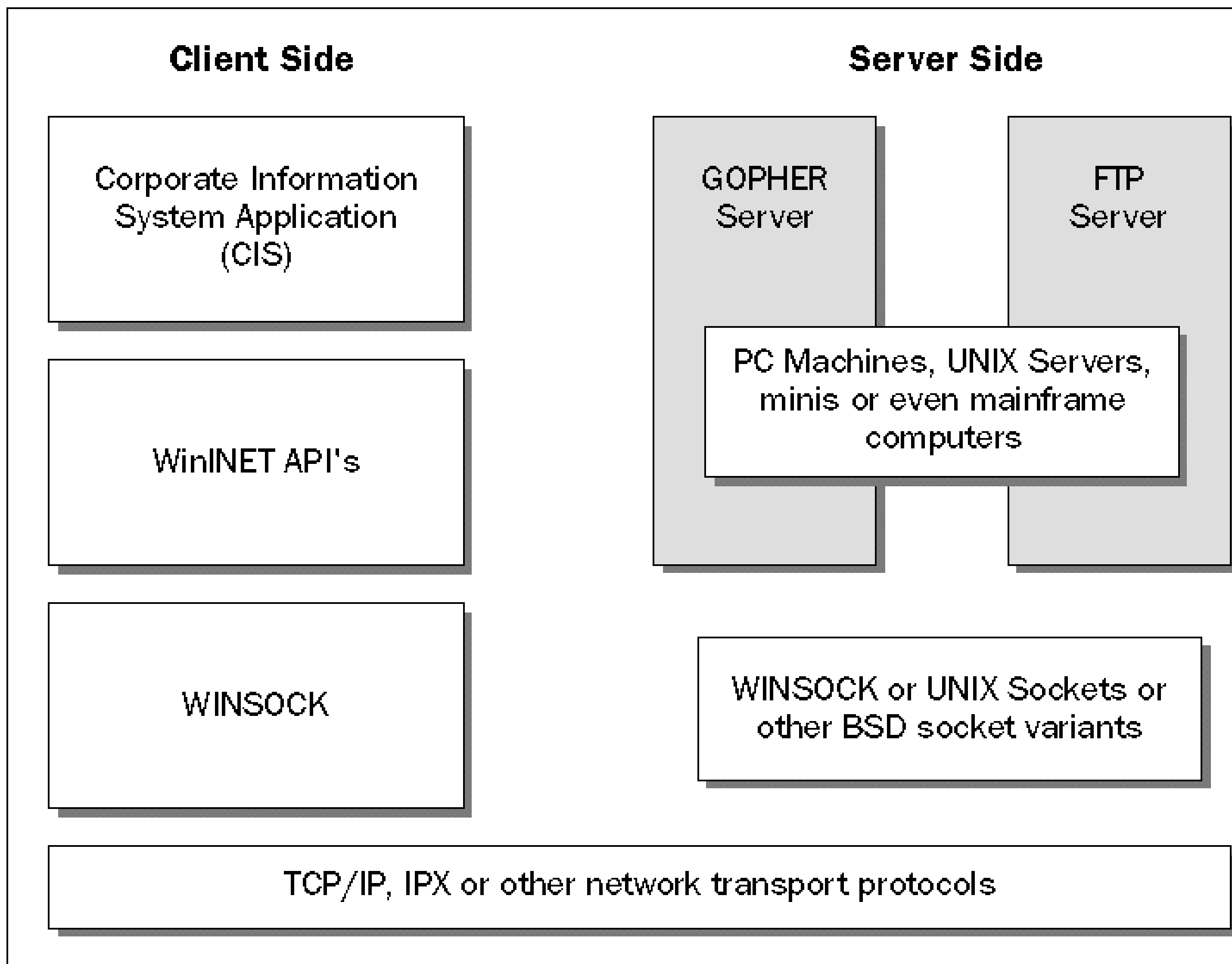
Next, we'll describe the design and implementation of our intranet 'corporate information system' using the custom protocol. This will show how you can use the WinINet APIs to implement practical business applications today. The object-oriented design will leverage Visual C++ and MFC to make programming a lot easier. Finally, we'll examine how we can set up and test the application on a network.

WinINet Technology

WinINet is actually a fundamental component of the complex Microsoft ActiveX client architecture. It's the first available component from the ActiveX suite. See Chapter 17 for more coverage of the overall ActiveX strategy.

For our purposes, we can view WinINet simply as a set of new Win32 function calls which will facilitate the implementation of our network-based applications. Microsoft has announced that the WinINet extension will be built into all new versions of Windows, and will become a part of the standard Win32 API set. This will preserve our investment in using WinINet, which, for corporate developers and managers, is often a key issue.

We'll be using WinINet to access a standard FTP and a Gopher server in a very application-specific manner. In essence, we'll be using the standard FTP and Gopher protocol to create a new protocol that only our application will understand. This will enable us to illustrate the design of a complete intranet application within the limited space of this chapter.



The figure above shows how our application will be working over two distinct layers of networking **middleware**. The WinSock library isolates higher layers from transport dependencies and the complexity in programming to different transports. The WinINet protocol in turn insulates us from the difference between implementations of FTP, Gopher and HTTP servers, and the complexity in programming the protocol details. Actually, WinINet completely shields us from having to deal with WinSock at all. Overall, what we gain is the ability to keep a sharp focus on our application problem, without having to work out many of the details related to network programming.

The WinINet API

The WinINet API is currently provided in the form of a dynamic link library (DLL). Applications which make use of WinINet will be required to link with **`Wininet.lib`**, which is provided in the ActiveX developer's kit. Future versions of Win32 SDK will include the WinINet libraries and header files as a standard component.

WinINet provides an operations-based interface to the client features of the three most popular protocols of the Internet:

File Transfer Protocol (FTP)
Gopher Protocol

Hypertext Transfer Protocol (HTTP)

Our focus will be on the FTP and Gopher support offered by WinINet. Moreover, future Microsoft products will provide a much higher level of programming access to HTTP and HTML decoding, which will practically eliminate the need for HTTP within WinINet. See the ActiveX HTML document description in the next chapter for more details.

By using the WinINet API in our project, we can:

Avoid programming directly to Windows sockets and having to learn the idiosyncrasies of TCP/IP or sockets.

Avoid having to include code which is dependent upon specific FTP, HTTP or Gopher implementations.

Avoid having to track changes or extensions in the FTP, HTTP, Gopher or their associated security or authentication standards; Microsoft will track these changes and modify the WinINet implementation while maintaining the same API for us.

WinINet supports multithreaded applications, persistent caching and Unicode API versions. We won't be exploiting these features of WinINet in our project, but we'll briefly describe them here.

Multithreaded Application Support

The WinINet API is designed to be completely thread-safe and re-entrant, which is consistent with most Win32 APIs. In a typical networking application, better overall throughput may be obtained if certain operations can be carried out in parallel.

For example, on a moderately slow connection to a network, an application that needs to open multiple connections to servers may be able to attempt those connections simultaneously, instead of serially, one at a time. In Windows 95 or Windows NT, you can do this by creating additional threads. Because it is re-entrant and thread-safe, any thread created by an application can freely call the WinINet API functions without having to worry about corrupting the internal states maintained by the WinINet run time. Internal session synchronization is also guaranteed, even in multithreaded applications.

Persistent Caching Support

To improve information access performance, the WinINet APIs can make use of a persistent cache. This cache retains images of the most recently accessed network resources. If the same resource is requested by the application, it will be fetched from the cache, instead of across the network. WinINet carries this task out through a generic helper DLL, called **`Urlcache.dll`**.

One can immediately see drawbacks in a corporate information system having hidden caching active. Imagine the problems that you might from users inadvertently retrieving stale information from the cache, thinking that it's current. Fortunately, you can disable persistent caching explicitly when WinINet API calls are made.

In general, persistent caching isn't useful for intranet-based applications. It's more applicable to Internet access. Unlike the high speed access to information sources on an intranet, the typical Internet connections are restricted to analog modems of 28.8k bps. Typical Internet users don't really mind so much if the retrieved data is a few hours old. In these cases, the caching of recently used resources will dramatically improve the perceived responsiveness of the network.

The cache maintained by WinINet is a common URL cache which can be used by other Internet applications running on the same client PC. The space for caching is allocated right from the filesystem accessible by the local machine. Don't confuse this cache with the cache often provided by caching servers (i.e. the caching web server at your Internet service provider). Some interesting Internet applications, such as off-line resource viewers, can be written using WinINet persistent cache support.

Unicode API Support

Unicode-enabled applications allow for localization in countries which may make use of very large character sets, which applies to most languages of the Far East. Rather than a single-byte ANSI encoded `char` data type, a double-byte Unicode character is used instead. Unicode is a worldwide character encoding standard, with a unified character set, representing all the characters in modern computing, used across all languages of the international marketplace

WinINet APIs which return or take parameters that contains characters, strings, or structures containing characters or strings are sensitive to Unicode encoding. There are actually two version of each API in the WinINet library: one for the ANSI character encoding and one for the Unicode character encoding. Consistent with the Win32 standard, WinINet Unicode APIs can be enabled by setting a flag during the compilation process. The WinINet APIs themselves always take `LPCTSTR`, `LPTSTR` and `TCHAR` as parameters, instead of the usual `LPCSTR`, `LPSTR` and `char` types. This enables the calling application to use either ANSI or Unicode parameters. If the calling application is also consistent in using the `LPCTSTR`, `LPTSTR` and `TCHAR` types, it can also participate in the ANSI to Unicode switch should it be necessary.

The process of enabling an application to be used in multiple countries with different languages is generally termed **internationalization**, but this encompasses many more aspects of program design and implementation than just Unicode support alone. Coverage of internationalization techniques is beyond the scope of this chapter.

The WinINet Operational Model

This sections will discuss the WinINet operational model and the specific API functions that we'll be using in the implementation of our intranet project.

WinINet provides simultaneous connections to multiple protocol servers within one single-threaded or multithreaded application. The management of multiple connections is performed through a hierarchy of **Internet handles**. When an application is started, it must initialize the WinINet libraries via the `InternetOpen()` call and obtain an application session handle from WinINet. With this handle, the application can make a new connection to a protocol server and get a new connection handle in return from the system.

Synchronous and Asynchronous Operations

WinINet supports both **synchronous** and **asynchronous** API operations.

In a synchronous operation, any API calls made during the session (between `InternetOpen()` and when the application session handle is closed) will block until:

- The operation is properly completed.
- The time elapsed has exceeded specified timeouts.
- The operation is canceled by another thread.

You should use synchronous operations for any new WinINet applications. It is also the default mode of operation for WinINet. You can manage multiple concurrent active connections with synchronous mode operations by using one thread per connection (this is one situation where using multiple threads actually greatly simplifies the programming problem and enhances performance). To ensure that the performance objective will not be compromised, always remember to place a capacity limit on the maximum number of threads that can be created within an application. The number of threads that you can create before performance deteriorates depends on the hosting hardware platform and the nature of work performed by the threads. You should determine this via careful testing on the minimal hardware requirements of your application.

We'll describe the asynchronous mode of operation in the next paragraph, but since our sample application doesn't use it, we've included this just to shed some light on the reason for many of the optional parameters in the WinINet API.

If the application session is opened with the `INTERNET_FLAG_ASYNC` flag set when `InternetOpen()` is called, the

WinINet library will attempt to service API calls asynchronously whenever possible. In this case, calls to the API will almost always return immediately. In the asynchronous mode, if an API call returns **TRUE**, the operation has completed. If the API call returns **FALSE**, the **GetLastError()** API should be called and verified against the **ERROR_IO_PENDING** value. An **ERROR_IO_PENDING** value indicates that the API operation is being completed asynchronously. Meanwhile, the application can get on with any other work. When the operation is finally completed, the status callback function will be called with the **INTERNET_STATUS_REQUEST_COMPLETE** code.

You can have several pending API calls active during asynchronous operation. If this is the case, how will the single status callback function tell which request the completion message is for? The answer lies in an optional context parameter in all APIs which supports asynchronous operation. Typically, you can use this **DWORD** context parameter to store a pointer or an index into data structures containing more information relating to the specific API calling instance. For asynchronous operations, this context parameter is mandatory for API calls.

To summarize, asynchronous mode applications must:

Call **InternetOpen()** with the **INTERNET_FLAG_ASYNC**.

Define a status callback function and register it via **InternetSetStatusCallback()**.

Always call APIs with a context parameter.

Check for an API pending situation by verifying that the return code is **FALSE** and **GetLastError()** returns **ERROR_IO_PENDING**.

Decode the context parameter when the status callback function is called with **INTERNET_STATUS_REQUEST_COMPLETE**.

It's clear that asynchronous mode applications can be quite difficult to write and test. Their style is reminiscent of Windows 3.1 communications programming and is mostly supported just for compatibility. The asynchronous mode is most suited to applications which must be single-threaded and stay responsive to the end user, although you can also use it in extreme performance and memory sensitive situations where the overhead of threading may be a concern. In demanding situations like this, though, there may be other more suitable mechanisms, such as writing directly to WinSock and implementing only the elements of the protocol used.

Error Handling

Since WinINet is an extension of the Win32 API, it reports errors in exactly the same way as other Win32 APIs. If an API fails, an application should check the error code, which can be done via the Win32 **GetLastError()** function. Specifically for FTP and Gopher operations, however, more extensive error reporting support is available via the **InternetGetLastResponseInfo()** API. This API provides more extensive error text whenever **GetLastError()** returns **ERROR_INTERNET_EXTENDED_ERROR**.

Consistent with Win32 conventions, an API typically returns **TRUE** for successful operations and **FALSE** for failed or pending operations. Some APIs return an **HINTERNET** handle, where a **NULL** return value also indicates operation failure.

In synchronous mode multithreaded operations, be aware that the **GetLastError()** and **InternetGetLastResponseInfo()** functions are guaranteed to be thread-safe. Each thread will receive the expected error information. However, you should always check the return code from **InternetGetLastResponseInfo()** to make sure that the function has succeeded.

Internet Handles

The Internet handle type, **HINTERNET**, is used throughout the WinINet functions. There is no equivalence between Win32 file handle and Internet handles, which means that you shouldn't use the Internet handles in Win32 file IO functions.

Internet handles are being used to represent different system objects in WinINet:

An application's WinINet session and its associated run-time support environment; this is returned by

the **InternetOpen()** API.

One of potentially many application connections to a remote protocol server; this is returned by the **InternetConnect()** API.

A specific 'file-find' handle returned by the **FtpFindFirstFile()**, or **GopherFindFirstFile()** APIs.

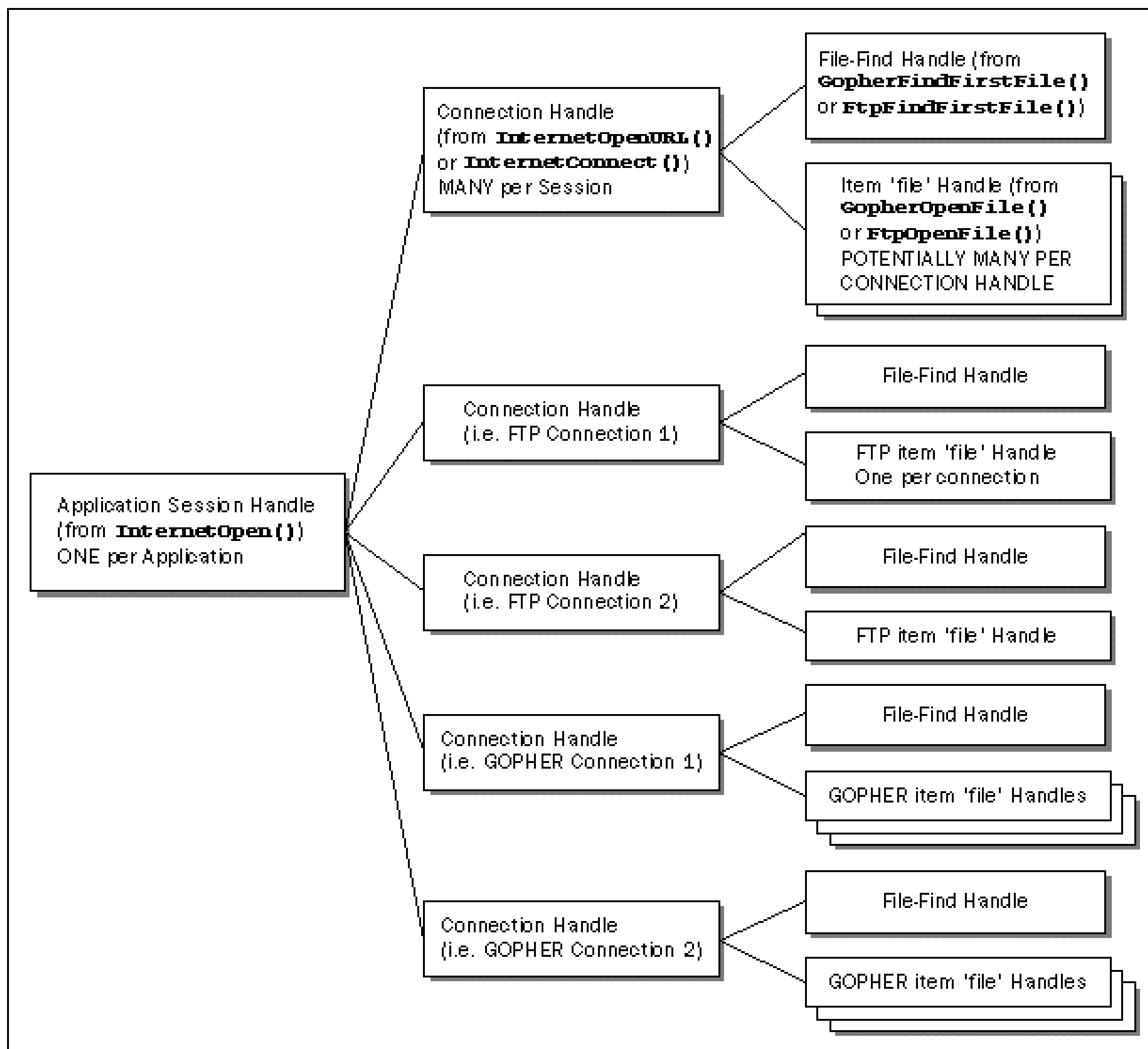
One or more 'file' or item level handles for Internet resources being manipulated through WinINet functions; this handle is returned from **FtpOpenFile()** or **GopherOpenFile()** API call.

Each application session handle can be associated with many application connection handles, and each application connection handle may be associated with a file-find handle. Each application connection can also be associated with many Internet 'file' level resource handles. It's the application's responsibility to track this hierarchy of handles and to ensure that handles are closed properly when they are no longer needed.

A typical application will manage the handles like this:

1. Call **InternetOpen()** to establish a session handle and store it.
2. Open one or more connections to FTP, Gopher or HTTP servers via **InternetOpenURL()** or **InternetConnect()**. Store one handle per connection that has been opened.
3. For each connection, examine the available 'files' or resource items via the **FtpFindFirstFile()**, **GopherFindFirstFile()** and **InternetFindNextFile()** APIs. The **FtpFindFirstFile()** and **GopherFindFirstFile()** APIs are used to obtain a 'file-find' handle, which is used by the **InternetFindNextFile()** API to enumerate the available 'files'. Close the 'file-find' handle using **InternetCloseHandle()** after the required 'file' is found.
4. Create one or more 'file' item level handles via the **FtpOpenFile()** or **GopherOpenFile()** APIs. Manipulate these 'files' via their handles. The WinINet implementation of **FtpOpenFile()** allows only one open 'file' level handle per server connection at any time.
5. Close these 'file' handles via **InternetCloseHandle()** as soon as they're no longer required.
6. Close the handle associated with a connection via **InternetCloseHandle()**.
7. After making sure all connection handles and their associated 'file-find' and 'file' item handles are closed, close the session handle from **InternetOpen()** with the **InternetCloseHandle()** call.

The next figure shows the hierarchy of **HINTERNET** handles and their relationships:



The API Functions

The set of WinINet API functions divides into four groups:

General Internet functions
FTP functions
Gopher Functions
HTTP Functions

We won't be covering the HTTP functions in this chapter.

These functions are fully documented by Microsoft on the ActiveX Developers Kit, so we won't bore you with the details here.

An Overview of General Internet Functions

The following is a description of the general Internet functions provided by WinINet, which are quite a mixed bag because they don't fit into the operation of a specific protocol. They include functions that are used to initialize WinINet or set WinINet applications: global flags and options, functions to set callback for asynchronous

operations, functions for obtaining extended error information and functions to create protocol-specific connections.

In an attempt to minimize the number of new Win32 functions, there are several common functions in the group which are used specifically in conjunction with connection level functions. These are **`InternetGetNextFile()`**, **`InternetReadFile()`**, **`InternetWriteFile()`**, which are 'factored out' of the protocol-specific set of APIs to reduce the API count.

Function Name	Description
`InternetOpen()`	Initializes the WinINet run-time support system for this application.
`InternetCloseHandle()`	Release the resources held by the Internet handle.
`InternetReadFile()`	Reads data from FTP, Gopher or HTTP source 'file'.
`InternetWriteFile()`	Writes data to an open 'file' in an FTP session.
`InternetSetStatusCallback()`	Sets a callback function to report status information.
`InternetFindNextFile()`	Continues 'file' enumeration, used in conjunction with **`FtpFindFirstFile()`** or **`GopherFindFirstFile()`**.
`InternetQueryDataAvailable()`	Queries the amount of data available in the current 'file'.
`InternetQueryOption()`	Obtain current values of various options flags.
`InternetSetOption()`	Sets a particular options flag.
`InternetGetLastResponseInfo()`	Retrieves extended error text information.
`InternetConnect()`	Opens a protocol session (FTP, HTTP, or Gopher) on the net, and logs on the user.

An Overview of FTP Functions

FTP functions are available to traverse remote server directories, enumerate remote files, add and delete files and directories or send and receive files. There is also an escape function to issue raw FTP commands to the remote server.

Function Name	Description
`FtpFindFirstFile()`	Starts enumerating files in the current directory, works in conjunction with **`InternetFindNextFile()`** function for enumeration of files.
`FtpGetFile()`	Retrieves an entire file from the server and placed it in a local file.
`FtpPutFile()`	Copies an entire file from the local file system to the server.
`FtpDeleteFile()`	Deletes a file on the server if possible.
`FtpRenameFile()`	Renames a file on the server if possible.
`FtpOpenFile()`	Opens a file on the server for either reading or writing, works in conjunction **`InternetReadFile()`** and **`InternetWriteFile()`**.
`FtpCreateDirectory()`	Creates a new directory on the server if possible.
`FtpRemoveDirectory()`	Deletes a directory on the server if possible.
`FtpSetCurrentDirectory()`	Changes the client's current directory on the server.
`FtpGetCurrentDirectory()`	Returns the client's current directory on the server.
`FtpCommand()`	Issues an raw FTP command to the server.

An Overview of Gopher Functions

Gopher functions are available to traverse server menus and request transfer to remote locators. Support functions are also available to create a locator from scratch, as well as deciphering the attributes of a Gopher object.

Function Name	Description
`GopherFindFirstFile()`	Starts enumerating a Gopher directory listing.
`GopherOpenFile()`	Starts retrieving a Gopher object.
`GopherCreateLocator()`	Forms a Gopher locator for use in other Gopher function calls.
`GopherGetAttribute()`	Retrieves attribute information on the Gopher object.

The FTP and Gopher Protocols

In a nutshell, FTP is a protocol which allows networked computers to exchange files. The protocol takes care of differences in dissimilar computers running different operating systems.

The Gopher protocol, on the other hand, provides a text-based menu-driven way to access a group of network resources. Most of the interaction within a Gopher session involves navigating and traversing hierarchically linked menu entries.

We'll now go on to describe the two protocols, both of which work on top of TCP/IP, but before we do, we need to clarify several basic concepts of TCP/IP. TCP/IP provides services delivering packets between two computers representing two end-points on the network. A TCP connection is reliable because packets sent through a TCP connection are guaranteed to arrive at the other end in the same sequence that they were sent, and without being corrupted. Other protocols within the TCP/IP suite don't provide such reliable delivery, nor do they guarantee sequencing.

The FTP Protocol

The FTP protocol is a TCP protocol and makes only TCP connections between two computers.

This is what happens during an FTP session:

1. The FTP server listens on a well-known TCP port (usually 21) for requests; this is done via a **passive open** on the port.
2. The FTP client initiates an active open on the server's TCP port and a TCP connection is established between the client and server; this is the control connection used to pass the client's command to the server and the server's replies to the client.
3. A data connection, separate from the control connection, is made each time the client and server agree to transfer a file; this data connection is made and destroyed for each file transfer.

All commands sent through the control connection are in ASCII.

The data connection is established by the client sending a `PORT` command to the server. After making a passive open (open a port for the purpose of accepting TCP connections) to an available port, the client sends a `PORT` command to the server with the available port number. Once the server has received this over the control connection, it will make an active connection to the client's available port when necessary. This will create the connection to send the data sent.

One the server has successfully received the `PORT` command, the client can request a file retrieval via the `RETR` command. The server will then open the data connection, send the file and close the data connection to signify the end of transmission.

If the client sends a `STOR` command instead, the server will open the data connection ready to receive the file. The

client should close the data connection at the end of transmission to signify the end of transmission.

Typically, FTP applications will hide the details of the protocol from the end user, greeting her instead with a graphical user interface, or much higher level text commands.

However, even a cursory understanding of what goes on underneath will give the user give some clues to what's happening and to some of the configurable options on a highly configurable library like WinINet.

The Gopher Protocol

The Gopher protocol is a TCP protocol which makes only TCP connections between two computers.

The Gopher protocol is less complex than the FTP protocol. Unlike FTP, it has no concept of a session, and connections don't stay up for any length of time, just long enough to transfer the requested information. Instead, the following happens:

1. The Gopher server listens on a well known TCP port (usually 70) for requests, via a passive open on the port.
2. The Gopher client initiates an active open on the server's TCP port and a TCP connection is established between the client and server. The client immediately sends a **retrieval string** to the server.
3. The Gopher server sends all it has to say about the retrieval string to the client and closes the connection.

This is repeated for every Gopher request. Both the retrieval string and the information from the server are always in ASCII.

The information from the server is usually a series of lines in ASCII, ending with a carriage return and line feed character. The last line sent by the server before closing the connection is a single period (`.`).

If the Gopher protocol is so simple, how does it work? The magic lies in the retrieval string sent to the server and the information returned from the server. Each line returned from the server before the period is a **locator**. (You'll see a lot more about this later.) The locator is in ASCII and consists of the following fields, separated by tab characters (except for the **Gopher type code**, which is always the first character):

- A Gopher type code
- The friendly name of the item
- The **selector**, or retrieval string
- The hostname of the computer containing the item
- The port number of the service

The Gopher type code indicates what the item is and will be tabulated later. Two typical values are text files (0) or directories (1), but the defined list is quite comprehensive. The friendly name is a human readable description of the item content. The selector is the retrieval string to be sent to the remote host in another Gopher transaction. The hostname and port number tell the Gopher client how to reach the next server.

By examining the type code, the Gopher client knows what to do with the rest of the fields, and what to expect when the host server is contacted via the selector. For example, if the type indicates a binary file, the next Gopher request to the selector will return the binary file and the Gopher client should receive every byte until the server closes the connection. It shouldn't attempt to parse carriage return and line feed characters.

Our Protocol

By combining the power of Gopher in presenting a complex hierarchy of information, and the two way data transfer capability of the FTP protocol, we create our own protocol for CIS.

We need to tell the CIS application what picture and text to display on the button, and, if it corresponds to a form, which form to pop up. We can do this by modify Gopher non-intrusively, without extending it or changing the basic protocol (this is done so that regular Gopher clients can be used to access the CIS server site for testing). We simply attaching extra information to the item friendly name and using the friendly name itself for the button label. For example, for a button that says Welcome to Hawaii! and is used to display bitmap number 4 and form number 2, the item friendly name will be: **`Welcome to Hawaii!(4,2)`**. The information inside the bracket will be stripped before the string is used as the button label.

We use FTP as it's only for forms submission, since Gopher doesn't provide a mechanism for sending information from the client to the server. When a form is to be submitted, CIS will collapse all the information on the form, together with the time, date and ID of the button pressed and send the entire collection via an FTP-based put file command. The server must be set to allow anonymous logins to have write permission. Data from the forms will always be deposited into the default root directory of the FTP server, and the forms will have unique names because each filename will be keyed by user name, as well as a unique integer based on the client system time.

A Corporate Information System

The competitive business environment has driven most corporate MIS departments to re-evaluate how they collect and distribute their information. Most of the inherited, heavy process, paper-based systems are being streamlined and/or replaced by faster automated electronic systems. Slow internal publishing processes are being made redundant by online systems which distribute information accurately and promptly. The conventional routing of paper forms via internal mail is quickly being displaced by electronic forms and custom workflow applications.

Large corporations are increasingly being pushed to develop systems to distribute information to their remote branch offices, remote employees, affiliates and even customers. Whether or not a business has the competitive edge often comes down to how quickly and reliably it can send out accurate information to interested parties.

For corporate MIS developers, systems based on existing database and file access technology can often be sufficient for small group data sharing. To implement a corporate wide intranet-based system, however, these conventional means of sharing/distributing information often either don't fit the mark, or become too complex to administer.

Thanks to the maturity of Internet technologies, you fulfill rapid information dissemination requirements of your application. To illustrate the point, we will implement the framework for one such system. The following is a hypothetical 'mini feature specification' for the CIS project. Feature or User Requirement Specifications are standard requirement in almost all corporate development projects.

"CIS must provide the user with a non-intimidating, easy to use interface for navigating and accessing corporate information resources. Current corporate users are most familiar with the 'windows and push-button' oriented interface of Microsoft Windows and X-Windows workstations.

CIS must allow users to easily navigate the corporate network to find and view the information they need. It must also provide a mechanism for collecting information from the user. This mechanism must be totally customizable. Initially, it should support customizable forms.

CIS should be designed to use standard protocols, thus avoiding the expense involved in developing custom server software.

CIS must allow individual departments within our enterprise to publish and provide their own information resources at their own site over the worldwide corporate TCP/IP network. The administration of server sites must be simple enough for the current pool of UNIX server administrators, as well as the new Windows NT server administrators, to handle.

CIS must allow for the user interface and data access to be customized and localized at each of the individual branches according to local requirements.

CIS must provide an upward compatibility path should corporate MIS decide to adopt an alternative information sharing infrastructure. This is especially true for the World Wide Web based technologies currently being investigated."

If all this sounds like a tall order, it is! However, it will probably ring a familiar bell for most contemporary consultants and analysts working in an MIS or downsizing environment. These are very frequent and real requirements!

'Why don't we just give them the Web?' is one question that immediately comes to mind. That strategy has at least three problems, the first of which is the issue ofuser friendliness. Until everyone in the corporate structure has become familiar with the Internet, the complex user interface of most current day browsers can be quite daunting. While most will know how to push buttons, open windows or fill out an e-form, they will find it difficult to comprehend what a persistent cache, Java console, or mime-encoded document is.

The second problem involves the handling of error conditions. As we have control of the source code we can avoid the critic messages like 'object not found error 1.07e?', 'Server does not have a DNS entry' or 'URL has moved' that Internet browsers tend to report. Instead, we can give user friendly messages which also have some meaning for the support people down the corridor.

The third problem is a strange one, involving something rather intangible. For some reason, probably due to the phenomenal success of the Internet and the myth and hype surrounding it, many users associate web browsers with security problems. They tend to place more trust in rigid custom applications developed in-house than they do in having the exact same information presented, often more attractively, on web pages. With the announcement of new technologies, such as ActiveX, the landscape here may change with time. ActiveX will blur the distinction between what we know as the 'desktop metaphor' and the Internet web browser. ActiveX will also provide many more opportunities for an application to exert control over the presentation of Internet based information to the user. The accelerated evolution towards seamless integration of the Internet with the desktop will greatly benefit those developing intranet applications.

Designing the CIS

All right, it's time to bring out the drawing board and design our system. To comply with all the requirements, we've decided to do custom object-oriented development using Visual C++ and MFC. We'll use WinINet to handle the network communications required for the project. Imagine the following hypothetical high-level or product design specification:

"CIS will provide the user with a simple push button interface to navigate corporate information resources.

CIS will allow the user to navigate the corporate network by pushing a panel of buttons. Users will be able to locate the information they need without having to use additional tools or changing user interface. CIS will also let them customize the forms. Forms can be designed using the resource editor and ClassWizard tools provided with standard Microsoft Visual C++.

CIS will make use of the standard, well tested, FTP and Gopher protocols from the Internet. Standard FTP and Gopher servers can be set up for servicing the CIS application. No server- based development will be required for CIS operations.

CIS will allow individual departments to manage and publish their own information resources. It will be easy for system administrators familiar with maintaining UNIX FTP and Gopher to administer the CIS hosts.

The CIS user interface is extensible through custom C++ programming. The basic structure of CIS will allow the addition of Unicode support for branches with foreign language requirements.

CIS will provide an upward migration path towards integration with the World Wide Web, should it be adopted to

distribute corporate wide information in the near future. The CIS application will support a seamless transition between itself and standard web browsers in a subsequent release."

We have tried to satisfy every one of the feature/user requirements in our design of the CIS.

CIS will present the user with a panel of push buttons. Each push button will contain a picture (bitmap) and a description of the information that the user will see if they click it or a description of a new navigation point. When the user clicks a button, CIS will either:

Present the user with a new panel of push buttons to select from.

Display a text file of information.

Display a custom electronic form for the user to fill out.

As it processes the request that the user makes through the push button, CIS may transparently transfer the user across different servers on the corporate network.

The user interface and custom forms will be coded based on MFC. WinINet and the standard Gopher protocol will allow the user to navigate through the information. Specifically, we'll be making extensive use of Gopher's hierarchical menu, its ability to transparently link across multiple servers in one request, and its ability to serve ASCII text files.

Since the Gopher protocol is designed only to handle navigation and information read requests, we need an alternative mechanism to implement the submission of electronic forms. WinINet's support for the FTP protocol fits the bill perfectly. The data from the forms will be transferred to a file on a remote FTP server using WinINet FTP for manual or automated processing.

To begin the actual design of the CIS, we'll proceed to:

- Wrap WinINet in a hierarchy of C++ classes.
- Design and implement the required operation support data structures according to the standard MFC document/view model.
- Design and implement the required user interface, including a new button class to handle our custom requirements.
- Design and implement the linkage between the WinINet back end, user interface and internal data structures.

Wrapping WinINet with C++

Let's add a wrapper around the WinINet extensions to make it easier to call from C++. Since MFC will eventually offer a complete wrapping for the WinINet APIs in subsequent Visual C++ versions, we'll try to do an adequate job at hand, but not attempt to solve all the generic problems.

In wrapping WinINet, we'll try to:

Hide the Internet handles hierarchy as much as possible and provide automatic close when necessary.

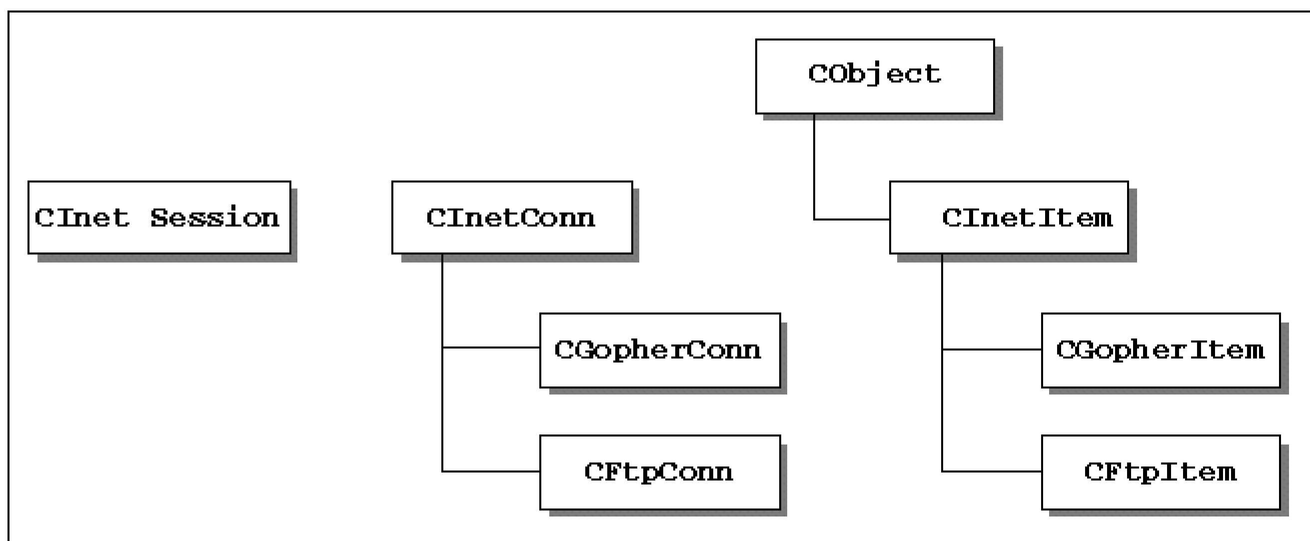
Provide frequently used, default parameters to the WinINet APIs, making the calling code easier to write, simpler to read and maintain.

Provide useful data object classes for our CIS application.

Our basic WinINet wrapping includes the following seven classes:

`CInetSession`	**`CInetConn`**	**`CFTPConn`**
`CGopherConn`	**`CInetItem`**	**`CFTPItem`**
`CGopherItem`		

The next figure shows the derivation relationship between these classes:



In the following pages, we'll describe the design rationale for each of the classes and how we intend them to be used. We'll also examine the interrelationships between the classes.

CInetSession Class

The **`CInetSession`** class represents a complete Internet session. It hides the global Internet session handle required when the WinINet run time is initialized for a specific application. Member functions provided actually wrap many of the general functions. These include **`Get`**/**`Set`** functions for global options, and the establishment of application sessions/connections.

The member functions of the **`CInetSession`** class do the following:

Initialize and take down WinINet, via **`Open()`** and **`Close()`** members.

Create protocol connections, via **`Connect()`**, **`ConnectGopher()`**, and **`ConnectFtp()`** members.

Configure WinINet protocol parameters, via **`SetConnectionInfo()`**,

SetConnectionOptions(), and **SetDataOptions()**.
Track outstanding asynchronous operations, via **IncAsyncCount(), DecAsyncCount()** and **GetAsyncCount()**.
Maintain the internal 'open connection' list, via **DeleteConn()**.

Before any WinINet operations can be invoked, the application must call the **Open()** member to initialize WinINet through the application's **CInetSession** object.

CInetSession works like a protocol connections factory. You instantiate it once to initialize the WinINet environment, then you call the connection functions repeatedly, once for each of the protocol connections you require. The connection functions actually create new **CInetConn**, **CGopherConn** or **CFTPConn** objects with its **Connect()**, **ConnectGopher()** and **ConnectFtp()** member functions. (We'll describe these classes later.) The pointer to each newly created **CInetConn, CGopherConn** or **CFTPConn** object is stored in a private **m_OpenConn** list, to track all the currently opened connections in this session. The **close()** operation uses the **m_OpenConn** list to close up any connections which may still be open.

In asynchronous mode, the **IncAsyncCount()**, **DecAsyncCount()** and **GetAsyncCount()** functions help to track the pending operation. These are simple functions which manipulate an internal counter. You should use **IncAsyncCount()** each time an asynchronous operation is submitted, and call the **DecAsyncCount()** within the status callback function each time an asynchronous operation is completed. This will allow the **CInetSession** class to determine whether any outstanding asynchronous operations are still pending, before allowing the WinINet session to close.

CInetSession Member Functions

The following table list and describes each member of the **CInetSession** class:

Member Function	Description
CInetSession()	A constructor which initializes the handle.
~CInetSession()	A destructor which closes the handle if necessary.
Close()	Explicitly closes the Internet session handle.
Connect()	A general function to connect to a remote protocol server and log on the user.
ConnectFTP()	Connects to an FTP remote protocol server and logs on the user.
ConnectGopher()	Connects to a Gopher remote server and logs on the user.
Open()	Opens the Internet session.
SetConnectionInfo()	Sets time-out and retry values for the Internet connection.
SetConnectionOptions()	Sets control channel time-outs associated with FTP.
SetDataOptions()	Sets data connection time-outs and retries.
IncAsyncCount()	Increments the outstanding asynchronous operations counter.
DecAsyncCount()	Decrements the outstanding asynchronous operations counter.
GetAsyncCount()	Retrieves the current asynchronous operations count.
DeleteConn()	Deletes the connection specified from the connections tracking list.

CInetSession::Open()

Once you have created an instance of this class, you must first open an Internet session by calling **Open()**. The definition is as follows:

```
BOOL CInetSession::Open(LPCTSTR Agent, UINT Flags,
```

```
  UINT AccessType , LPCTSTR ProxyName , LPCSTR ProxyBypass,
   INTERNET_STATUS_CALLBACK lpfnNetCallBack)
{
    if((m_hSession = InternetOpen(Agent,
        AccessType,
        ProxyName,
        ProxyBypass,
        Flags
        )) == NULL)
        return FALSE;
    if (NULL != lpfnNetCallBack)
         InternetSetStatusCallback(m_hSession, lpfnNetCallBack);
    return TRUE;
}
```

As you can see, this member is simply a wrapper for the **InternetOpen()** API from WinINet. We simply pass on the parameters passed to **Open()** to **InternetOpen()**. **InternetOpen()** returns a **HINTERNET** handle if successful, or **NULL** otherwise, which we store in **CInetSession::m_hSession** as we'll need this handle in the other methods. To save work for people using asynchronous mode, the status callback function pointer can be passed in during a call to **Open()**. If this is supplied, the member function will make a call to **InternetSetStatusCallback()** to register the callback function with WinINet.

The **Agent** parameter is a string, with the name of your application. **Accesstype** can be:

Flag	Meaning
INTERNET_OPEN_TYPE_PRECONFIG	Preconfigured (through the registry).
INTERNET_OPEN_TYPE_DIRECT	Direct to Internet.
INTERNET_OPEN_TYPE_PROXY	Through CERN proxy.

For most intranet applications, use **INTERNET_OPEN_TYPE_DIRECT**. For Internet applications, with a CERN security proxy between the client station and the net, use **INTERNET_OPEN_TYPE_PROXY** and supply the proxy name in the **ProxyName** parameter. You can use **ProxyBypass** to supply a list of proxy-bypass hostnames, i.e. access to this list of hosts will not go through the proxy.

ProxyName and **ProxyBypass** can both be set to **NULL** for intranet applications.

The **Flags** parameter can be set to **INTERNET_FLAG_OFFLINE** or **INTERNET_FLAG_ASYNC**. The **INTERNET_FLAG_OFFLINE** is used mainly for Internet applications and indicates that all access to Internet resources should be from the persistent cache only.

INTERNET_FLAG_ASYNC is the flag which indicates to WinINet that all connections and APIs from this application should be made in the asynchronous mode. For most intranet applications, including our CIS, using **0** for **Flags** is sufficient.

CInetSession::Connect()

After opening a session, you will need to make a connection to the service that you want to use. For this, we have provided three methods: **Connect()**, **ConnectFTP()** and **ConnectGopher()**. Each of these use the **InternetConnect()** API, the only difference being the return type. **Connect()** allows connection to either FTP, Gopher or HTTP servers, returning a pointer to a **CInetConn** object. **ConnectFTP()** obviously connects to FTP services returning a pointer to a **CFTPConn** object, while **ConnectGopher()** connects to Gopher services, returning a pointer to a **CGopherConn** object.

The definition of **Connect()** is:

```
CInetConn * CInetSession::Connect(LPCTSTR ServerName,
```

```
    UINT Service, UINT Flags, INTERNET_PORT ServerPort,
    LPCTSTR UserName,LPCTSTR Password, UINT Context)
{
    HINTERNET hConnect;
    ASSERT(m_hSession != NULL);

    if((hConnect = InternetConnect(
        m_hSession,
        ServerName,
        ServerPort,
        UserName,
        Password,
        Service,
        Flags,
        Context
        )) == NULL)
        return ((CInetConn *) NULL);
    else
    {
        CInetConn * ab = new CInetConn(hConnect, this);
        if (NULL != ab)
            m_OpenConn.AddHead(ab);
        return ab;
    }
}
```

After making sure that we have a valid session handle, we call the **InternetConnect()** API, passing along the parameters. If **InternetConnect()** returns a valid **HINTERNET** handle, we create a new **CInetConn** object, passing the handle and a class pointer to the constructor. The newly created **CInetConn** class will use the handle for its operations, and will use the class pointer to access **CInetConn::DeleteConn()** to manage open connections.

The parameters we pass to **InternetConnect()** are as follows.

We use the handle from **InternetOpen()** stored in **m_hSession** for the first parameter.

The **ServerName** and **ServerPort** should contain the hostname and port number for the connection. The hostname can also be entered as an IP address. You need not specify the **ServerPort** (i.e. just specify **0**) if the 'well-known' port for a service is used.

The **UserName** and **Password** is used for FTP session login. Set them to **NULL** for other services. If you set them to **NULL** for FTP, you'll automatically get anonymous login using the user's e-mail address from the registry.

Service specifies the type of service that you are connecting to and can be either **INTERNET_SERVICE_FTP**, **INTERNET_SERVICE_Gopher**, or **INTERNET_SERVICE_HTTP**.

The only valid **Flags** parameter is **INTERNET_CONNECT_FLAG_PASSIVE** for FTP service. This specifies that passive mode should be used for all FTP connections.

Context is the context parameter passed to the callback function if one is registered. This is usually used in the asynchronous mode of API access.

CInetSession::ConnectGopher() and CInetSession::ConnectFTP()

The **ConnectGopher()** and **ConnectFTP()** methods are similar to **Connect()**. We simply fill in some of the parameters to **InternetConnect()** with default parameters, and return pointers to different classes:

```
CFTPConn * CInetSession::ConnectFTP(LPCTSTR ServerName,
    LPCTSTR UserName, LPCTSTR Password, UINT Flags,
    UINT Context)
{
    HINTERNET hConnect;
    ASSERT(m_hSession != NULL);
```

```
    if((hConnect = InternetConnect(
        m_hSession,
        ServerName,
        0,
        UserName,
        Password,
        INTERNET_SERVICE_FTP,
        Flags,
        Context
        )) == NULL)
        return ((CFTPConn *) NULL);
    else
    {
        CFTPConn * ab =  new CFTPConn(hConnect, this);
        if (NULL != ab)
            m_OpenConn.AddHead(ab);

        return ab;
    }
}

CGopherConn * CInetSession::ConnectGopher(LPCTSTR ServerName,
UINT Context)
{
    HINTERNET hConnect;
    ASSERT(m_hSession != NULL);

    if((hConnect = InternetConnect(
        m_hSession,
        ServerName,
        0,
        NULL,
        NULL,
        INTERNET_SERVICE_GOPHER,
        0,
        Context
        )) == NULL)
        return ((CGopherConn *) NULL);
    else
    {
        CGopherConn * ab = new CGopherConn(hConnect, this);
        if (NULL != ab)
            m_OpenConn.AddHead(ab);

        return ab;
    }
}
```

CInetSession::Close()

```
BOOL CInetSession::Close()
{
    if (m_asyncCount == 0)
    {
        // close all remaining open connections
        while(! m_OpenConn.IsEmpty())
        {
            m_OpenConn.GetHead()->CloseConnection();
            m_OpenConn.RemoveHead();
        }
        if(NULL != m_hSession)
            InternetCloseHandle(m_hSession);
        return TRUE;
    }
    else
        return FALSE;
}
```

This is the function used to close the WinINet session. If asynchronous mode is used, the **Close()** will fail if any tracked asynchronous operations are still pending. Otherwise, this function will check the list of currently opened connections and close anything else that is still open. Finally, it will call **InternetCloseHandle()** on the session handle.

The **InternetCloseHandle()** function is a catchall for closing any WinINet handles. This applies to the application session handle, any protocol connection handles, or FTP or Gopher 'file' handles.

CInetSession::SetConnectionInfo()

```
BOOL CInetSession::SetConnectInfo(UINT uiTimeout , UINT uiRetries, UINT uiBackoff)
{
    if(InternetSetOption(m_hSession, INTERNET_OPTION_CONNECT_TIMEOUT,
        &uiTimeout, sizeof(UINT)) == FALSE)
        return FALSE;

    if(InternetSetOption(m_hSession, INTERNET_OPTION_CONNECT_RETRIES,
        &uiRetries, sizeof(UINT)) == FALSE)
        return FALSE;

    return(InternetSetOption(m_hSession, INTERNET_OPTION_CONNECT_BACKOFF,
        &uiBackoff, sizeof(UINT)));
}
```

CInetSessionSetConnectInfo() sets the connect timeout in milliseconds, the number of connect retries, and the connect backoff timing values when an Internet connection is attempted. The member functions simplifies the required calls to the **InternetSetOption()** API by grouping the related values together and making all the calls within the same function.

The **InternetSetOptions()** API is used to set a specific Internet option (the second parameter) with the value pointed to by the third parameter for the session corresponding to the handle specified in **m_hSession**.

We also need to pass the size of the parameter that we're passing to **InternetSetOption()**,which, in our case, is the size of an **UINT**.

Although we only set the connection options (**INTERNET_OPTION_CONNECT_TIMEOUT**, **INTERNET_OPTION_CONNECT_RETRIES** and **INTERNET_OPTION_CONNECT_BACKOFF**), other options which can be set currently includes:

Option	Meaning
INTERNET_OPTION_CONNECT_TIMEOUT	The time-out in milliseconds when attempting an Internet connection.
INTERNET_OPTION_CONNECT_RETRIES	The maximum number of retries in attempt to make Internet connection.
INTERNET_OPTION_CONNECT_BACKOFF	The number of milliseconds to wait between retries.
INTERNET_OPTION_CONTROL_SEND_TIMEOUT	The time-out in milliseconds when trying to send over the control channel in FTP sessions.
INTERNET_OPTION_CONTROL_RECEIVE_TIMEOUT	The time-out in milliseconds when waiting to receive over the control channel in FTP.
INTERNET_OPTION_DATA_SEND_TIMEOUT	The time-out in milliseconds when trying to send data.
INTERNET_OPTION_DATA_RECEIVE_TIMEOUT	The time-out in milliseconds when trying to receive data.
INTERNET_OPTION_ASYNC_PRIORITY	Sets the priority of this download if it's an asynchronous download.
INTERNET_OPTION_CONTEXT_VALUE	Sets the context value associated with this Internet

	handle.
INTERNET_OPTION_USERNAME	Sets the user name associated with a handle returned by the InternetConnect API.
INTERNET_OPTION_PASSWORD	Sets the password associated with the handle returned by InternetConnect API
INTERNET_OPTION_READ_BUFFER_SIZE	Sets the size of the read buffer for **FtpGetFile()**.
INTERNET_OPTION_WRITE_BUFFER_SIZE	Sets the size of the write buffer for **FtpPutFile()**.

CInetSession::SetControlOptions()

To set timing options specific to the FTP control channel, we again use similar wrapping as **CInetSession::SetConnectionOptions()** in **CInetSession::SetControlOptions()**.

```
BOOL CInetSession::SetControlOptions(UINT uiSendTimeout, UINT uiRecvTimeout)
{
    if(InternetSetOption(m_hSession, INTERNET_OPTION_CONTROL_SEND_TIMEOUT,
        &uiSendTimeout, sizeof(UINT)) == FALSE)
        return FALSE;

    return(InternetSetOption(m_hSession,
        INTERNET_OPTION_CONTROL_RECEIVE_TIMEOUT,
        &uiRecvTimeout, sizeof(UINT)));
}
```

CInetSession::SetDataOptions()

Again, we combine the send timeout value and the receive timeout value for the FTP data channel, to be set by the **CInetSession::SetDataOptions()** function. The construction is identical to earlier **CInetSession::SetxxxOptions()** member functions. The more obscure options which are not covered by the trio of **CInetSession::SetConnectionOptions()**, **CInetSession::SetControlOptions()**, and **CInetSession::SetDataOptions()** must be changed explicitly using **InternetSetOption()** API calls if necessary.

```
BOOL CInetSession::SetDataOptions(UINT uiSendTimeout, UINT uiRecvTimeout)
{
    if(InternetSetOption(m_hSession, INTERNET_OPTION_DATA_SEND_TIMEOUT,
        &uiSendTimeout, sizeof(UINT)) == FALSE)
        return FALSE;

    return(InternetSetOption(m_hSession,
        INTERNET_OPTION_DATA_RECEIVE_TIMEOUT,
        &uiRecvTimeout, sizeof(UINT)));
}
```

CInetSession::IncAsyncCount() and CInetSession::DecAsyncCount()

```
UINT CInetSession::IncAsyncCount()
{
    return (++m_asyncCount);
}

UINT CInetSession::DecAsyncCount()
{
    m_asyncCount--;
    m_asyncCount = (m_asyncCount < 0)? 0:m_asyncCount;
    return m_asyncCount;
}
```

These two functions increment and decrement the outstanding asynchronous operations counter maintained by the **CInetSession** class. They are supplied to help the user to implement asynchronous mode applications. The **IncAsyncCount()** member should be used each time an asynchronous operation is submitted, and the **DecAsyncCount()** member should be called within the status callback function when an asynchronous operation

has completed. Their implementations are trivial. Strictly speaking, the counter **m_asyncCount** should be protected via some synchronization mechanism (i.e. a semaphore), since **DecAsyncCount()** may be simultaneously called from a separate thread context as an **IncAsyncCount()**. However, the increment and decrement operation is essentially atomic on 80x86 based machines.

CInetSession::GetAsyncCount()

```
UINT CInetSession::GetAsyncCount()
{
 return m_asyncCount;
}
```

This function simply provide access to the current value of the outstanding asynchronous operations counter.

CInetSession::DeleteConn()

```
BOOL CInetSession::DeleteConn(CInetConn * myCon)
{
    POSITION aPos = m_OpenConn.Find(myCon);
    if (NULL == aPos)
        return FALSE;
    else
        m_OpenConn.RemoveAt(aPos);
    return TRUE;
}
```

CInetSession maintains a list of new connections which it creates. Each member contains a pointer to the corresponding **CInetConn** derived object. The list is declared as a templated type-safe list:

```
protected:
    CTypedPtrList<CPtrList, CInetConn *> m_OpenConn;
```

An element is added to this list every time **CInetSession::Connect()**, **CInetSession::ConnectGopher()**, or **CInetSessionConnectFTP()** succeeds. An element is removed from the list through a call to the **CInetSession::DeleteConn()** each time a previously created connection is closed or deleted. The **DeleteConn()** function simply locates the supplied **CInetConn** pointer parameter in the **m_OpenConn** list. If found, it will delete the element from the list.

CInetConn Class

The **CInetConn** is a base class for different application sessions, which we'll call **protocol connections**, to differentiate them from the Internet session established between WinINet and the application.

Although this class is used as a base class for **CGopherConn** and **CFTPConn**, we can still use it for a generic connection to a service (i.e. when we use **CInetSession::Connect()**). In particular, this class can be used to access arbitrary URL based resources using the **OpenUrl()**, **ReadFile()** and **CloseFile()** functions.

To use this class for reading URLs, you need to:

1. Create a **CInetConn** object using the **CInetSession::Connect()** function.
2. Call **CInetConn::OpenUrl()** with the required URL.
3. Keep calling **CInetConn::ReadFile()** until end of file is reached.
4. Call **CInetConn::CloseFile()** to close the URL file stream.
5. Repeat steps 2 to 4 as many times as necessary.
6. Call **CInetConn::CloseConnection()** to terminate the connection.

A pointer back to the Internet session object is maintained within **CInetConn**, and inherited by both **CGopherConn**

and **CFTPConn**. This pointer may be accessed via the **CInetConn::GetSession()** member, and is used internally by **CInetConn** to delete the pointer from the session object's 'currently open connections' list.

For each Internet session established, we can open any number of protocol connections. The **CFTPConn** and **CGopherConn** classes implement these connections. The **CInetConn** base class maintains the Internet handle corresponding to the connection. The **CGopherConn** and **CFTPConn** classes have member functions which wrap some of the possible operations provided by WinINet for these protocols.

CInetConn Member Functions

The members of **CInetConn** give basic IO capabilities to **CGopherConn** and **CFTPConn**, and are tabulated below:

Member Function	Description
CInetConn()	A constructor which initializes the connection protocol handle and session pointer.
~CInetConn()	A destructor which closes the connection handle if necessary.
OpenUrl()	Opens a specified URL file stream.
ReadFile()	Reads from an opened URL file stream.
CloseFile()	Closes the currently open URL file stream.
GetSession()	Gets a pointer to the session from which this **CInetConn** object was instantiated.
CloseConnection()	Closes the currently open connection.

CInetConn::OpenUrl()

```
BOOL CInetConn::OpenUrl(LPCTSTR url, DWORD flags, DWORD context, LPCSTR headers, DWORD
hdrLength)
{
    ASSERT(m_hConnection != NULL);
    m_hFile = InternetOpenUrl( m_hConnection,url, headers, hdrLength,
        flags, context);
    return ((NULL == m_hFile)? FALSE: TRUE);
}
```

This function is a thin wrapper for the **InternetOpenUrl()** call. We first verify that the open connection, **m_hConnection**, is valid, then we call **InternetOpenUrl()**, passing the parameters. The URL to be opened is specified in the **url** parameter. Optional headers can be provided through the **headers** and **hdrLength** parameters. If supplied, the **headers** parameter should be in standard HTTP header format for HTTP server processing. The **headers** string can be zero delimited, in which case **hdrLength** can be set to **-1L**. Otherwise, the length specified by **hdrLength** will be taken to be valid against the header string. For asynchronous mode, a context can be supplied via the **context** parameter.

The **InternetOpenUrl()** call also takes a **flags** parameter. The value for **flags** can be one or more of:

Flag	Meaning
INTERNET_FLAG_RELOAD	Get the data again from the source.
INTERNET_FLAG_DONT_CACHE	Do not use caching in retrieving the URL. This flag is most appropriate for most intranet applications.
INTERNET_FLAG_RAW_DATA	Return **WIN32_FIND_DATA** for FTP and **GOPHER_FIND_DATA** for gopher URLs.
INTERNET_FLAG_SECURE	Request SSL or PCT service.

INTERNET_FLAG_EXISTING_CONNECT Try to reuse existing connection in opening the URL if possible.

Most intranet applications, like CIS, can use the default value of **INTERNET_FLAG_DONT_CACHE | INTERNET_FLAG_EXISTING_CONNECT**.

If this call is successful, a file level handle, **m_hFile**, is returned and kept within the **CInetConn** class. This handle is subsequently used in **ReadFile()** calls.

CInetConn::ReadFile()

```
BOOL CInetConn::ReadFile(LPVOID lpBuffer, DWORD byteIn, LPDWORD byteOut)
{
    ASSERT((m_hConnection != NULL)&&(m_hFile != NULL));
    return InternetReadFile(m_hFile, lpBuffer, byteIn, byteOut);

}
```

Once a URL resource has been opened, a file stream is available from which data can be read. The **CInetConn::ReadFile()** function takes a buffer and fills it with bytes from the open stream. Internally, it simply calls the **InternetReadFile()** WinINet API.

For the **InternetReadFile()** API, the first parameter is a 'file' level handle (i.e. **m_hFile**). The syntax of the API is modeled on the Win32 **ReadFile()** call. **lpBuffer** points to the buffer area to store the data read. This function guarantees that **byteIn** number of bytes will always be read unless an end of file condition is reached. In an end of file condition, the return value will be **TRUE**, but the **byteOut** parameter will be set to less than **byteIn**. If **byteIn** number of bytes isn't available, the API will block until the specified number of bytes are read or if end of file is reached (unless the session is operating in the asynchronous mode).

CInetConn::CloseFile()

```
void CInetConn::CloseFile()
{
    if (NULL != m_hFile )
        InternetCloseHandle(m_hFile);

}
```

The **CInetConn::CloseFile()** function simply closes the internal file level handle. It should be used in between every URL read, since each new URL fetch requires a separate **CInetConn::OpenUrl()** call. Note that only the file level handle is closed; the protocol connection handle is left intact.

CInetConn::GetSession()

```
CInetSession * CInetConn::GetSession()
{
    return pm_hSession;
}
```

CInetConn::GetSession() provides external access to the session handle. This is typically used to make session level calls given a **CInetConn** or derived object.

CInetConn::CloseConnection()

```
void CInetConn::CloseConnection()
{
    CloseFile();
    if (NULL != m_hConnection)
    {
        pm_hSession->DeleteConn(this);
        InternetCloseHandle(m_hConnection);
        m_hConnection = NULL;
```

```
        }
    }
```

This function is called when you need to close the connection without destroying the **CInetConn** object. The function is implemented by first closing any open file level handles, follow by closing of the connection level handle, **m_hConnection**, using **InternetCloseHandle()**. It also maintains the open connections list within the associated **CInetSession** object by calling its **CInetSession::DeleteConn()** function.

CGopherConn Class

Derived from **CInetConn** class, the **CGopherConn** class specializes it for the Gopher protocol operations. The simplicity of the Gopher protocol has kept the number of APIs and their parameters down. The members in the **CGopherConn** class may be used for:

- Manipulating Gopher locators, via the **CreateLocator()** and **GetLocatorType()** member functions.
- Enumerating Gopher menus, via the **GetFirstItem()** and **GetNextItem()** member functions.
- Accessing Gopher item contents, via the **OpenFile()** and **ReadEntireFile()** member functions.

CGopherConn protocol connection objects are created during a session through **CInetSession::ConnectGopher()** calls. Connections to multiple Gopher servers can be maintained per session through the instantiation of multiple **CGopherConn** objects.

A typical Gopher based application can use this class following these steps:

1. Create a **CGopherConn** object using the **CInetSession::ConnectGopher()** function.
2. Request the user to enter a Gopher host.
3. Call **CGopherConn::CreateLocator()** with the supplied host and all default parameters to locate the root Gopher directory on the host.
4. Call **CGopherConn::GetFirstItem()** with the locator to obtain the first **CGopherItem**.
5. Call **CGopherConn::GetNextItem()** to enumerate all available Gopher items at this level.
6. Display the friendly name of all the available Gopher items for the user to select.
7. When the user selects an item, check its locator type via the **CGopherConn::GetLocatorType()** member, repeat from step four if the selected item is a directory; if the selected item is a text file, call the **CGopherConn::GetEntireFile()** to read the file and display it to the user.
8. Keep repeating from step 4 until user aborts.
9. Disconnect from the session by destroying the **CGopherConn** object.

CGopherConn Member Functions

A complete list of member functions is given in the following table:

Member Function	Description
CgopherConn()	A constructor which initializes the connection protocol handle and session pointer.
~CgopherConn()	A destructor which closes the connection protocol handle if necessary.

CreateLocator()	Takes the components and makes a locator.
GetFirstItem()	Obtains the first Gopher item in an enumeration.
GetLocatorType()	Examines a locator and returns its type.
GetNextItem()	Obtains the next Gopher item during an enumeration.
OpenFile()	Opens a Gopher file given a locator.
ReadEntireFile()	Reads an entire Gopher file from remote into a **CString** object.

CGopherConn::CreateLocator()

```
BOOL CGopherConn::CreateLocator(LPCTSTR Host, INTERNET_PORT Port,
    LPCTSTR FriendlyName, LPCTSTR Selector,
    UINT Type, CString & Locator)
{
    ASSERT(m_hConnection != NULL);

    LPTSTR tmpStr = Locator.GetBuffer(coniMAX_LOCATOR_SIZE);
    if (tmpStr == NULL)
        return FALSE;

    DWORD tmpSize = coniMAX_LOCATOR_SIZE;
    if(GopherCreateLocator(
        Host,
        Port,
        FriendlyName,
        Selector,
        Type,
        tmpStr,
        &tmpSize) == FALSE)
    {
        *tmpStr = '\0';
        Locator.ReleaseBuffer();
        return FALSE;
    }

    Locator.ReleaseBuffer(tmpSize);
    return TRUE;
}
```

CGopherConn::CreateLocator() fabricates a Gopher locator strings by combining parts which makes it up: a hostname, a port number, a friendly name, a selector string and a Gopher locator type. Acceptable locator types are documented later in the **CGopherConn::GetLocatorType()** function. Internally, **CGopherConn::CreateLocator()** is implemented by calling the **GopherCreateLocator()** WinINet API call. **CGopherConn::CreateLocator()** will build the locator string into a **CString** reference variable.

The **GopherCreateLocator()** API doesn't require any Internet handle. It accepts component parts and makes up a Gopher locator. Typically, a Gopher locator is created to call the **GopherFindFirstFile()** function to begin enumeration of available Gopher items from the remote server.

Host specifies the Gopher server hostname or IP address. **Port** is the port on which the Gopher server is listening, passing a value of **INVALID_PORT_NUMBER** will use the well known port number for Gopher. **FriendlyName** selects the file or directory to be displayed by the server, a **NULL** value will obtain the default Gopher directory. **Selector** is the string to be sent to the Gopher server to obtain information, again use **NULL** for this parameter if you're constructing a Gopher locator for use with the **GopherFindFirstFile()** API. **Type** specifies the type of the Gopher item. **Locator** is a pointer to a buffer to hold the locator returned. **lpdwBufferLength** holds the size of **Locator** upon entry and the actual size of the locator upon return.

As in CIS, you frequently need to get at the root directory of a specific host. For this, we can call

CGopherConn::GopherCreateLocator() with,

hostname = Gopher host name
port = **INVALID_PORT_NUMBER** (use the well-known port)
FriendlyName = **NULL** (use default Gopher directory)
Selector=**NULL**

to obtain a locator which we will pass to **CGopherConn::GetFirstItem()** for enumerating the root directory.

CGopherConn::GetFirstItem()

```
BOOL CGopherConn::GetFirstItem( CGopherItem & gItem,
    LPCTSTR Locator, LPCTSTR SearchString,
    DWORD Flag, DWORD Context)
{
    ASSERT(m_hConnection != NULL);

    if ((gItem.m_hItem = GopherFindFirstFile(
        m_hConnection,
        Locator,
        SearchString,
        &(gItem.m_Data),
        Flag,
        Context)) == NULL)
        return FALSE;
    return TRUE;
}
```

CGopherConn::GetFirstItem() is used in enumeration of a Gopher directory. It takes as parameters an input **Locator** parameter, typically obtained from a previous **GopherCreateLocator()** call. It can also take an optional **SearchString**, **Flag**, and **Context** parameter. Upon return, the input **CGopherItem** object will be filled with information for continued enumeration through repeated **CGopherConn::GetNextItem()** calls. The function is implemented by calling **GopherFindFirstFile()** WinINet API and passing the matching parameters.

GopherFindFirstFile() requires a protocol connection level handle, which, in our case, is **m_hConnection**. It will connect to the Gopher server and locate the requested documents, files, directory tree, or other Gopher compatible search resources. The returned 'file-find' handle is stored in **m_hItem** member of the input **CGopherItem** reference object: **gItem**.

Locator can be a locator created by **CreateLocator()** pointing to a remote Gopher server, as in our case. It can also be **NULL** if the default top-most level of the Gopher directory is desired. The **SearchString** parameter is used only with index servers where extra search criteria are required. We need to pass a pointer to a **GOPHER_FIND_DATA** structure, which the API will fill with a locator and information on the first Gopher item for enumeration. We do this by passing a pointer to the **m_data** member of the input **CGopherItem** reference object : **&(gItem.m_Data).**
Flag can have one or more of these values:

Flag	Meaning
INTERNET_FLAG_RELOAD	Get the data again from the source.
INTERNET_FLAG_DONT_CACHE	Do not use caching in retrieving the URL. This flag is most appropriate for most intranet applications.
INTERNET_FLAG_RAW_DATA	Return **WIN32_FIND_DATA** for FTP and **GOPHER_FIND_DATA** for gopher URLs.
INTERNET_FLAG_SECURE	Request SSL or PCT service.
INTERNET_FLAG_EXISTING_CONNECT	Try to reuse the existing connection to open the URL if possible.

Context is the context information that will be passed to the status callback function during the asynchronous mode of operations.

CGopherConn::GetLocatorType()

```
BOOL CGopherConn::GetLocatorType( LPCTSTR Locator, DWORD & Type)
{
    return(GopherGetLocatorType(Locator, &Type));
}
```

This member is simply a wrapper for the **GopherGetLocatorType()** API. Given a Gopher locator retrieved from the remote server, this function scans the locator and determines its type. **Locator** contains the null-terminated Gopher locator string. **Type** points to a **DWORD** where the type will be returned. There are different types of locator, all listed here:

Type	Description
GOPHER_TYPE_TEXT_FILE	An ASCII text file.
GOPHER_TYPE_DIRECTORY	A Gopher directory.
GOPHER_TYPE_CSO	A CSO phone book server.
GOPHER_TYPE_ERROR	An error condition.
GOPHER_TYPE_MAC_BINHEX	A Macintosh file in BINHEX format.
GOPHER_TYPE_DOS_ARCHIVE	A DOS archive file.
GOPHER_TYPE_UNIX_UUENCODED	A UUENCODED file.
GOPHER_TYPE_INDEX_SERVER	An index server.
GOPHER_TYPE_TELNET	A Telnet Server.
GOPHER_TYPE_BINARY	A binary file.
GOPHER_TYPE_REDUNDANT	Refers to a duplicated server.
GOPHER_TYPE_TN3270	A TN3270 server.
GOPHER_TYPE_GIF	A GIF graphics file.
GOPHER_TYPE_IMAGE	An image file.
GOPHER_TYPE_BITMAP	A bitmap file.
GOPHER_TYPE_MOVIE	A movie file.
GOPHER_TYPE_SOUND	A sound file.
GOPHER_TYPE_HTML	An HTML document.
GOPHER_TYPE_PDF	A PDF file.
GOPHER_TYPE_CALENDAR	A calendar file.
GOPHER_TYPE_INLINE	An inline file
GOPHER_TYPE_UNKNOWN	The item type is unknown.
GOPHER_TYPE_ASK	An Ask+ item.
GOPHER_TYPE_GOPHER_PLUS	A Gopher+ item.

CGopherConn::GetNextItem()

```
BOOL CGopherConn::GetNextItem(CGopherItem & FirstItem,
    CGopherItem & NewItem)
{
    // the file-find handle lives with the first Gopher Item
    return(InternetFindNextFile(
        FirstItem.m_hItem,
        (LPVOID) (&NewItem.m_Data)) );
}
```

This method is used in conjunction with **GetFirstItem()** to get all the items available at a specified locator, and

simply wraps the **InternetFindNextFile()** API.

The **InternetFindNextFile()** requires two parameters. The first parameter must be a 'file-find' level handle. This handle is filled into the **CGopherItem** object by the **GetFirstItem()** member function. The second parameter is a void pointer to a **GOPHER_FIND_DATA** structure. We obtain this information from the two **CGopherItem** objects passed to the method.

Each call to **GetNextFile()** with the same **FirstItem** parameter will fetch the next available item. The call will fail (return code equals **FALSE**) and the **GetLastError()** API will return **ERROR_NO_MORE_FILES** if the end of list is reached.

CGopherConn::OpenFile()

```
BOOL CGopherConn::OpenFile( LPCSTR locator, DWORD flags,
    LPCSTR View, DWORD context)
{
    ASSERT(m_hConnection != NULL);

    if (NULL != m_hCurFile)
    {
        InternetCloseHandle(m_hCurFile);
        m_hCurFile = NULL;
    }

    m_hCurFile = GopherOpenFile( m_hConnection,
    locator, View, flags, context);

    if (NULL == m_hCurFile)
        return FALSE;
    else
        return TRUE;
}
```

Once the desired Gopher information has located, and before it can be accessed, the Gopher data stream must be 'opened' and 'read'. This is done with the API calling the **GopherOpenFile()** and **InternetReadFile()** functions. The next function, **CGopherConn::ReadEntireFile()** takes this approach. **CGopherConn::OpenFile()** simply wraps the **GopherOpenFile()** WinINet API. It first makes sure that the 'file level handle' member **CGopherConn::m_hCurFile** is not currently open, and, if it is, closes it.

The **GopherOpenFile()** function opens a Gopher file at the remote server. A file level handle is returned by the function if successful. This file handle can be used in **InternetReadfile()** API to access the remote file, in our case through the **CGopherConn::ReadEntireFile()** wrapper.

This function requires a connection level handle **m_hConnection**. It takes a locator, **locator**, typically from the enumeration process using **CGopherConn::GetFirstFile()** or **CGopherConn::GetNextFile()**. Some Gopher servers offer more than one view of the file. **View** specifies which view of the file to open. In our case, this doesn't apply and we'll use **NULL**. **flags** controls the use of the cache and can be any values allowed for **CGopherConn::GetFirstItem()**. **context** is context information passed to the status callback function for asynchronous operations.

CGopherConn::ReadEntireFile()

```
BOOL CGopherConn::ReadEntireFile( LPCTSTR locator,
    CString & wholeFile)
{
    LPTSTR aBuf;
    DWORD readSize;

    if (OpenFile(locator))
    {
        aBuf = new char [MAX_BUF_SIZE];
```

```
            if (NULL != aBuf)
            {
                while( InternetReadFile(m_hCurFile, aBuf, MAX_BUF_SIZE,
                    &readSize))
                {
                    if(0 == readSize) break; // this is EOF
                    // assume all text files
                    aBuf [readSize] = '\0';
                    wholeFile = wholeFile + aBuf;
                }
                InternetCloseHandle(m_hCurFile);

            }
            else
            {
                delete [] aBuf;
                return TRUE;
            }
            delete [] aBuf;
        }
        else
            return FALSE;
        return TRUE;
    }
```

Leveraging the flexibility of a **CString** to handle very large variable size strings and the structure of our **CGopherConn** class definition, the **CGopherConn::ReadEntireFile()** function opens a Gopher information locator pointed to by **locator** and reads all its content into a supplied **CString** reference variable: **wholefile**. The implementation is straightforward. First the **CGopherConn::OpenFile()** function is used to open a file level handle, then **InternetReadFile()** is called to repeatedly read the remote file until end of file is reached. As the file is being read, a temporary buffer **aBuf** is used to contain the pieces. Meanwhile, the pieces in **aBuf** are repeatedly concatenated into the **CString** reference input variable: **wholefile**.

Our CIS application makes extensive use of this function to greatly simplify Gopher text file transfers.

CFTPConn Class

CFTPConn is another **CInetConn**-derived class. It specializes the protocol connection for FTP operations by providing a comprehensive set of member functions. **CFTPConn** objects and the protocol connection is created by **CInetSession::ConnectFTP()** calls. You can have several FTP connections simultaneously active within one **CInetSession**. Some of the services offered by the **CFTPConn** class include:

Remote Directory Manipulation, via **CreateDirectory()**, **GetCurrentDirectory()**, **RemoveDirectory()** and **SetCurrentDirectory()**.

File Manipulation, via **DeleteFile()**, **GetFile()**, **PutFile()**, **SubmitAFile()** and **RenameFile()**.

Most **CFTPConn** member functions provides a thin wrapping over the corresponding WinINet API functions, supplying hidden handles and applicable default parameters wherever applicable. Specifically, the enumeration functions of the WinINet API for FTP have not been wrapped. This is mainly due to the very simplistic requirement of CIS, which doesn't need to find or traverse directories on the remote FTP server.

A typical FTP-based application can use the **CFTPConn** class following these steps:

1. Create a **CFTPConn** object using the **CInetSession::ConnectFTP()** function, taking from the user the appropriate host name, user ID and password as required
2. Ask the user for the file name on the remote host to be downloaded, or allow the user to select a local file for upload (if you want to be able to allow the user to select from remote files, the **CFTPConn** class must be extended to cover enumeration).

3 Call `CFTPConn::PutFile()` to upload a file or `CFTPConn::GetFile()` to download a file.

4 Repeat step 2 and 3 until all desired file transfers have been done.

5 Disconnect from session by destroying the `CFTPConn` object.

CFTPConn Members Functions

Here is a table of the member functions:

Member Function	Description
`CFTPConn()`	A constructor which initializes the handle.
`~CFTPConn()`	A destructor which closes the handle if necessary.
`CreateDirectory()`	Creates a directory on the remote server.
`DeleteFile()`	Deletes a file on the remote server.
`GetCurrentDirectory()`	Gets the current working directory from the remote server.
`GetFile()`	Retrieves a file from the remote server to local disk.
`PutFile()`	Sends a file from the local disk to the remote server.
`RemoveDirectory()`	Removes a directory from the remote server.
`RenameFile()`	Renames a file on the remote server.
`SetCurrentDirectory()`	Changes the current working directory on the remote server.
`SubmitAFile()`	Opens a file for writing on the remote server, fills the file with content from a `CString` variable, names the file with a unique name composed of user ID and a random number and completes transfer before returning.

CFTPConn::CreateDirectory()

```
BOOL CFTPConn::CreateDirectory(LPCTSTR DirName)
{
    ASSERT(m_hConnection != NULL);

    return(FtpCreateDirectory(m_hConnection,
        DirName));
}
```

This function is a thin wrap over the `FtpCreateDirectory()` WinINet function, supplying only the connection level handle `m_hConnection`. It will attempt to create a directory on the remote system. `DirName` is the name of the directory to create, the allowable length is dependent on the specific FTP server implementation. The path of the directory specified can be absolute or relative to the current working directory on the remote server.

CFTPConn::DeleteFile()

```
BOOL CFTPConn::DeleteFile(LPCSTR Name)
{
    ASSERT(m_hConnection != NULL);

    return(FtpDeleteFile(m_hConnection,
        Name));
}
```

This function deletes the named file, `Name`, from the remote system at the current directory. It's a thin wrap over the `FtpDeleteFile()` WinINet function. The file path specified, `Name`, can be either an absolute or a relative path from the current directory.

CFTPConn::GetCurrentDirectory()

```
BOOL CFTPConn::GetCurrentDirectory(CString & DirName)
{
    ASSERT(m_hConnection != NULL);
    LPTSTR tmpStr = DirName.GetBuffer(MAX_PATH);
    DWORD tmpSize = MAX_PATH;

    if (NULL == tmpStr)
        return FALSE;
    if(FALSE == FtpGetCurrentDirectory( m_hConnection,
        tmpStr,
        &tmpSize))
    {
        *tmpStr = '\0';
        DirName.ReleaseBuffer();
        return FALSE;
    }
    DirName.ReleaseBuffer();
    return TRUE;
}
```

This function wraps the **FtpGetCurrentDirectory()** WinINet function. It returns the current directory on the remote server in a supplied **CString** reference parameter: **DirName**. An assumption is made that the maximum length of a file path does not exceed the value of the **MAX_PATH** constant.

CFTPConn::GetFile()

```
BOOL CFTPConn::GetFile( LPCTSTR RemoteFile, LPCTSTR NewFile,
    BOOL FailIfExist,
    UINT Attributes, UINT Flags, UINT Context)
{
    ASSERT(m_hConnection != NULL);

    return(FtpGetFile( m_hConnection,
        RemoteFile,
        NewFile,
        FailIfExist,
        Attributes,
        Flags,
        Context));
}
```

This function performs a file download from the remote server to the local machine in one shot. It wraps the **FtpGetFile()** WinINet API function and supplies a connection level handle, **m_hConnection**.

For the **FtpGetFile()** function, **m_hConnection** is a connection-level handle. **RemoteFile** contains the remote file name to be retrieved. **NewFile** contains the local file name to be written. **FailIfExist** controls whether the API will fail if the local file already exists. **Attributes** specifies the attributes of the newly created file. It can contain any flags and attributes allowed by the Win32 **CreateFile()** API. **Flags** can be either **FTP_TRANSFER_TYPE_ASCII** or **FTP_TRANSFER_TYPE_BINARY**, although it's best to use **FTP_TRANSFER_TYPE_BINARY** for PC to PC file transfers. **Context** is context information passed to the status callback function during asynchronous mode of operation.

A return value of **TRUE** indicates that the file is successfully retrieved.

CFTPConn::PutFile()

```
BOOL CFTPConn::PutFile(LPCTSTR LocalFile, LPCSTR RemoteFile,
    UINT Flags, UINT Context)
{
    ASSERT(m_hConnection != NULL);

    return( FtpPutFile( m_hConnection,
        LocalFile,
        RemoteFile,
```

```
            Flags,
            Context));
    }
```

This function uploads a file from the local computer to the remote computer in one shot. It provides a wrap of the **FtpPutFile()** WinINet API call.

For **FtpPutFile()**, **m_hConnection** is a connection level handle. **LocalFile** contains the path of the local file to be transferred. **RemoteFile** contains the path where the remote file is to be created. **Flags** can be either **FTP_TRANSFER_TYPE_ASCII** or **FTP_TRANSFER_TYPE_BINARY**, use the default **FTP_TRANSFER_TYPE_BINARY** for PC to PC file transfers. **Context** is context information for the status callback function during asynchronous mode of operation.

A return value of **TRUE** indicates that the file is successfully retrieved.

CFTPConn::RemoveDirectory()

```
    BOOL CFTPConn::RemoveDirectory(LPCTSTR DirName)
    {
        ASSERT(m_hConnection != NULL);

        return(FtpRemoveDirectory(m_hConnection,
            DirName));
    }
```

This function removes a directory from the remote server. It wraps the **FtpRemoveDirectory()** WinINet API.

The directory specified by **DirName** can be absolute or relative to the remote working directory. A return value of **TRUE** indicates that the remote directory has been successfully removed.

CFTPConn::RenameFile()

```
    BOOL CFTPConn::RenameFile(LPCTSTR ExistingName, LPCSTR NewName)
    {
        ASSERT(m_hConnection != NULL);

        return(FtpRenameFile(m_hConnection,
            ExistingName,
            NewName));
    }
```

This function wraps the **FtpRenameFile()** WinINet API.

It renames a file on the remote FTP server if possible. **m_hConnection** is a connection level handle, **ExistingName** contains the name of the existing file on the remote server. **NewName** contains the new name of the file.

If this function returns **TRUE**, the file has been renamed successfully.

CFTPConn::SetCurrentDirectory()

```
    BOOL CFTPConn::SetCurrentDirectory(LPCTSTR DirName)
    {
        ASSERT(m_hConnection != NULL);

        return(FtpSetCurrentDirectory(m_hConnection,
            DirName));
    }
```

This function is a wrap for the **FtpSetCurrentDirectory()** API.

It changes the current working directory of an FTP connection. **m_hConnection** is a connection-level handle.

DirName contains the path of the desired current directory. The path can be relative to current directory or a full path.

A return value of **TRUE** indicates the current directory has been successfully changed.

CFTPConn::SubmitAFile()

```
BOOL CFTPConn::SubmitAFile(CString & myContent)
{
    char fname[MAX_PATH];
    GetTempFileName(".",CIS_UID_PREFIX,0, fname);

    if (NULL != m_curFTPFile)
    {
        InternetCloseHandle(m_curFTPFile);
        m_curFTPFile = NULL;
    }

    m_curFTPFile = FtpOpenFile(m_hConnection, fname, GENERIC_WRITE,
        FTP_TRANSFER_TYPE_BINARY, 0);

    TRACE("Filename is %s\n", fname);
    if (NULL != m_curFTPFile)
    {
        DWORD tpLength = myContent.GetLength();
        DWORD tpWritten;
        LPTSTR aBuf;
        if (InternetWriteFile(m_curFTPFile,
            aBuf = myContent.GetBuffer(tpLength),
            tpLength, &tpWritten))
        {
            while (tpWritten < tpLength)
            {
                tpLength = tpLength - tpWritten;
                aBuf += tpWritten;
                if (FALSE ==
                    InternetWriteFile(m_curFTPFile,
                    aBuf,
                    tpLength, &tpWritten))
                    break;
            }
        }
        myContent.ReleaseBuffer();
        InternetCloseHandle(m_curFTPFile);
    }
    else
        return FALSE;
    return TRUE;
}
```

SubmitAFile() is a highly customized function specifically designed to work with the CIS application. It's designed to upload a file containing a variable size string, input as a **CString** reference variable **myContent**, to the root directory of a connected FTP server using a temporary name and anonymous FTP. **SubmitAFile()** does its work by calling **FtpOpenFile()** to open the remote file for upload, then repeatedly calling **InternetWriteFile()** until the file upload is completed. CIS counts on this function for uploading abstracts from user input forms to the FTP server.

CGopherItem and CFTPItem Classes

Last but not least, we have the item classes, which correspond to FTP and Gopher directories, links or files. **CInetItem** is the virtual base class for **CGopherItem** and **CFTPItem**. **CInetItem** maintains the 'file-find' level handle associated with the item in **m_hItem**. Ideally, **CGopherItem** and **CFTPItem** should actually maintain the file level handle associated with the item. Unfortunately, the FTP protocol only allows one open file per protocol connection, which makes the **m_hItem** member of **CFTPItem** rather meaningless. The following are the class definitions:

```
class CInetItem
{
// Construction
public:
    VOID CloseItemHandle();
    CInetItem();
    virtual ~CInetItem();

    HINTERNET m_hItem;
};

class CFTPItem:public CInetItem
{
public:
    CFTPItem();
    virtual ~CFTPItem();

    WIN32_FIND_DATA m_Data;

};


class CGopherItem:public CInetItem
{
public:
    BOOL GetLocator(CString & locator, DWORD & GType);
    CGopherItem();
    virtual ~CGopherItem();
    GOPHER_FIND_DATA m_Data;
};
```

'File-find' level handles are used in enumerating the elements available in an FTP directory or at a locator location within Gopher. To make the relationship between the classes even more complicated, only the first 'file-find' level handle in an enumeration list is really significant. This is partially due to protocol restrictions, and partially due to Microsoft's desire to mimic the GetFirst-FindNext style of programming familiar to DOS programmers in WinINet. During an enumeration of items, the first 'file-find' handle returned from a WinINet **FtpGetFirstFile()** or **GopherGetFirstFile()** call is repeatedly passed into an **InternetGetNextFile()** call to obtain successive items. WinINet (actually the remote server) keeps the state of the session internally between calls to **InternetGetNextFile()**.

To work around the complications described above, the **CInetItem** handle is valid only if it is the first item on an enumerated list. You could design a 'file-find' object purely for handling the enumeration process, but it would be overkill for our CIS project.

CFTPItem and **CGopherItem** classes mainly contain data corresponding to items from a FTP or Gopher enumeration. For example, the CIS application maintains a map of **CGopherItem** objects in its document class in order to quickly retrieve a Gopher item associated with the user button push. **CInetItem** is actually the only class in the hierarchy derived from the MFC **CObject** base class. This adds serialization support for the **CFTPItem** and **CGopherItem** which will allow us to save these objects or a list of these objects easily to a file.

Implementation of the CIS

To implement CIS, we first start with a skeletal set of code generated from App Wizard. The base code is generated with the following App Wizard options selected:

MDI Application
OLE Full Server Enabled
OLE Automation Enabled
MFC in Static Library

We won't take advantage of most of the enabled OLE options until a later chapter, but generating the base project with the appropriate options will avoid awkward retrofitting when we need the support later on.

The generated project provides us with these classes:

CAboutDlg
CCISOLE2App
CCISOLE2SrvItem
CInPlaceFrame
CChildFrame
CCISOLE2Doc
CCISOLE2View
CMainFrame

To the basic framework set of code, we add the new functionality. CIS operates within a single document framework, but we are enabling MDI support to make the later transition to OLE simpler. Most of the MDI handling capability is effectively disabled by eliminating the capability to create or open a new document. This is accomplished by removing the associated menu item and toolbar buttons.

To display the user interface, we divide the client area of the window into fifteen equal areas and draw the buttons tiled into this area. The **OnDraw()** function of the **CCISOLE2View** class handles this; **m_labels** is the array of buttons.

```
void CCISOLE2View::OnDraw(CDC* pDC)
{
    CCISOLE2Doc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

      < code segment skipped >

    UINT wUnit, hUnit;
    CRect myRect;
    GetClientRect(&myRect);
    wUnit = myRect.Width()/5;
    hUnit = myRect.Height()/3;
    for (int i=0; i<5; i++)
    {
        for (int j=0; j<3; j++)
        {
            UINT tmpIdx = (i*3)+j;
            m_Labels[tmpIdx].Create( aStr,
                WS_CHILD | WS_VISIBLE |
                BS_PUSHBUTTON | BS_OWNERDRAW,
                CRect(i*wUnit,j*hUnit,(i+1)*wUnit-1,
                (j+1)*hUnit-1),this, tmpIdx);

            m_Labels[tmpIdx].SetBitmaps( decoded, aBitmap);
            m_Labels[tmpIdx].SetWindowText(aStr); // hidden text!
        }
}

< code segment skipped>
}
```

The **CButton** and the **CBitmapButton** classes provided by MFC are inadequate for our purposes, as we need to display both a custom bitmap and a button text on the button. Both the bitmap and text must also resize when the client area changes. We must create a custom button class, **CCISButton**, which we'll describe in a later section.

To create the custom forms that can be triggered through the decoding of a retrieved Gopher item, we use the dialog editor to create the dialog resource **IDD_INFOVIEW**, **IDD_ORDER**, **IDD_SIGNUP**, and **IDD_SUGGEST**. We then use ClassWizard to create the corresponding C++ **CDialog** derived classes, **CFormDView**, **CFormOrder**, **CFormSignup**, and **CFormSugg**. These are the general purpose information viewing form, a stationary supplies order form, an activities sign-up form and a comment submission form, respectively.

CIS can be extended by adding more custom forms here. The code in these form are largely untouched from how they were generated in, except that member variables are defined for the fields and a **GetFormInfo()** function is added for all the submittable forms. The purpose of **GetFormInfo()** is to construct a single **CString** variable containing the concatenated string representation of all the field values and a form identification, each separated by a **|** character. The application will call this function to create the form submission content. If you want to make CIS more useful, you can definitely add more data validation and custom code to these forms.

The User Interface

To provide a very easy to use interface, the information will be presented to the user as a panel of buttons. As we mentioned above, the **CButton** and **CBitmapButton** classes in MFC are very simple. We need a button class which can display both a bitmap and variable text and which can scale with the client area of the hosting frame window. To accomplish this, we create our own class, called **CCISButton**.

The **CCISButton** class implements a **SetBitmaps()** function, which allows the application to specify the bitmaps and the text to be displayed on the button; up to four bitmaps and one button label text string can be specified. The **DrawItem()** function is overridden for drawing the button border, the bitmap and the text string when the button is in one of it's possible states (enabled, disabled, focused and pressed).

```
void CCISButton::DrawItem(LPDRAWITEMSTRUCT lpDIS)
{
    ASSERT(lpDIS != NULL);
    ASSERT(m_bitmap.m_hObject != NULL);     // required

    // use the main bitmap for up, the selected bitmap for down
    CBitmap* pBitmap = &m_bitmap;
    UINT state = lpDIS->itemState;
    if ((state & ODS_SELECTED) && m_bitmapSel.m_hObject != NULL)
        pBitmap = &m_bitmapSel;
    else if ((state & ODS_FOCUS) && m_bitmapFocus.m_hObject != NULL)
        pBitmap = &m_bitmapFocus;   // focused
    else if ((state & ODS_DISABLED) && m_bitmapDisabled.m_hObject != NULL)
        pBitmap = &m_bitmapDisabled;   // for disabled

    // draw the whole button
    CDC* pDC = CDC::FromHandle(lpDIS->hDC);
    CDC memDC;

    // create a backing store for drawing
    memDC.CreateCompatibleDC(pDC);
    CBitmap* pOld = memDC.SelectObject(pBitmap);
    if (pOld == NULL)
        return;     // destructors will clean up

    CRect rect;
    rect.CopyRect(&lpDIS->rcItem);

    ASSERT(m_bitmap.m_hObject != NULL);
    CSize bitmapSize;
    BITMAP bmInfo;
    VERIFY(m_bitmap.GetObject(sizeof(bmInfo), &bmInfo) == sizeof(bmInfo));

    // throw the bitmap on the button, make it fit
    pDC->StretchBlt(rect.left, rect.top,
        rect.Width(), rect.Height(),
```

```
            &memDC, 0, 0, bmInfo.bmWidth,
            bmInfo.bmHeight, SRCCOPY);

        // get a scale font
        UINT desiredFontSize = rect.Height()/10;
        LOGFONT logFont;    memset(&logFont, 0, sizeof(LOGFONT));
        logFont.lfHeight = desiredFontSize;

        CFont font;
        CFont* pOldFont = NULL;
        if (font.CreateFontIndirect(&logFont))
            pOldFont = pDC->SelectObject(&font);

        // find out how much space the label will take
        CRect textRect = rect;
        textRect.DeflateRect(2,2);  // take account of margins
        pDC->DrawText( m_buttonText,
            &textRect, DT_CALCRECT | DT_WORDBREAK | DT_CENTER);
        if (textRect.Height() < rect.Height())
        {
            // otherwise too much text, cannot print text
            // manually center it
            UINT padding = (rect.Width() - textRect.Width()) /2;
            textRect.OffsetRect( padding,
            rect.Height() - textRect.Height() - desiredFontSize/2);

            pDC->DrawText( m_buttonText,
                &textRect, DT_WORDBREAK | DT_CENTER);
        }

        // restore font
        if (pOldFont != NULL)
            pDC->SelectObject(pOldFont);

        COLORREF topLeft, bottomRight;
        // pressed vs normal button details
        if (state & ODS_SELECTED)
        {
            topLeft = RGB(127,127,127);
            bottomRight = RGB(255,255,255);
        }
        else
        {
            topLeft = RGB(255,255,255);
            bottomRight = RGB(127,127,127);
        }
        // Draw the 3D frame, 2 pixels thick
        rect.DeflateRect(1,1);
        pDC->Draw3dRect(&rect, topLeft, bottomRight);
        rect.DeflateRect(1,1);
        pDC->Draw3dRect(&rect, topLeft, bottomRight);

        memDC.SelectObject(pOld);
    }
```

The default **MM_TEXT** mapping mode is used to simplify the placement calculation and drawing operation. To give the button a realistic feel, we handle the left mouse button down message and repaint the button with a white 'flash'.

```
    void CCISButton::OnLButtonDown(UINT nFlags, CPoint point)
    {
        CRect rect;
        GetClientRect(&rect);

        CClientDC dc(this);

        // Flash and Draw Immediately
        dc.FillSolidRect(&rect, RGB(255,255,255));
        Invalidate();
        RedrawWindow();
```

```
        CButton::OnLButtonDown(nFlags, point);
    }
```

Document and View Class

For CIS, the application class is **CCISOLE2App**, the document class is **CCISOLE2Doc**, and the view class is **CCISOLE2View**. The application class maintains an **CInetSession** object, **m_InetSess**, corresponding to the Internet session between WinINet and the CIS application.

The document class contains a pointer to any active Gopher and/or FTP protocol connection objects. The Gopher connection object, **m_Gopher**, is created when the application starts up, in the first **OnNewDocument()** call and is maintained for the entire session. The FTP connection object, **m_FTP**, is created and destroyed on-demand when form submission occurs. This happens in the **SubmitForm()** member function.

The document class also contains an MFC template-based **CMap** class, containing a Gopher friendly name to facilitate Gopher locator mapping for the currently active Gopher menu directory.

```
    CMap<CString, LPCTSTR, CGopherItem, CGopherItem &> m_GopherMap;
```

This map is updated each time the **FillGopherMap()** is called. It happens when the user clicks on a button representing a Gopher link item or a Gopher menu directory.

The view class is responsible for:

Presenting the button interface to the user.
Intercepting button presses and providing appropriate action.
Calling the document class to traverse a Gopher menu directory.
Displaying the appropriate form for the button and submitting the results.
Displaying a remote text file to be viewed by the user.

Most of the work of presenting the button interface is done within the **OnDraw()** member function, as we discussed previously. Since the **CCISButton**s will resize themselves, the only thing you need to do when you are resizing the client area is to redraw the buttons.

The Gopher locator map is actually maintained in the document class. To 'find again' the Gopher locator corresponding to a button, the original Gopher friendly name must be passed back in a call to the document class's **GetLocator()** function. However, remember that the Gopher friendly name contains forms and button graphic information, and doesn't correspond to the button label. To overcome this difficulty, the actual window text field of the window associated with a button is used. In other words, the button label supplied in the **SetBitmaps()** call of the **CCISButton** object is different from the one supplied to the **SetWindowText()** call. The hidden window text always contain the full Gopher friendly name and can be used to find the Gopher locator. A couple of helper functions, **DecodeForm()** and **DecodeString()** are used to decode the Gopher friendly name to find the desired bitmap to display on the face of the button, and the desired form to display when a user clicks a button.

With the sample CIS, I have implemented three custom forms and a capability to read a text file of information. The **OnCommand()** function of the view class is overridden to intercept the user clicking any buttons. If the user clicks a button that corresponds to a form, the form is displayed. The text viewing form requires a call to the **ReadEntireFile()** member function of the **CGopherConn** object belonging the document class. Processing of all other forms requires calling the forms **GetFormInfo()** function to retrieve the entries, then sending the form to the remote FTP server via the document class's **SubmitForm()** member function. The implemented behavior of CIS is minimal, but can be easily extended to provide more custom forms, or more specialized document viewing or submitting capabilities.

Testing on the Network

To test the CIS over a network, you must have access to a Gopher server and an FTP server. The Microsoft Internet

Information Server (IIS) provides an HTTP (World Wide Web) service, as well as FTP and Gopher services. IIS version 1.0 is available for free download from the Microsoft web site (**`http://www.microsoft.com/`**). Version 2.0 and subsequent release will be available as a standard component of the new Windows NT server releases (4.0 and later). Note that IIS version 1.0 can only run on the server version of Windows NT. Later releases may be able to run on the workstation Windows NT.

For our example and all our script files, we'll only be using the IIS, but you may use any Gopher and FTP servers, including non-PC based implementations (i.e. UNIX variants). All configuration files are in ASCII text and can be readily edited using any ASCII editor.

Unlike a stand-alone application, testing a custom network-based application requires significant resources and preparation. You can use a **loop back** system to test CIS. The loop back system will run both the development software, the client CIS, as well as the FTP and Gopher servers from the IIS. This configuration is sufficient for initial testing, but to do so you'll need a Pentium machine with 32 megabytes of memory running Windows NT server Version 3.51 at service pack level 4 or later.

In actual production testing, however, loop back testing can't replace testing over the network. Much of the subtleties of network application-based programming won't surface until the application is actually exercised over a network. The following assumes that you either have a loop back setup or at least two machines on the network, one running NT server and another the CIS application.

Setting up the Test Bed

By definition, Internet is TCP/IP, so you must make sure that TCP/IP is properly installed on your computer. If you don't have an actual LAN card, install the dial-up adapter using the network applet in the control panel of Windows NT. If you have two machines on the network, make sure that a ping from one machine to another is okay, and vice versa.

To simplify the installation and testing, you should also make sure that a WINS server is running in your network. You can start the WINS service from the Services applet in the Control Panel. WINS will automatically map the NetBIOS-based machine name (i.e. the machine name you set up Windows NT with) to the required IP address.

I've supplied a sample corporate information data set for testing. To test it, you must have set up IIS using all default values. You should test IIS to make sure that it's working. In addition, make sure that the root of the FTP and Gopher hierarchy are descending from the same directory (typically **`Inetsrv`**), and that the Gopher root is named **`gophroot,`** while the FTP root is named **`ftproot`**. Install the data following the instructions on the CD-ROM. After the extraction, change directory to **`gophscpt`** and run the **`Mktree`** batch file, followed by the **`Try`** batch file. These files will use the IIS **`gdsset`** command to create the necessary Gopher locator hidden files. You can edit this file to customize the application.

Once you've set this up, try a regular Gopher client and make sure that you can access both the FTP and Gopher features. Most web browsers can also be used as a Gopher or FTP client. If you're using a web browser such as Microsoft Internet Explorer or Netscape Navigator, you can type in gopher://<pc name> or ftp://<pc name> to test it. There's no point in trying CIS unless both FTP and Gopher servers of the IIS are working properly. Also, make sure that write access as well as anonymous login is enabled for FTP. You can configure this through the IIS administration applet.

Finally, once you have IIS FTP or Gopher properly set up, the data files extracted and the Gopher menus created using the batch file, you're ready to test CIS. Before you run CIS, make sure that you have compiled CIS with the hostname of the server appropriately entered in **`Cisconfig.h`**.

Try out CIS. Test the menu traversal, try viewing a remote file and try submitting a form. Having gone through the design from scratch, and finally seeing it in action, by now, you'll probably have many ideas on how you can extend this skeletal intranet application for practical use. The next section will provide some more suggestions.

Extending CIS

At this stage, the CIS is far from complete. It is, however, an effective tool for experimenting with Internet/intranet-based programming, using the WinINet APIs and standard protocols. The to-do list for CIS is very long, so the following is only a partial list:

- Add more robust forms support (i.e. complete validation).
- Add more information rendering support (i.e. graphic and sound).
- Add a history list for the most recently visited locations.
- Implement archive capability to save and restore session configurations.
- Improve usability through implementation of status callback function.
- Improve the configuration procedure (i.e. fewer hard-coded constants).
- Implement robust error handling and recovery.
- Optimize the redraw to improve screen update time.
- Revisit the class structures to better divide up the work.
- Add full OLE Automation support.
- Integrate with other Internet and intranet tools.

Summary

When you consider the functionality embodied in CIS, it's amazing how little custom programming is actually required to achieve it. By leveraging off well-known and extremely well-tested FTP and Gopher protocol as our protocol building blocks, we've saved tremendous design and testing efforts associated with implementation of new custom protocols. We've made it possible to use non-PC servers (i.e. UNIX system), and have made it easier to generate and maintain standard ASCII files. We've made system configuration a lot more flexible and by using the WinINet library, we've avoided the daunting task of programming the standard protocols via Winsock, allowing us to focus on more complex functionality. Also, through the encapsulation of object-oriented design, we've wrapped the WinINet library and partitioned the problem into manageable pieces.

In this chapter, we've discussed Internet and intranet programming, and the reasons for its rising importance over the next decade. We then examined WinINet, Microsoft's operating system extension to support Internet/intranet programming. Using a hypothetical CIS as a sample, we proceeded to design and implement a usable, pragmatic, extensible framework for creating intranet. Finally, we've discussed how to test CIS, and have illustrated the complexity associated with network application testing.

The other simple, yet very important message that we've tried to get across in these pages is that the most successful Internet/intranet applications will make the most pragmatic use of both existing and new basic network programming building blocks (protocols, middleware libraries). For the professional engineer/analyst/programmer, it's important to be familiar with as many of the standard network programming building blocks as possible. These are the basic tools for the new generation of network- based applications, ultimately leading to the utopian world of true distributed computing.

In the blink of an eye, between January and March of 1996, Microsoft have managed to provide literally hundreds of new network programming building blocks through their ActiveX campaign. In the next chapter, we'll explore the very exciting ActiveX architecture and its associated network programming building blocks.

ActiveX Documents and Objects

The CIS application that we created in the previous chapter made use of the WinINET extension to give us a flexible framework upon which to build custom intranet/Internet applications. Following the rapid adoption of intranet-based web browsing by major corporations, we'll examine how to seamlessly integrate the CIS custom application into a web browser.

For example, if another department within our hypothetical corporation has standardized a new World Wide Web browser for their information sharing, we may be asked to fulfill our original design promise by enhancing the CIS to work in harmony with the browser. Access to the CIS should be transparent to the web browser user . The new Microsoft ActiveX architecture provides a mechanism to accomplish exactly this. ActiveX is the new branding for the family of technologies implementing Microsoft's new Internet strategy.

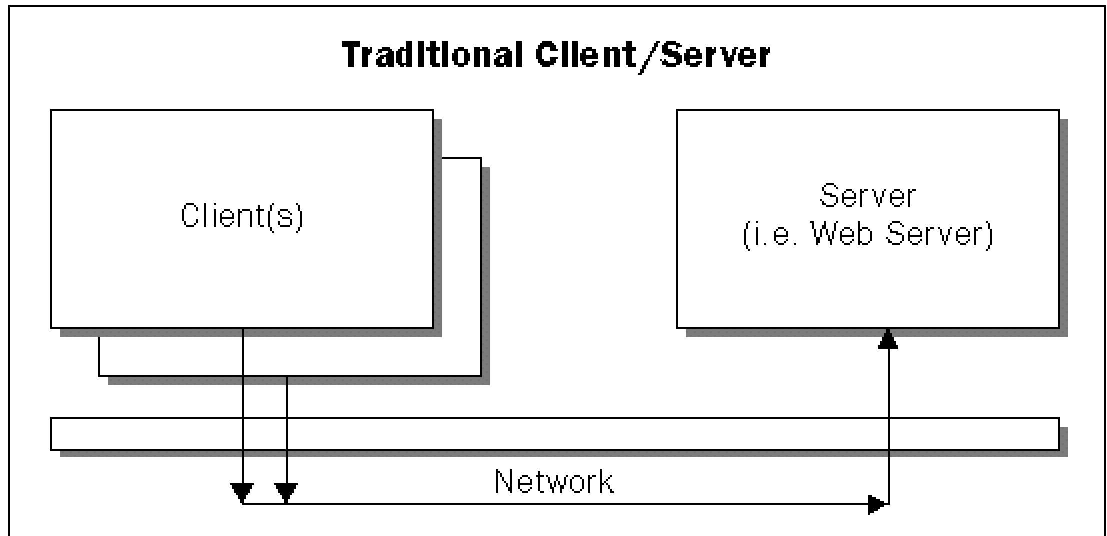
In this chapter, we'll first take a look at Microsoft's Internet strategy, explaining many of the new terms and concepts. We'll then proceed to examine an Internet extension of Microsoft's OLE technology known as **Active Documents** (or Doc Objects in Microsoft internal lingo). We'll then show you step-by-step how to enable a Doc Object within an Internet application by adding Active Document support to our CIS application. Finally, we'll test the new CIS using two new Active Document containers, including the new Internet Explorer 3.0.

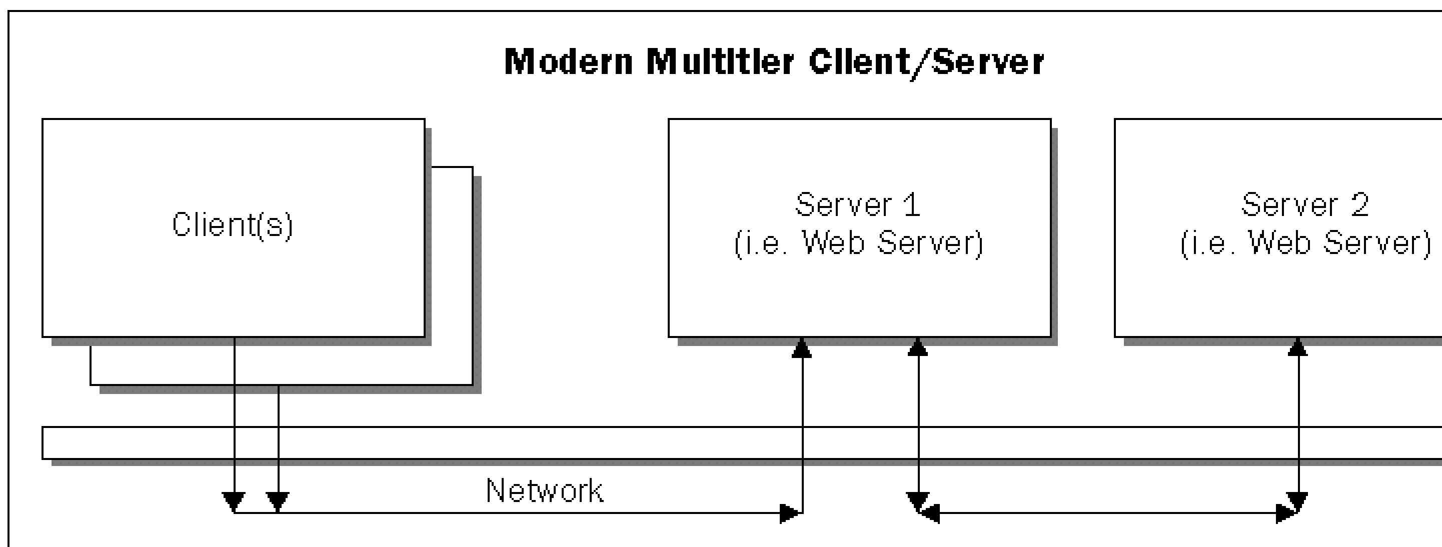
Microsoft's ActiveX Internet Strategy

In March 1996, Microsoft announced their unifying Internet strategy to the public from their Professional Developer's Conference in San Francisco. This marked the birth of Microsoft's ActiveX campaign. Appealing to the public's familiarity with the current World Wide Web, Microsoft introduced a coherent family of technologies which will bring the 'passive' world of web browsing into Microsoft's vision of an 'active' Internet universe. Microsoft is leveraging its dominance as the desktop PC platform and exploiting several leading edge web technologies, including real-time dynamic web page generation, digital signatures, secure transactions and client-side scripted applets. Beneath this is a whole new set of building blocks, tools, protocols, object access models and APIs that promise to make it much easier to write useful Internet/intranet-enabled applications.

The Client/Server Model

By far the most popular model of network programming on the Internet is the client/server model. Following this trend, the ActiveX technologies group is split between a client subset and a server subset (see below). In a typical scenario, an user will be interacting with a client application, which will then retrieve and manipulate information provided and processed by one or more server computers distributed throughout the Internet and/or the intranet.





A system where the handling of the client request involves more than one server is often called a **multitiered** client/server system. The ActiveX architecture is totally compatible with the multitiered model. For example, a client request for the latest corporate report may involve the retrieval of a web page from the web server, incorporating data retrieved in real time from a SQL database server, and triggering an e-mail message to the administrator.

Sorting through the Jargon

Let's take a look at how Microsoft plan to make the Internet active, at the same time promoting their software as the best choice for developing and servicing intranet applications.

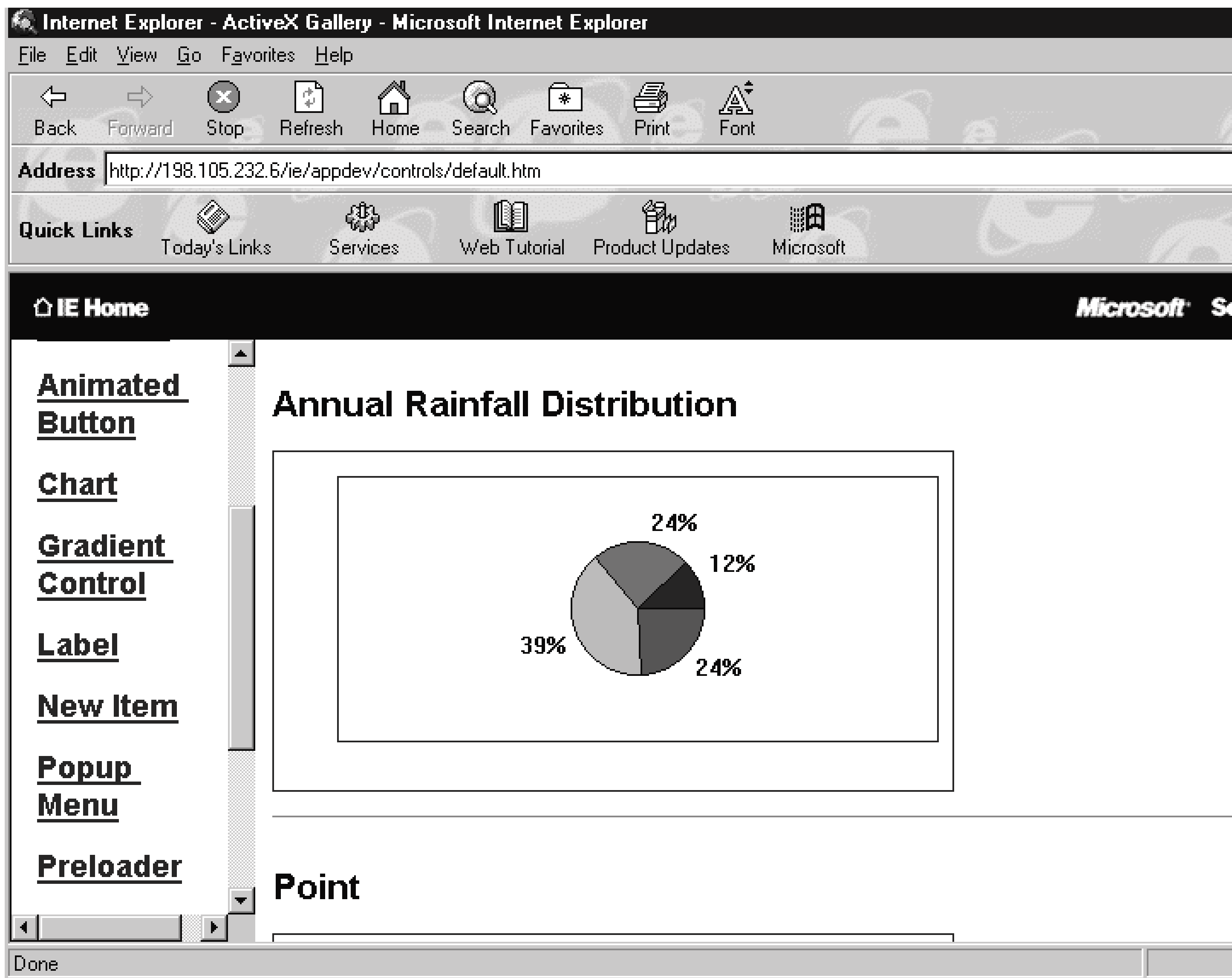
First, let's understand what's happening on the PC desktop. This can be a PC connected to the Internet occasionally via analog modem or ISDN dialup, or it can be a PC permanently or intermittently connected to a local area network offering Internet or intranet services. No discussion of the client technology will be complete without first taking a look at the Internet Explorer.

The Internet Explorer Universal Client

In December of 1995, Microsoft released their World Wide Web browser, called Internet Explorer 1.0, which is based on technologies licensed by Spyglass Inc. In March of 1996, an upgrade of the Internet Explorer, called version 2.0 was officially released. Internet Explorer 2.0 has features competitive with the leading browser from Netscape Inc. and also provides various bug fixes.

Microsoft will release Internet Explorer 3.0 in the second quarter of 1996. While appearing innocently to the user as an upgrade from Explorer 2.0, the architecture of Internet Explorer 3.0 will be radically different from 2.0 and will enable rapid transition to the fully-fledged ActiveX Shellview shortly after.

The Shellview will be an enhanced version of the current Explorer (file and system explorer, not the Internet explorer) applet provided with Windows 95 and Windows NT, which will integrate the navigation of resources on your computer (i.e. disk drive, printers), with those on the network that the computer is connected to (i.e. remote disk drive, remote printers, remote database servers), as well as the Internet/intranet (i.e. FTP sites, web servers, audio libraries). Furthermore, Shellview will display and operate any compliant document resource anywhere within the navigation hierarchy without activating another application. Document resources will be able to optionally implement a 'hyperlinking' interface, which will allow any non-HTML-based document to provide 'WWW-like' hyper-jumps within the navigation universe. By integrating the Windows Explorer and the Internet Explorer, and allowing activation of applications within the Explorer, Microsoft will make it possible for the user to stay within the Explorer for his or her entire computing experience. This will allow Microsoft to leverage the universally acclaimed 'easy-to-use' user interface of the World Wide Web.



Behind the scenes, Microsoft have completely rewritten Internet Explorer 3.0 to use components. Unlike its monolithic cousin, Internet Explorer 3.0 is simply an Active Document container. It's a very small program which hosts the HTML Active Document, provides a display area and handles message dispatch for user selection and input. Most of the work we associate with the browser will be performed by the HTML Active Document, a COM-based DLL supplied by Microsoft. Segregating the container from the document provides great flexibility. Software developers can now provide their own value-added container frame, within which they can host the Microsoft HTML Active Document, providing full Internet Explorer 3.0 functionality within their own product (subject to licensing from Microsoft, of course). An Active Document container can also host any other COM-based DLLs containing Active Document, providing more ways to present information (both network-based and local), including proprietary file formats. Microsoft will definitely provide Active Documents support for all their Microsoft Office applications, which means that the user will be able to navigate down to and display any Office application documents within the navigation universe without leaving the Internet Explorer.

As an additional bonus, the Microsoft Internet Explorer 3.0 container will also support a new HyperLinking interface, which will allow non-HTML documents (i.e. a Microsoft Word document) to contain a hyper-reference (graphical or text) to other local or networked documents or resources. Any application vendors can implement the

Active Document interfaces, allowing their application to run in-frame within Explorer 3.0 or other Active Document containers. To the user, this means that even non web pages can contain functional hyper-links to other web or non-web pages. Users will be able to jump in and out of web pages and legacy documents without having to distinguish between them.

The ActiveX Stack

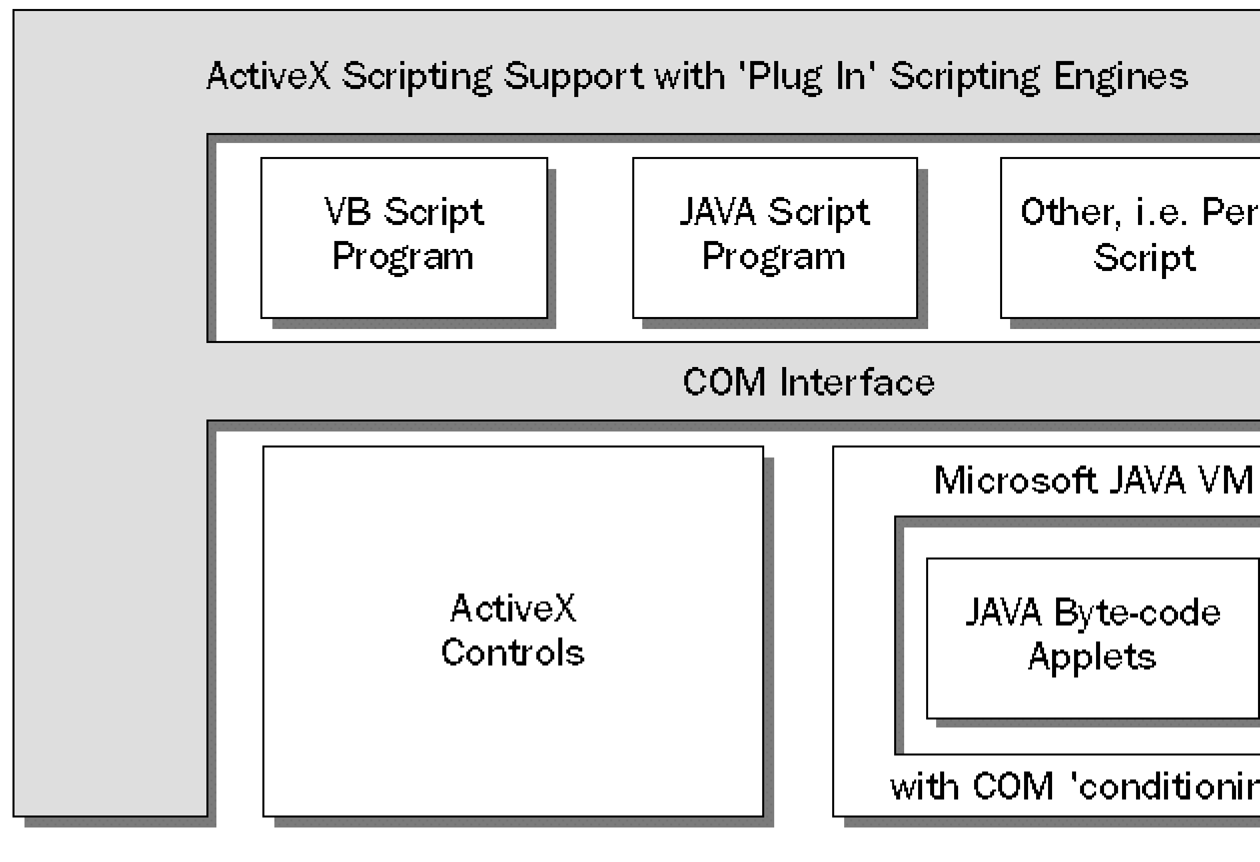
Another very important component layer underneath the new Internet Explorer 3.0 is formed by the ActiveX scripting engines and ActiveX controls, which are actually existing technologies wrapped up in a new name.

Many Visual Basic and Borland Delphi programmers are already familiar with the VBX and OCX controls. These are drop-in pieces of preprogrammed functionality which the hosting program can make use of by setting properties and calling methods. What makes them unlike common subroutine libraries is their object-like capability to handle their own initialization, memory allocation and user interface requirements. Microsoft has renamed OCX as ActiveX controls, and fully enabled them for distributed computing. All OCX (ActiveX) controls are COM objects: functional subsystems that are coded to Microsoft Component Object Model standards.

There's an undercurrent of excitement running through the community of web page authors, sparked by Sunsoft's Java technology. **Java** is a programming language similar to C++, but with the unique property that Java programs may be embedded into a web page. When a user accesses a web page with an embedded Java program, the browser (i.e. Netscape or Internet Explorer) will activate and execute the program. Embedded Java programs are frequently called **applets**. These applets enable processing on the client without incurring network traffic or server access. To ensure security, a set of trusted services on the client side provide limited run-time support for Java applets.

To participate in this excitement, Microsoft has licensed Sunsoft's Java technology for use in their Internet Explorer and ActiveX products. Microsoft also plan to provide a Visual Java development tool, code named Jakarta. At the same time as supporting Java, Microsoft will also support and promote an alternative scripting technology based on their Visual Basic product. This scripting technology will be called **VBScript**, and Microsoft promises to make the source code for the scripting engine freely available for any purpose. To put more control on Java applets, Microsoft have heavily wrapped the script execution environment with an ActiveX access layer. Java applets will be executing within a virtual machine hosted within the ActiveX scripting engine. The same virtual machine will expose Java classes as COM objects. This will make it possible for non-Java script programs (i.e. VBScript) to call and make use of Java objects. At the same time, run-time support will be provided for Java applets to make use of any COM objects, including ActiveX controls. Adding even more flexibility, Microsoft has defined interfaces which will make it possible for third parties to totally replace the ActiveX scripting engine. This will allow for Internet scripting using languages other than VBScript.

ActiveX controls can already be utilized by a very large set of development tools, including Visual C++, Visual Basic, Borland Delphi, Borland C++ , PowerBuilder, etc. Add to this the suite of up-coming web page authoring tools, and the ActiveX scripting support and you have a real winner. Web page authors can drag-and-drop a variety of functional pieces onto their web page, write simple script language programs to coordinate their functions and immediately publish their creation on the Web. Internet/intranet application developers can easily use the large pool of available ActiveX controls to create the client side of their application.



Some work needs to be done to modify the existing architecture for OLE controls into an Internet savvy family of ActiveX controls. The major constraint here is the bandwidth available for Internet access. Today, when an OLE control is activated by the hosting application, its code and associated data are quickly brought into memory from the disk or over a high speed LAN. The OLE control is immediately ready for action. ActiveX controls, on the other hand, must frequently be brought over from an Internet server through very slow connections. For the near future, these connections will be largely restricted to analog modems working at about 28.8kbps, so the actual foot-print of the ActiveX controls must be minimized to reduce the download time. Furthermore, a scheme must be developed to allow the user to feel that the web page or application is still responsive while the code and/or the data is being downloaded. To make all this possible, Microsoft will be promoting an updated ActiveX control specification, currently known as **OCX96**, as well as a new object name resolution mechanism called **Asynchronous Moniker**.

> *Current OLE control technology depends on some rather hefty run-time libraries (MFC and C++ run-times) having been installed on the user's machine. In the world of web users, there's no guarantee at all that these run-time DLLs will in fact be there, even if the user's machine is running an MS Windows. So ActiveX controls must be built with a certain degree of stand-alone capability.*

Underneath the scripting support and ActiveX controls, Microsoft need to provide services to locate network resources and to retrieve and store network information. To unify the methods of locating resources (i.e. servers, files, printers, information items) in the navigation universe, Microsoft have provided a URL (Universal Resource Locator) moniker layer. This is an enhancement of OLE monikers and will allow URLs to be used to locate

networking resources. A URL is a string such as "http://www.microsoft.com/" which contains a protocol specification, and a 'path' to the machine and resource identification.

Once a resource is located in the navigation universe, what the application can do with it will depend on the application, as well as the protocol through which the resource can be accessed. This is called a **binding**. Initially, Microsoft will support the standard set of HTTP, FTP and Gopher bindings, as well as providing for addition of any application specific binding and/or protocols. The support for HTTP, FTP and Gopher protocols will be provided through the WinINET extension to the Win32 API.

The ActiveX Server Architecture

On the server, side, Microsoft is attempting to maximize the leverage of its existing Windows NT server product and the BackOffice suite of server products. Microsoft has made its Internet Information Server 1.0 (IIS) widely available for free over the Internet. IIS contains Microsoft's web server.

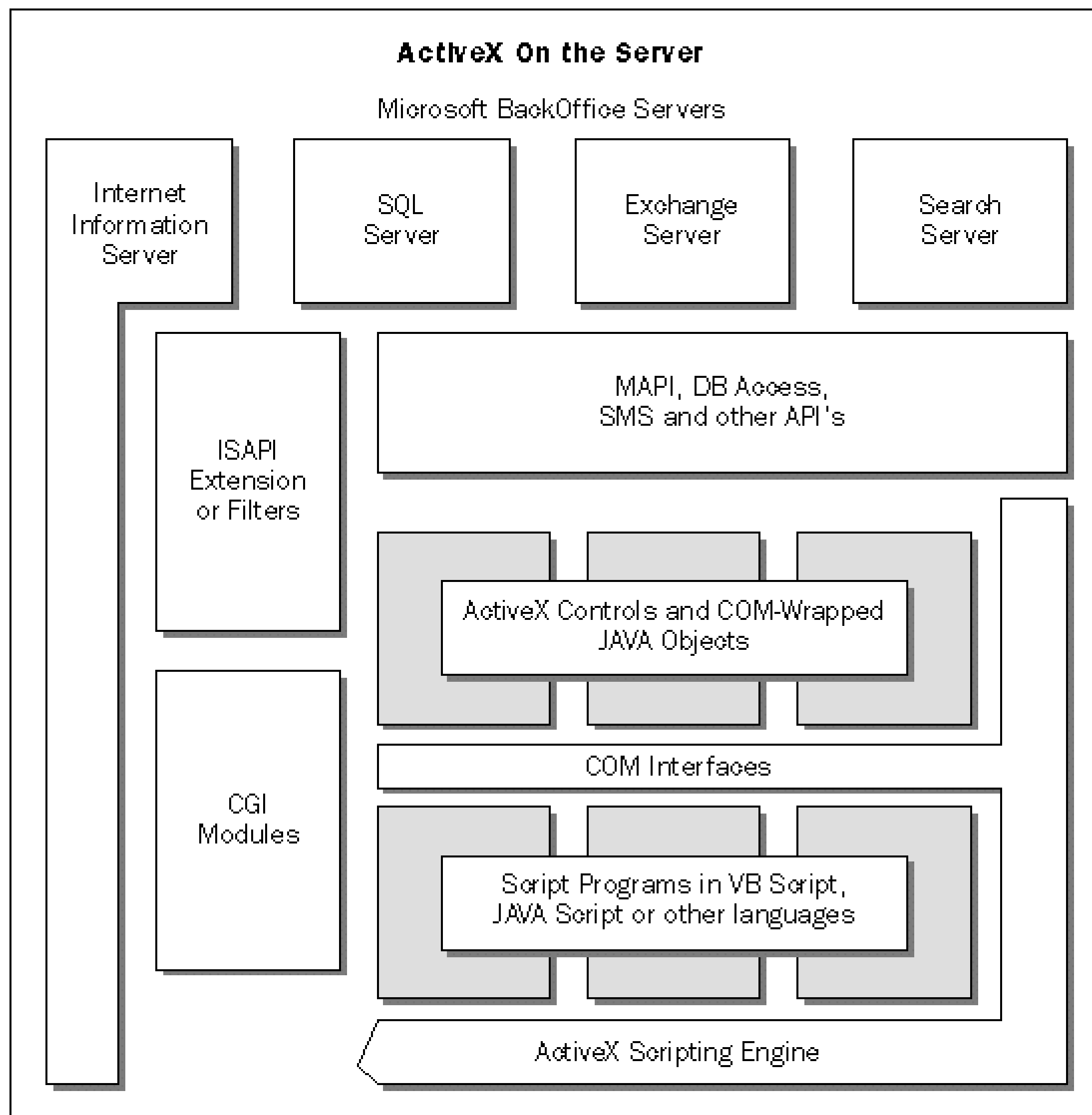
To allow the development of Internet and intranet-based applications within IIS, Microsoft have implemented the Common Gateway Interface (CGI) standard, as well as its own Internet Server Application Programming Interface (ISAPI). Both interfaces allow for information collected from a web site or a browser to be passed into an external program for processing. The program may also return information formatted by HTML to be displayed on the client browser. Having evolved from a UNIX-centric environment, most current day Internet/intranet server applications make use of CGI. The key difference between CGI and ISAPI is that CGI requires the spawning of a new process for each request handled by CGI, whereas ISAPI runs in-process for multiple requests. ISAPI accomplishes this through the use of custom dynamic-linked libraries and Windows NT's multithreading support.

Essentially, the BackOffice suite consists of several data management servers, tailored for large corporations. The SNA Server manages the connection to legacy host systems, the Exchange Server manages a corporation's internal and external messaging, the System Management Server manages a corporation's networked-based assets and the SQL Server manages all relationally structured data. As such, these servers already answer some very compelling and proven business needs.

Deploying these servers over the Internet will be straightforward, since they can operate over TCP/IP networks. To this end, Microsoft have announced the availability of PPPT (Point to Point Protocol Tunneling) to allow BackOffice users to create virtual private networks over a public Internet connection. This means that unmodified BackOffice clients can now have secure access to the BackOffice servers over the Internet.

However, in some instances, organizations need have control over publishing information managed by the BackOffice servers as web pages. This is currently done by writing an ISAPI or CGI application which accesses the BackOffice servers via their corresponding APIs (i.e. MAPI for Exchange Server, ODBC for SQL Server, etc.). However, to make this process easier, especially for the upcoming ActiveX scripts and ActiveX controls, new connector and proxy components will be introduced, which will allow IIS to directly access information managed across the BackOffice suite. A version of the Internet Database Connector is supplied with the IIS to allow for the dynamic creation of web pages containing live data from an ODBC source, without coding.

To simplify the implementation of a server Internet/intranet application, Microsoft is determined to move the ActiveX run-time stack over to the server. By positioning ActiveX behind Windows NT server's encryption, security and authentication layers, Microsoft have provided these capabilities to services written using server ActiveX.



The availability of the ActiveX stack on the server will mean that the development of Windows NT server applications is no longer the restricted domain of guru C/C++ programmers. Instead, it will be open those using Visual Basic, Delphi, Java, PERL or any other supported scripting language. The ability to use many existing ActiveX controls on the server side will make a good base of prefabricated functionality available to anyone developing a server application from day one. There is no penalty for using interpreted scripting languages in the server application, as long as sufficiently powerful hardware is available.

The Holy Grail: Distributed COM

Since 1992, Microsoft have had a vision to enable distributed computing through their Component Object Model, allowing COM-based objects to be instantiated and utilized transparently over a network. Back in the Windows NT Professional Developer's conference of 1993, Microsoft demonstrated distributed COM and circulated sample code. At that point, distributed COM existed only in the barest of prototypical forms. Unsolved issues germane to widespread use of distributed COM, such as provision of adequate directory services, licensing mechanisms, and platform independence, remained.

Once again, in 1996, Microsoft has reintroduced distributed COM as a powerful futuristic enhancement to much of the ActiveX architecture. The same unsolved issues regarding the widespread use of distributed COM remain, but some progress has been made on some of them, and certain trends that have occurred in technology and the market have rendered them moot for a large proportion of users. The increasing dominance of Win32 platforms is rendering platform-independence irrelevant for a larger and larger segment of the market, and it has dawned on many that distributed object invocation mechanisms, such as distributed COM, can be quite useful, even in the absence of directory services, when coupled with HTTP as the mechanism for first establishing connections.

The fundamentals of distributed COM are simple: a intermediate proxy layer in the COM support run time will farm requests for COM interfaces and/or methods transparently over the network.

With distributed COM in place, the scripting engine on the client side can make use of server COM objects directly, bypassing an entire stack of software layers. By the same token, server Active Documents can communicate with the client user interface host directly.

Distributed COM can only be of limited use until several very tough technical and business issues have been addressed.

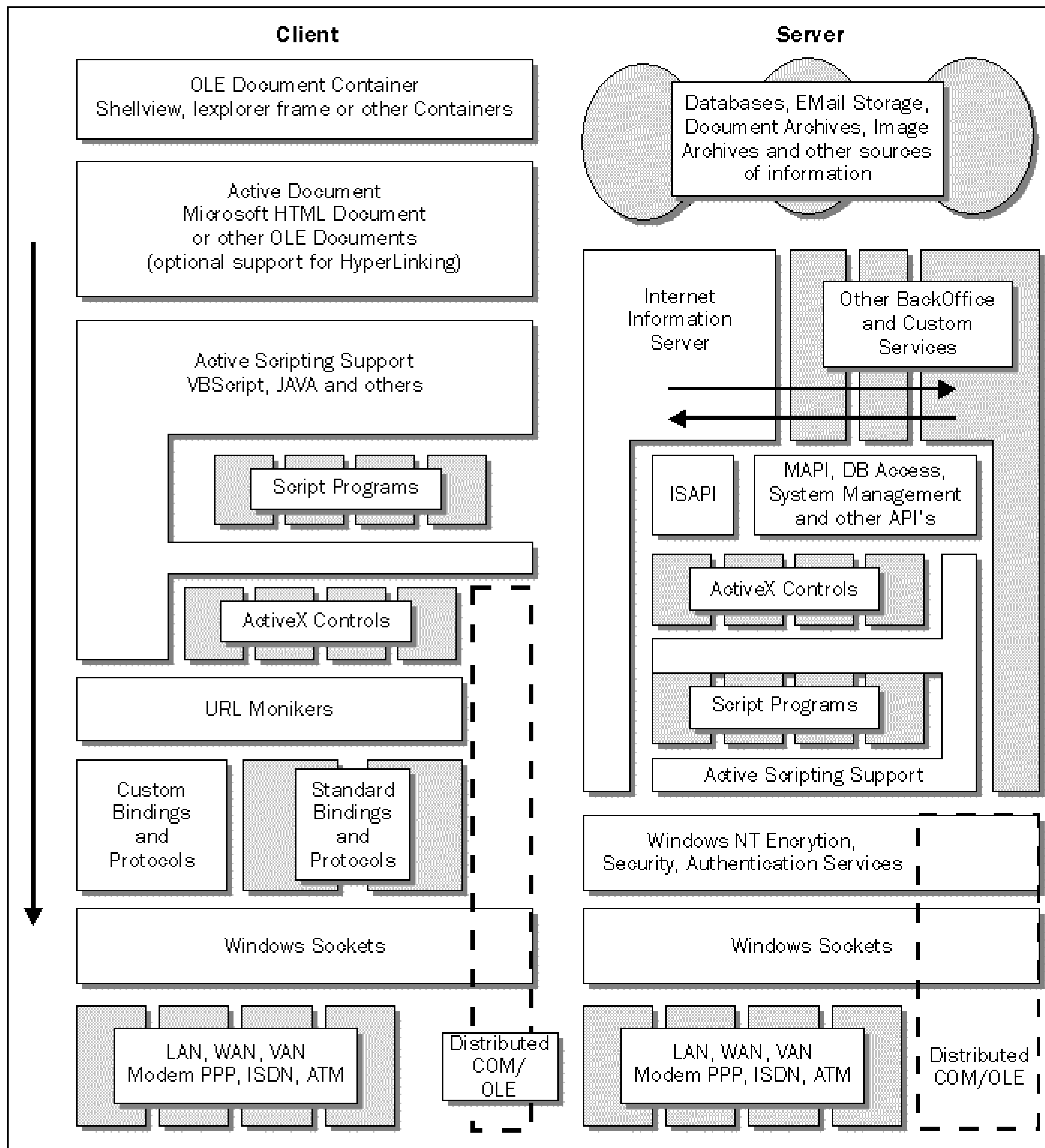
First and foremost, a directory service which manages objects and the ownership of information in the entire navigation space (from the local machine to right across the Internet) needs to be in place. Without this, it would take days to configure a moderate-sized application that makes use of hundreds of distributed objects each time it's used. Unfortunately, such a 'universal' directory service is not likely to be established in the near future, due to the fragmented landscape of existing network naming services, each with their own approach to solving this complex problem.

Secondly, component object authors would like to be rewarded for their effort. A solid network-wide licensing manager needs to be in place. There is currently no accepted business model for a truly distributed component world. How will providers of component objects over the Internet be paid? How will usage of the object be metered? These are very hard questions to answer.

Last but certainly not least, as the Internet is composed of millions of heterogeneous computers, a component object which is platform- dependent is a very poor solution.

Putting it Altogether

Let's put all the pieces that we have discussed so far together, and see what the new ActiveX architecture looks like.



Internet Explorer 3.0

Now it's time to take a quick look at the new ActiveX architecture in action. What happens when we run the Internet Explorer 3.0 executable? If we compare the alpha copy of the Internet Explorer 3.0 with Internet Explorer 2.0, we

can see the different EXEs and DLLs that are loaded:

Internet Explorer 3.0	Internet Explorer 2.0
Iexplore.exe	Ie20.exe
Mshtml.dll	Secbasic.dll
Hlink.dll	Secur32.dll
Oleaut32.dll	Msnsspc.dll
Urlmon.dll	Mcm.dll
Wininet.dll	Treenvcl.dll
Urlcache.dll	Svcprop.dll
Shdocvw.dll	Moscc.dll
	Tapi32.dll
	Mosmisc.dll
	Moscl.dll
	Sec_sspi.dll
Version.dll	Version.dll
Mpr.dll	Mpr.dll
Crtdll.dll	Crtdll.dll
Url.dll	Url.dll
Ole32.dll	Ole32.dll
Msvcrt20.dll	Msvcrt20.dll
Wsock32.dll	Wsock32.dll
Winmm.dll	Winmm.dll
Rpcrt4.dll	Rpcrt4.dll
Comctl32.dll	Comctl32.dll
Sage.dll	Sage.dll
Shell32.dll	Shell32.dll
User32.dll	User32.dll
Gdi32.dll	Gdi32.dll
Advapi32.dll	Advapi32.dll
Kernel32.dll	Kernel32.dll

In the Internet Explorer 2.0, besides the standard support DLLs, such as `Shell32.dll`, `Gdi32.dll`, etc., we can see the Spyglass heritage in the `Mos*.dll` series. What's more interesting is how Internet Explorer 3.0 has been divided into components. Gone is the `Mos*.dll` series of DLLs. Instead we have:

File	Contains
Iexplore.exe	Internet Explorer 3.0 main frame window.
Shdocvw.dll	ActiveX Shell Explorer Control; provides access to all the functionality of Internet Explorer through COM; also acts as Active Document container for the `Mshtml.dll` Active Document.
Mshtml.dll	ActiveX HTML Viewing Active Document Server; implements HTTP retrieval, decoding, and display.
Wininet.dll	ActiveX WinINET Win32 Extension.
Hlink.dll	ActiveX Hyperlinking Support.
Urlmon.dll	ActiveX URL Moniker Implementation.
Urlcache.dll	ActiveX System wide URL caching support.

If we cross reference this with ActiveX diagram, we can see that the client component architecture is well in place, even in this Alpha release of Internet Explorer 3.0.

Future updates will merge the Windows Explorer with the Internet Explorer by replacing the `Iexplore.exe` hosting frame with an `Explorer.exe` hosting frame. Later, Windows 95 and Windows NT versions will swap the shell for the hosting frame, which means that the browser becomes the user interface to the operating system and the user will never need to exit the browser (universal client) again.

Other Active Internet Fronts

The Microsoft Internet strategy encompasses much more than the ActiveX client/server architecture. It also represents Microsoft's complete shift in focus and attitude towards the Internet. The following sections will describe some of the other on-going Internet focused developments.

Microsoft have announced an entire new suite of products which use the ActiveX technology, including software, such as Frontpage, for creating web pages and managing web sites, various Internet Assistants (for Word, Excel, PowerPoint, etc.) and Internet Studio. Also becoming available are ActiveX development tools, such as Visual Basic Scripting Edition (for creating, debugging and testing of VB script programs) and Visual J++ (formerly called Jakarta, for creating, debugging and testing of Java applets).

Adding to the BackOffice Suite will be a new Merchant Server for doing electronic commerce transactions, a high-end public Internet Server, code-named Normandy, a 'next generation' Directory Server , and a Search Server. All in all, BackOffice will become a potent suite of interoperating servers to handle any Internet or intranet site requirements.

To protect against malicious programs hidden in the form of downloadable ActiveX components, a 'code signing' mechanism will be available. Using digital signatures, a hierarchy of 'trusted certificate agencies' and public key encryption, it will ascertain the origin of a piece of code (i.e. an ActiveX control) and make sure that it hasn't been tampered with during transmission.

To facilitate live collaboration over the Internet/intranet, Microsoft have announced the ActiveX Conferencing API and standards. Anticipating the shift from 2D graphical web pages to the future 3D web worlds, enabled by new standards such as VRML, Microsoft has fortified support for 3D graphics in its operating system through the Direct 3D API initiative. Adapting to demands from the public, hungry for better net-based multimedia support, Microsoft has introduced ActiveMovie for standard-based video playback, and has begun to implement quality of service (QOS) support into Windows NT.

QOS allows networking applications to negotiate, reserve and guarantee bandwidth. This is a fundamental feature for high speed network interconnect technologies, such as ATM (asynchronous transfer mode). Most new networking hardware, the new proposed version of the Internet Protocol and the new version of the Winsock standard, all have provision for QOS. By providing QOS support within the operating system, Microsoft have finally guaranteed the quality of service between client and server applications. Finally, the users will be able to get crystal clear, non breaking, constant bit-rate video or audio (as good as the TV and radio that we are used to) delivered directly over the Internet.

Active Documents and Doc Objects

A detailed and complete analysis of the ActiveX architecture is beyond the scope of this book, so, instead, we'll concentrate on a part of ActiveX, called Active Documents or Doc Objects.

Despite the understandable excitement over the Internet/intranet and new ActiveX-based technologies, we must acknowledge that abandoning the huge software base that has been written without these technologies in mind is unrealistic. It would be wonderful, though, to be able to 'activate' these applications and make them net-savvy with just a little work! Microsoft have tried to address these issues with Doc Objects. Essentially, Doc Objects allow conventional applications to participate in ActiveX after just a few relatively simple modifications.

In real terms, this means that someone using an ActiveX-compatible browser to surf web pages can view a link containing a document from a legacy application. The user can view the document in any format (standard or proprietary) and directly from within the browser, so they won't be aware of the transition. To the user, it appears as another web page, potentially providing new ways of interacting.

All of a sudden, you can publish information on your intranet in proprietary formats that your users are already familiar with. In our case, we're going to take the CIS from the WinINET chapter and retrofit it to become an ActiveX document server. The corporate intranet users in our hypothetical corporation can migrate to Microsoft Internet Explorer as their standard desktop browser, and still enjoy the simple custom behaviors provided by the CIS.

Beneath the Hood

How do Doc Objects work?

The support provided by Doc Objects is divided between the container and the server. A Doc Object container is typically a 'value added frame'. Good examples are the Microsoft Office Binder provided with Microsoft Office is such a container, the Internet Explorer frame provided with Internet Explorer 3.0, or the Explorer frame (combining local system navigation with net navigation) to be provided with Microsoft's Internet update.

Doc Object servers are similar to older OLE in-place editing document servers. As a matter of fact, existing in-place editing servers can be readily modified to be Doc Object servers.

When an Doc Object container hosts a Doc Object server, it gives control of the entire client area of the container to the server. This is unlike the OLE embedding/in-place editing server, which only gets a small window in the document being edited. Menu merging is performed, and the toolbar provided by the server is posted, in addition to the container's main toolbar. This is shown in the figure below, when our CIS is hosted inside the Internet Explorer 3.0.

Another difference between an OLE embedding/in-place editing server and a Doc Object server is that the user doesn't need to double-click the Doc Object to make it active. The Doc Object is always in-place active, as soon as it is given control of the container's client area.

Finally, since the Doc Object takes over the entire editing area of the container, it is also responsible for controlling the layout of the printed output. Printing is done through a print method, and no metafile is necessary, since it's no longer just a portion of a larger output, as with conventional OLE embedding.

The OLE COM Foundation

It would be very difficult to understand Doc Objects without some fundamental understanding of OLE and COM. We've covered the foundation of COM in previous chapters, so I'll give a very brief recap just to refresh our memory on the subject matter.

OLE is an umbrella term for a family of object-oriented distributed computing technologies from Microsoft. COM is the common tie between the OLE technologies. It stands for **Common Object Model** and is a binary object model. This binary model is language-independent and soon will be operating system and platform-independent. The model specifies the life cycle of an object, prescribes how objects expose their capabilities, how they interact with each

other, and how object interactions can span process and machine boundaries.

COM objects expose their functionality and interact through a set of agreed-upon **interfaces**. An interface is a collection of functions and has a globally unique ID (GUID) associated with it. Each new version of an interface has to have a new GUID. The GUID is a very, very large number which guarantees that each and every interface defined by anyone on earth at anytime past or present will be unique.

The most basic interface is called **`IUnknown`**. All COM interfaces are based on it and, more importantly, all objects must implement it. **`IUnknown`** implements three functions **`AddRef()`**, **`Release()`**, and **`QueryInterface()`**. **`AddRef()`** and **`Release()`** are used in object life cycle control via reference counting. Whenever an object gives out an interface pointer, **`AddRef()`** is called to increment a reference counter. When the calling component no longer needs the interface, **`Release()`** is called to decrement the reference counter. It is generally safe to unload the object from memory if the reference count goes to zero.

`QueryInterface()` is used to determine the interfaces that an object will support. When **`QueryInterface()`** is called with the unique ID of a particular interface, the object will return a pointer the requested interface if it is supported. Since every COM interface is based on **`IUnknown`**, and **`IUnknown`** implements the **`QueryInterface()`** function, we can get an interface pointer to any other interface that an object supports. Once a pointer to an interface is obtained, any function provided by that interface can be called. This allows a component to gain access to all the capabilities of an object given just one single interface pointer. Helper functions which instantiate or locate objects only have to return one interface pointer.

This is enough of an OLE refresher for us to proceed with an examination of the interfaces implementing Doc Object server functionality.

The New Interfaces

Assuming that you're starting with an application which supports OLE document embedding and in-place activation, the application would have already implemented the following required interfaces:

```
IPersistStorage
IPersistFile
IOleObject
IDataObject
IOleInPlaceObject
IOleInPlaceActiveObject
```

To support Doc Object, we need to modify the implementation of some of these interfaces. In addition, the following new Doc Object interfaces must be implemented:

```
IOleDocument
IOleDocumentView
IOleDocumentCommandTarget
IPrint
```

The last two interfaces, **`IOleDocumentCommandTarget`** and **`IPrint`**, are required only if you need to pass command events from the container to the document, or you need to support printing through the container, respectively.

We'll be taking a detailed look at how we implement these interfaces when we add Doc Object support to the CIS example from the previous chapter. But first, let's take a brief look at our overall approach to the conversion..

An Easy MFC Approach

Coding an Active Document server from scratch is not for the faint of heart, due to the large number of interfaces and functions that must be supported. Fortunately, this is seldom necessary, as MFC already provides most of the capabilities through some well-tested base class code from Microsoft.

An application developed using MFC, generated via the Visual C++ App Wizard with full OLE server support would have default support for many of the required interfaces. The following lists the required OLE document embedding and in-place activation interfaces automatically supported by MFC-generated code, as well as the MFC classes which implement the required interface:

Required Interface	MFC Class Implementing the Interface
IOleObject	**COleServerDoc**
IDataObject	**COleServerDoc**
IPersistStorage	**COleServerDoc**
IPersistFile	**COleLinkingDoc**
IOleInPlaceObject	**COleServerDoc**
IOleInPlaceActiveObject	**COleServerDoc**

You can find all of the above interface declarations in the interface maps for the **COleServerDoc** and **COleLinkingDoc** classes in the source file `Msdev/Mfc/Include/Afxole.h` supplied with Visual C++.

To complete the embedding/in-place editing support, we need to customize:

document serialization
the redrawing of the in-place frame for the activated and non-activated states
the management of the negotiation for the size and location of the in-place frame
menu and toolbar merging.

We'll be looking at specific implementation details in the next section with our CIS application.

With Visual C++ 4.1, Microsoft have provided a sample program called BINDSCRB which adds Active Document support to the sample SCRIBBLE program used in the MFC tutorial. It's called BINDSCRB because it makes SCRIBBLE compatible with the BINDER component of Office 95, which is, essentially, the very first Active Document container. Put another way, Microsoft have borrowed the 'embedded document with full control of container's client area' technology from the Office 95 BINDER application as a template for Active Document. Besides being a good sample of Active Document server programming, it also provides valuable base classes which can be used in any other MFC application wishing to use this technology. We'll be borrowing heavily from BINSCRB in our CIS conversion.

Specifically, BINDSCRB provides a **COleDocObjectServer** class to replace the **COleServerDoc** class; a **COleDocObjectServerItem** class to replace the **COleServerItem** class; and a **CDocObjectIPFrameWnd** class to replace the **COleIPFrameWnd** class. These classes are derived from the classes that they replace, and they make the required modifications to the base class behavior, making them compatible with Active Document. BINDSCRB also implements the new Doc Object interfaces required. Here's a list of these required interfaces, and the new class responsible for implementing it:

Required Interface	New Class Implementing the Interface
IoleDocument	**ColeDocObjectServer**
IoleDocumentView	**ColeDocObjectServer**
IoleDocumentCommandTarget	**ColeDocObjectServer**

IPrint	ColeDocObjectServer

The declaration for the above interface map is in the `Binddoc.h` file from the BINDSCRB project. We'll examine how these new classes modify their base classes in the next section, when we add Active Document support to CIS.

Activate the CIS

We now proceed to retrofit our CIS program to include support for Active Document. The Active Document container we will be using to test the CIS will be the alpha version of Internet Explorer 3.0. As an Active Document hosting container, there are a number of limitations associated with this release:

It cannot support multiple views per document.
It can only view one document at a time.
It does not make use of the **`IPrint`** interface.
It doesn't fully support the usage of the **`IOleDocumentCommandTarget`** interface.

To cope with these limitations, our implementation will only provide a single view per document. For the **`IPrint`** and the **`IOleDocumentCommandTarget`** interface, we'll provide stubs, but not implement the details.

The procedures documented here are not unique to CIS. They can be used to add Active Document support to any MFC-based application.

We'll now examine how the new CIS will be implementing each of the required interfaces and their associated functions through the BINDSCRB supplied classes. Along the way, we'll get a detailed look at the new interfaces.

You may want to follow along by examining the source code provided on the accompanying CD-ROM. Be forewarned, though, that this project consists of a *very large* body of code, and navigating it may not be trivial! However, if you fasten your seatbelts and stay with me, you'll soon see that we're actually doing very little work to the original CIS during the conversion, thanks to the tremendous BINDSCRB code pool. We'll save the actual conversion process until the end. After the exercise, please join me in a silent prayer for the MFC crew to absorb this pool of code into the basic application framework ASAP!

Note that in the description following, interface functions are specified in MIDL notation where applicable. Compared to regular C/C++ syntax, this notation provides additional information for function arguments, differentiating input arguments from output arguments.

IOleDocument Interface

The document/view model of Doc Objects is almost identical to that of MFC. A view is a window into the data contained within the document. A document can be associated with many views. Each view can have a different display presentation of the data contained in a document. The **`IOleDocument`** interface provides the container with capability to create and manage views, as well as obtaining information on the document. **`IOleDocument`** has the following functions:

`CreateView()`
`GetDocMiscStatus()`
`EnumViews()`

CreateView() Function

```
HRESULT IOleDocument::CreateView([in] IOleInPlaceSite *pIPSite,
   [in] IStream *pstm, [in] DWORD dwReserved,
   [out] IOleDocumentView **ppView)
```

This function allows the document container to either request that the server creates a new view (when **`pstm`** is **`NULL`**), or loads a saved view from the **`pstm`** stream.

`pIPSite` is a pointer to the container's in-place site to be associated with the new view. This interface is provided by

the container's view site object. A view site object is responsible for managing the container's display area for a specific view. If **pIPSite** is **NULL**, the container must later call the view's **IOleDocumentView::SetInPlaceSite()** to set the site.

The new view's **IOleDocumentView** interface is returned in **ppView**. Once the view has been created successfully, the container can call **IOleDocumentView::Show()** or **IOleDocumentView::UIActivate()** to make it visible.

CIS supports only a single view, through `Binddcmt.cpp` from BINDSCRB, which it implements **CreateView()** via:

```
STDMETHODIMP CDocObjectServerDoc::XOleDocument::CreateView(
   LPOLEINPLACESITE pipsite, LPSTREAM pstm,
   DWORD dwReserved, LPOLEDOCUMENTVIEW* ppview)
{
...
   if (dwReserved == 0 && pThis->m_pDocSite != NULL)
   {
      // if view site is already
      // set, fail.
      if (pThis->m_pViewSite == NULL)
      {
         LPOLEDOCUMENTVIEW pView =
            (LPOLEDOCUMENTVIEW)pThis->GetInterface(&IID_IOleDocumentView);
         ASSERT(pView != NULL);

         // Set the site for the view
         hr = pView->SetInPlaceSite(pipsite);
         if (hr == NOERROR)
         {
            // Return the IOleDocumentView pointer
            pView->AddRef();
            *ppview = pView;
         }

         // If a saved view stream is provided
         // restore it
         if (pstm)
            hr = pView->ApplyViewState(pstm);
      }
   }
...
}
```

GetDocMiscStatus() Function

```
HRESULT IOleDocument::GetDocMiscStatus([out] DWORD *pdwStatus)
```

This provides properties of the Doc Object in a **DWORD pdwStatus**. It's implemented to let the container know what the Doc Object is capable of.

The bit values are:

Constant	Value	Description
DOCMISC_CANCREATEMULTIPLEVIEWS	**1**	The document can support multiple views.
DOCMISC_SUPPORTCOMPLEXRECTANGLES	**2**	The document can support complex rectangles.
DOCMISC_CANTOPENEDIT	**4**	The document has minimal or no user interface and cannot honor open edit.
DOCMISC_NOFILESUPPORT	**8**	The document supports **IpersistStorage**,

but not **IpersistFile**.

CIS doesn't create multiple views, doesn't support complex rectangles, supports open editing and saving or loading from a file. The bit-mask for this configuration is 0, so CIS/BINDSCRB implements **GetDocMiscStatus()**, again in `Binddcmt.cpp`, via:

```
STDMETHODIMP CDocObjectServerDoc::XOleDocument::GetDocMiscStatus(
   LPDWORD pdwStatus)
{
...
   *pdwStatus = 0;

   return NOERROR;
}
```

EnumViews() Function

```
HRESULT IOleDocument::EnumViews([out] IEnumOleDocumentViews **ppEnum,
   [out] IOleDocumentView **ppView)
```

The container calls this function to enumerate all the views provided by the Doc Object. **ppEnum** is a pointer to the **IEnumOleDocumentViews** interface of an enumerator object. If the document doesn't support multiple views, the pointer to a single view is returned via ***ppView**.

Since the CIS Doc Object will support only a single view, it calls **QueryInterface()** to get a pointer to the **IOleDocumentView** interface and returns it in ***ppView**:

```
STDMETHODIMP CDocObjectServerDoc::XOleDocument::EnumViews(
   LPENUMOLEDOCUMENTVIEWS* ppEnumView, LPOLEDOCUMENTVIEW* ppView)
{
...

   // We only support a single view
   *ppEnumView = NULL;
   HRESULT hr = QueryInterface(IID_IOleDocumentView, (LPVOID*)ppView);

   return hr;
}
```

IOleDocumentView Interface

The **IOleDocumentView** interface provides various functions for the container to manipulate, manage, or activate a particular view of a Doc Object. **IOleDocumentView** has the following functions:

SetInPlaceSite()
GetInPlaceSite()
GetDocument()
SetRect()
GetRect()
SetRectComplex()
Show()
UIActivate()
Open()
CloseView()
SaveViewState()
ApplyViewState()
Clone()

SetInPlaceSite() and GetInPlaceSite() Functions

```
HRESULT IOleDocumentView::SetInPlaceSite([in] IOleInPlaceSite *pIPSite)

HRESULT IOleDocumentView::GetInPlaceSite([out] IOleInPlaceSite **ppIPSite)
```

These functions allows a container to associate a view site object with a view, or returns a pointer to the currently associated site for the view.

For **`SetInPlaceSite()`**, if the view already has a site associated, the view must call **`IOleInPlaceObject::InPlaceDeactivate()`**, and then release the old site. The view should remember the site passed in **`pIPSite`**.

For **`GetInPlaceSite()`**, the view should return either a pointer to the site most recently set by the container or **`NULL`**. If a site is returned, the view must remember to do an **`AddRef()`** on the interface.

The implementation of these functions is straight forward in CIS, as they are almost a word-for-word translation of the above statements.

The implementation of **`SetInPlaceSite()`** is:

```
STDMETHODIMP CDocObjectServerDoc::XOleDocumentView::SetInPlaceSite(
    LPOLEINPLACESITE pIPSite)
{
    // if currently inplace active, deactivate
    if (pThis->IsInPlaceActive())
        pThis->m_xOleInPlaceObject.InPlaceDeactivate();

    // release the view site pointer
    if (pThis->m_pViewSite)
        pThis->m_pViewSite->Release();

    // store the site and addref
    pThis->m_pViewSite = pIPSite;
    if (pThis->m_pViewSite != NULL)
        pThis->m_pViewSite->AddRef();

    return NOERROR;
}
```

The implementation of **`GetInPlaceSite()`** is:

```
STDMETHODIMP CDocObjectServerDoc::XOleDocumentView::GetInPlaceSite(
    LPOLEINPLACESITE* ppIPSite)
{
...
    if (pThis->m_pViewSite)
        pThis->m_pViewSite->AddRef();
    *ppIPSite = pThis->m_pViewSite;

    return NOERROR;
}
```

GetDocument() Function

```
HRESULT IOleDocumentView::GetDocument([out] IUnknown **ppunk)
```

This function lets the container obtain a pointer to the **`IUnknown`** interface of the view's associated document, returned in **`ppunk`**. Having retrieved this interface, the container can obtain any other interface provided by the document object.

CIS/BINDSRB implements this interface simply through **`QueryInterface()`** as:

```
STDMETHODIMP CDocObjectServerDoc::XOleDocumentView::GetDocument(
LPUNKNOWN* ppUnk)
{
...
    HRESULT hr = pThis->m_xOleDocument.QueryInterface(IID_IUnknown,
        (LPVOID*)ppUnk);
    ASSERT(*ppUnk != NULL);

    return hr;
}
```

Note that **QueryInterface()** does the **AddRef()** for us.

SetRect() and GetRect() Functions

```
[input_sync] HRESULT IOleDocumentView::SetRect([in] LPRECT prcView)

HRESULT IOleDocumentView::GetRect([out] LPRECT prcView)
```

These functions enable the setting and retrieval of the viewport coordinates of the view window. When it receives this call, the view should resize itself to the new coordinates.

CIS/BINDSCRB implements **SetRect()** by delegating to a member function called **OnSetItemRects()**:

```
STDMETHODIMP CDocObjectServerDoc::XOleDocumentView::SetRect(
    LPRECT lprcView)
{
...
        pThis->OnSetItemRects(lprcView, lprcView);
        hr = NOERROR;
...
}
```

The handling of **OnSetItemRects()** in the **CCISOLE2Doc** class stores the coordinates for resizing of the buttons in CIS to fit the view window upon the next time the screen is re-drawn.

GetRect() returns the viewport coordinates in the client coordinates of the view window.

CIS/BINDSCRB delegates the function to the **GetItemPosition()** function:

```
STDMETHODIMP CDocObjectServerDoc::XOleDocumentView::GetRect(
    LPRECT lprcView)
{
...
    pThis->GetItemPosition(lprcView);
    return NOERROR;
}
```

This actually ends up calling the **GetItemPosition()** function of the base **COleServerDoc** class. The rectangle returned is the coordinates of the only item being edited in place, which coincides with the dimension of the view.

SetRectComplex() Function

```
[input_sync] HRESULT IOleDocumentView::SetRectComplex([in] LPRECT prcView,
   [in] LPRECT prcHScroll, [in] LPRECT prcVScroll, [in] LPRECT prcSizeBox)
```

This function is used by containers to set the coordinates of the view port, the scroll bars, and the size box. Simpler Doc Objects which don't support complex rectangles can return **E_NOTIMPL**.

If the Doc Object supports complex rectangles, it should immediately resize to the rectangle specified by **prcView**, and put the scrollbars and size box at **prcHScroll**, **prcVScroll**, and **prcSizeBox** respectively.

CIS/BINDSCRB doesn't support complex rectangles, so it simply returns **E_NOTIMPL**.

Show() Function

```
HRESULT IOleDocumentView ::Show ([in] BOOL fShow)
```

A view supporting this function should show or hide itself according to the value of the **fShow** flag. If the **fShow** flag is **TRUE**, the Doc Object should in-place activate without UI activation. Then it should show the view window. If **fShow** is **FALSE**, the Doc Object should call **IOleDocumentView::UIActivate(FALSE)** and then hide the view.

UIActivate() Function

```
HRESULT IOleDocumentView::UIActivate([in] BOOL fUIActivate)
```

The Doc Object should either activate or de-activate its user interface, according to the state of the **fUIActivate** flag. UI elements include the menus, toolbars and accelerators. Merging menus and toolbars should also be done here. If **fUIActivate** flag is **FALSE**, the Doc Object would typically make a call to **IOleInPlaceObject::UIDeactivate()** to deactivate the active object and its user interface elements.

CIS/BINDSCRB implements this function by delegating to the **OnActivateView()** function. **OnActivateView()** is a complex function which creates the in-place frame window as a child of the site obtained from the container, performs in-place activation and negotiation for merging of menus and toolbars. See the sample code file `Bindview.cpp` for its implementation. In future versions of the MFC, this functionality will become an integral part of the application framework, eliminating the need to recode this functionality for every new Doc Object server.

Open() Function

```
HRESULT IOleDocumentView::Open(void)
```

This function is equivalent to the **IOleObject::DoVerb(OLEIVERB_OPEN)**. The container wishes to open the view in an independent pop-up window. If the Doc Object doesn't support this 'open in a separate window' functionality, it should return **E_NOTIMPL**. It should also ensure that the **IOleDocument::GetMiscStatus()** returns a bit-mask with **DOCMISC_CANTOPENEDIT** set, as well as ensuring the bits in the DocObject registry key are set appropriately.

If the document supports its own frame, the implementation typically begins with a call to **IOleInPlaceObject::InPlaceDeactivate()**. Then the view displays the separate window.

While the separate window is displayed, the view holds on to all the interface pointers, as well as a pointer to the container's in-place site. If the container calls **IOleDocumentView::Show()** during this time, the view should hide and show the pop-up window accordingly. When the user closes the separate window, **IOleInPlaceSite::OnInPlaceActivate()** should be called to give the container a chance to optionally activate the UI.

Should it wish to close the view permanently, the container will officially call **IOleDocumentView::CloseView()**.

CIS/BINDSCRB doesn't support the use of a separate window, so it simply returns **E_NOTIMPL**.

CloseView() Function

```
HRESULT IOleDocumentView::CloseView([in] DWORD dwReserved)
```

The container calls this function to tell the view to close down completely. The view should hide itself by calling **IOleDocumentView::Show (FALSE)** and then release the site retained by calling **IOleDocumentView::SetInPlaceSite(NULL)**. A container will typically call this before it deletes the view.

CIS/BINDSCRB implements **CloseView()** exactly as described above:

```
STDMETHODIMP CDocObjectServerDoc::XOleDocumentView::CloseView(
   DWORD dwReserved)
{
...
   // Call IOleDocumentView::Show(FALSE) to hide the view
   Show(FALSE);

   // Call IOleDocumentView::SetInPlaceSite(NULL) to deactivate the object
   HRESULT hr = SetInPlaceSite(NULL);

   return hr;
}
```

SaveViewState() and ApplyViewState() Function

```
HRESULT IOleDocumentView::SaveViewState([in] IStream *pstm)

HRESULT IOleDocumentView::ApplyViewState([in] IStream *pstm)
```

These functions allow the container to ask the view to save or restore its state from the supplied stream **pstm**. The state can contain any attributes specific to the view, including appearance, type, size, zoom factor, insertion point, etc. When it's writing to the stream, the view must abide by the rules of **IPersistStream** and write its CLSID as the first element in the stream. When it's reading from the stream, the view must do its own validation. Typically, the container calls **SaveViewState()** before it deactivates the view, and uses **ApplyViewState()** after it has instantiated a new view.

CIS/BINDSCRB returns **E_NOTIMPL** for **SaveViewState()**, but does provide an implementation for **ApplyViewState()**. The **ApplyViewState()** implementation uses a **CArchive** to load the state and delegates to an **OnApplyViewState()** function. However, no state is archived or restored at this time. Due to the work -in-progress nature of the BINDSCRB sample, we can see this asymmetric behavior in `Bindview.cpp`.

```
STDMETHODIMP CDocObjectServerDoc::XOleDocumentView::ApplyViewState(
    LPSTREAM pstm)
{
...
    HRESULT hr = NOERROR;

    // Attach the stream to an MFC file object
    COleStreamFile file;
    file.Attach(pstm);
    CFileException fe;

    // load it with CArchive
    CArchive loadArchive(&file, CArchive::load |
        CArchive::bNoFlushOnDelete);
    TRY
    {
        pThis->OnApplyViewState(loadArchive);
        loadArchive.Close();
```

```
            file.Detach();
        }
        CATCH(COleException, pOE)
        {
            hr = pOE->m_sc;
        }
        AND_CATCH_ALL(e)
        {
            hr = E_UNEXPECTED;
        }
        END_CATCH_ALL

        return hr;
    }
```

Clone() Function

```
RESULT IOleDocumentView::Clone([in] IOleInPlaceSite *pIPSiteNew,
   [out] IOleDocumentView **ppViewNew)
```

The container will call this function when it needs to create a clone of the current view object on a new viewport and view site with the same state and context. This is very similar to the **IOleDocument::CreateView()** function and is often used to support the File/New... menu function supported by the container.

pIPSiteNew contains the in-place site to be associated with the view, and the new view is returned in **ppViewNew**.

CIS/BINDSCRB supports a single view only and does not support cloning so it returns **E_NOTIMPL**.

IOleDocumentCommandTarget Interface

The **IOleDocumentCommandTarget** interface provides the container with a way to query a Doc Object server for support of specific commands, as well as dispatching those commands to be executed by the Doc Object server. The interface is designed to be bidirectional and extensible. Sending of commands from the Doc Object to the container is supported by the container's implementation of the **IOleDocumentCommandTarget** interface. We can define new commands in the future, using the same dispatch mechanism. Each command is deemed to belong to a command group represented by a GUID.

The default command group is identified by **GUID=NULL** and contains the following standard Office 95 commands (recall that the BINDSRIB sample makes good o'SCRIBBLE Office 95 Binder compatible):

Command ID	**Description**
OLECMDID_OPEN	File Open
OLECMDID_NEW	File New
OLECMDID_SAVE	File Save
OLECMDID_SAVEAS	File Save As
OLECMDID_SAVECOPYAS	File Save Copy As
OLECMDID_PRINT	File Print
OLECMDID_PRINTPREVIEW	File Print Preview
OLECMDID_PAGESETUP	File Page Setup
OLECMDID_SPELL	Tools Spelling
OLECMDID_PROPERTIES	File Properties
OLECMDID_CUT	Edit Cut
OLECMDID_COPY	Edit Copy

OLECMDID_PASTE	Edit Paste
OLECMDID_PASTESPECIAL	Edit Paste Special
OLECMDID_UNDO	Edit Undo
OLECMDID_REDO	Edit Redo
OLECMDID_SELECTALL	Edit Select All
OLECMDID_CLEARSELECTION	Edit Clear
OLECMDID_ZOOM	View Zoom.
OLECMDID_GETZOOMRANGE	Get zoom range associated with the view.

The two functions in the **IOleCommandTarget** interface are:

QueryStatus()
Exec()

QueryStatus() Function

```
[input_sync] HRESULT IOleCommandTarget ::QueryStatus(
   [unique][in] const GUID *pguidCmdGroup, [in] ULONG cCmds,
   [in,out][size_is(cCmds)] OLECMD *prgCmds,
   [unique][in,out] OLECMDTEXT *pCmdText);
```

This is an optional interface. The container calls this function to query the Doc Object for the support of various commands (or vice versa in the other direction) belonging to a command group **pguidCmdGroup**. An array of **OLECMD** structures is passed in via **prgCmds**. **cCmds** indicates the number of commands contained in the array. The container is interested in the support for the commands specified in the **cmdID** structure member of this array. The Doc Object's implementation should fill in the **cmdf** field of each command with a bit-mask consisting of a combination of the following **OLECMDF** enumeration:

Bit-mask Constant	Description
OLECMDF_SUPPORTED	The Doc Object supports this command.
OLECMDF_ENABLED	This command is available and enabled.
OLECMDF_LATCHED	This command is a toggle and is on.
OLECMDF_NINCHED	This command is a toggle but no state information is available.

The **OLECMD** structure for **prgCmds** is defined to contain the following members:

Type	Member	Description
ULONG	**cmdID**	ID for command
DWORD	**Cmdf**	flags for command
enum	**OLECMDTEXTF**	Indicates what a command target object should store into the **OLECMDTEXT** structure

We can call this function to obtain the name or status bar text for a command. To determine whether the container wanted this service, check the **OLECMDTEXTF** member of the elements of the **prgCmds** array. They can contain values from the following **OLECMDTEXTF** enumeration:

Constant	Description
OLECMDTEXTF_NONE	No text return is required.
OLECMDTEXTF_NAME	The name of command should be returned.
OLECMDTEXTF_STATUS	The status string of command should be returned.

If the **OLECMDTEXTF** member contains **OLECMDTEXF_NAME**, the object should place the name of the command into the **Rgwz** member of **pCmdText**. If the member contains **OLECMDTEXTF_STATUS**, the implementation should place the status line text into the **Rgwz** member. The **cwActual** member should be updated to reflect the actual length.

The **pCmdText** parameter is of an **OLECMDTEXT** structure, defined to contain the following members. Only the first command tagged in the **prgCmds** array will have text returned.

Type	Member	Description
DWORD	**cmdtextf**	Indicates what the implementation should do.
ULONG	**CwActual**	The number of characters filled in by the implementation.
ULONG	**CwBuf**	The input size of the buffer.
wchar_t	**Rgwz**	The input buffer to hold the returned text.

CIS/BINDSCRB doesn't support this optional interface. The function is simply stubbed in the `Bindtarg.cpp`.

Exec() Function

```
HRESULT IOleCommandTarget ::Exec([unique][in] const GUID *pguidCmdGroup,
   [in] DWORD nCmdID, [in] DWORD nCmdExecOpt,
   [unique][in] VARIANTARG *pvaIn, [unique][in,out] VARIANTARG *pvaOut)
```

This function is a caller's request for the object to perform a command or display help. The command is specified by the **pguidCmdGroup** and **nCmdID** combination. It is the caller's responsibility to ensure that the command is supported by the object via an **IOleCommandTarget::QueryStatus()** call. The action for the object to take is specified via the **nCmdExecOpt** parameter. This parameter can contain values from the **OLECMDEXECOPT** enumeration:

Value	Description
OLECMDEXECOPT_PROMPTUSER	Execute the command after taking user input.
OLECMDEXECOPT_DONTPROMPTUSER	Execute the command silently; no user input is required.
OLECMDEXECOPT_DODEFAULT	Let the called object decide whether to prompt user or not.
OLECMDEXECOPT_SHOWHELP	Show the help for the command.

If the command to be executed requires input parameters, the **pvarIn** parameter will point to a **VARIANTARG** variable containing one or more input values. If the command returns one or more values, the appropriate **VARIANTARG** variable must be created and returned by the implementation as **pvarOut**.

CIS/BINDSCRB doesn't support this optional function, so the code in `Bindtarg.cpp` provides a stub.

IPrint Interface

IPrint is an optional interface to be implemented by views. It provides the container with a command to print the

view (via a function, instead of as a request to create a metafile, as in the case with embedded OLE items), a command for setting the initial page number when the container is printing several objects or documents together and a command to retrieve information associated with printing from the view.

The **IPrint** interface has three functions:

> **SetInitialPageNum()**
> **GetPageInfo()**
> **Print()**

SetInitialPageNum() Function

```
HRESULT IPrint::SetInitialPageNum([in] LONG nFirstPage)
```

The container calls this function to set the first page to print for this document. **nFirstPage** is a **LONG** and may actually be negative, to indicate an offset from the currently displayed page. Not all Doc Objects would implement this capability, as it may not make sense for certain printing applications.

CIS/BINDSCRB does not support the **IPrint** optional interface.

GetPageInfo() Function

```
HRESULT IPrint::GetPageInfo([out] LONG *pnFirstPage, [out] LONG *pcPages)
```

The container calls this function to obtain information about the pages to be printed. If **pnFirstPage** is not **NULL** upon input, it will contain the page number of the first page to be printed upon return. **pcPages** will contain a count of all pages to be printed, as long as it is not **NULL** upon input.

Print() Function

```
HRESULT IPrint::Print([in] DWORD grfFlags, [in,out] DVTARGETDEVICE **pptd,
    [in,out] PAGESET **pppageset, [unique][in,out] STGMEDIUM *pstgmOptions,
    [in] IContinueCallback *pcallback, [in] LONG nFirstPage,
    [out] LONG *pcPagesPrinted, [out] LONG *pnLastPage)
```

The container calls this to print the view on the printer specified by the **pptd** parameter. **pptd** is a Win32 **DVTARGETDEVICE** structure. The implementation should check the **dmOrientation** field and attempt to honor the paper orientation during printing. Other information, such as paper size and number of copies, can be obtained by the **DEVMODE** member. **grfFlags** can contain values from the **PRINTFLAG** enumeration:

Constant	Description
PRINTFLAG_MAYBOTHERUSER	Allows the view to interact with the user. No interaction is allowed if not set (i.e. must print completely in one shot with no user interface).
PRINTFLAG_PROMPTUSER	If the **PRINTFLAG_MAYBOTHERUSER** is set, this indicates to the view that it should prompt the user for print options using a print dialog.
PRINTFLAG_USERMAYCHANGEPRINTER	Allows the user to change printer when they are presented with a print options dialog. This option is only valid if both **PRINTFLAG_MAYBOTHERUSER** and **PRINTFLAG_PROMPTUSER** flags are set.
PRINTFLAG_RECOMPOSETODEVICE	Asks the view to recompose itself to the specified device. Otherwise, the view is free to compose base on any former

	device associations.
PRINTFLAG_PRINTTOFILE	Prints to the file named by the **portname** member of the **DVTARGETDEVICE** structure.

If either **PRINTFLAG_MAYBOTHERUSER** or **PRINTFLAG_PROMPTUSER** flags are specified, **pptd** and **ppPageSet** may be different upon return to the caller. **nFirstPage** specifies the page number of the first page to be printed and should be used to override any pre-set first page number.

pstgmOptions is a pointer to a serialized property sheet, specifying view-specific state information which the container should hold and pass back on subsequent calls to the **Print()** function. The caller doesn't know the format of this information, so can't examine or modify this structure. This parameter may be used to print the view with the same auxiliary settings or configurations on different printers. The parameter can be **NULL** if it is not needed.

ppPageSet contains a **PAGESET** structure which controls which pages are to be printed. The members of the **PAGESET** structure are:

Type	Member	Description
ULONG	**cbStruct**	The size of the structure in bytes (must align on 4 bytes boundary).
BOOL	**FoddPages**	Indicates that only odd-numbered pages in the set are to be printed.
BOOL	**FevenPages**	Indicates that only even-numbered pages in the set are to be printed.
PAGERANGE *	**RgPages**	Contains pairs of page range, which are sorted in increasing order and are not overlapping. Taken together, this list specifies the pages to print.
ULONG	**CpageRange**	The number of page-range pairs in **RgPages**.

The **RgPages** member of **PAGESET** is a **PAGERANGE** structure, containing a pair of page number. It is defined to be:

Type	Member	Description
LONG	**nFromPage**	The first page in this print range.
LONG	**nToPage**	The last page in this print range. If it is **PAGESET_TOLASTPAGE**, then all pages between **nFromPage** and the last legal page will be printed.

After the printing operation, when the **Print()** function returns, **pcPagesPrinter** contains the number of pages actually printed and **pnLastPage** returns the number of the last page printed.

The printing operation itself is blocking, with respect to the calling container. However, the implementation of **IPrint()** is required to periodically call the **IContinueCallBack::FContinuePrint()** function through the interface **pCallback** parameter. The guideline here is to consider the human response speed and poll **pCallback** frequent enough so that the user can cancel the printing without delay.

IContinueCallBack Interface

The **IContinueCallBack** interface is a generic interface for periodic callback, allowing for a lengthy blocking operation to be canceled. It is defined to contain two functions:

FContinue()
FContinuePrinting()

FContinuePrinting() is actually a specialization of **FContinue()**.

FContinue() Function

```
HRESULT IContinueCallback::FContinue(void)
```

This asks whether operation should continue. A return value of **S_OK** indicates it is all right to continue operation, while **S_FALSE** indicates that the operation should be aborted as soon as possible.

FContinuePrinting() Function

```
HRESULT IContinueCallback::FContinuePrinting(LONG cPagesPrinted, LONG nCurrentPage,
LPOLESTRING pszPrintStatus)
```

This asks caller (container) whether the blocking printing operation should be continued. It's used to allow the user to cancel the operation (as in the **FContinue()** function), as well as allowing the container to display status reporting user interface element to the user (i.e. page being printed).

cPagesPrinted should contains the total number of pages printed thus far. **nCurrentPage** contains the page number of the page being printed. Note that it is entirely possible to call **FContinuePrinting()** multiple times with the same **cPagesPrinted** and **nCurrentPage** values. This will depend on how long it takes to print a page. **pwszPrintStatus** are status reporting text messages passed to the caller. The caller may display these messages in a status box.

As in **FContinue()**, a return value of **S_OK** indicates that printing should continue. A **S_FALSE** or **E_UNEXPECTED** return value indicates that printing should be aborted as soon as possible.

Modifications to Embedding/In-place Editing Code

As well as implementing new Active Document interfaces, it is also necessary to modify the base OLE embedding/in-place editing code to support Active Document activation. The following will examine the modifications necessary, and shows how CIS/BINDSCRB implements these modifications.

IOleObject::SetClientSite() Modification

The original implementation of **`IOleObject::SetClientSite()`** must be modified to detect whether the site supplied from the container will support **`IOleDocumentSite`**. If it does, we must behave like an Active Document, otherwise, regular in-place server behavior will be appropriate.

CIS, through code taken from the file `Oleobjct.cpp` from the BINDSCRB sample, implements **`CDocObjectServerDoc::XDocOleObject::SetClientSite()`** as:

```
STDMETHODIMP CDocObjectServerDoc::XDocOleObject::SetClientSite(
    LPOLECLIENTSITE pClientSite)
{
...
    // Perform normal SetClientSite processing.
    hr = pThis->m_xOleObject.SetClientSite(pClientSite);

    // release old document site
    if (pThis->m_pDocSite != NULL)
    {
        pThis->m_pDocSite->Release();
        pThis->m_pDocSite = NULL;
    }

    // Check to see whether this object should act
    // as a document object by querying for
    // IOleDocumentSite, storing it in m_pDocSite
    if (pClientSite != NULL)
        hr = pClientSite->QueryInterface(IID_IOleDocumentSite,
            (LPVOID*)&pThis->m_pDocSite);
    return hr;
}
```

IOleObject::DoVerb() Modifications

The **`DoVerb()`** function of the **`IOleObject`** interface must be changed to support **`OLEIVERB_HIDE`**, **`OLEIVERB_OPEN`** and **`OLEIVERB_SHOW`** slightly differently.

For CIS, these are implemented in the **`CDocObjectServerItem`** class' **`OnHide()`**, **`OnOpen()`**, **`OnShow()`** function respectively.

An Active Document server should never receive **`OLEIVERB_HIDE`** while it's in-place activated (which is all the time, unless it supports 'open as a separate window' functionality). If this verb is received, we should report a problem. CIS implements this through `Binditem.cpp` from BINDSCRB, as:

```
void CDocObjectServerItem::OnHide()
{
   CDocObjectServerDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);

   if (pDoc->IsDocObject())
      AfxThrowOleException(OLEOBJ_E_INVALIDVERB);
   else
      COleServerItem::OnHide();
}
```

For **OLEIVERB_OPEN**, we need to activate the Active Document:

```
void CDocObjectServerItem::OnOpen()
{
   CDocObjectServerDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);

   if (pDoc->IsDocObject())
      pDoc->ActivateDocObject();
   else
      COleServerItem::OnOpen();
}
```

The **OLEIVERB_SHOW** handling is the same as **OLEIVERB_OPEN**:

```
void CDocObjectServerItem::OnShow()
{
   CDocObjectServerDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);

   if (pDoc->IsDocObject())
      pDoc->ActivateDocObject();
   else
      COleServerItem::OnShow();
}
```

The document performs **ActivateDocObject()** by asking the stored **IOleDocumentSite** from the container to activate the current view.

Modification to In-place Activation

The in-place activation code must be modified from normal OLE embedding/in-place activation by:

Drawing the view's scrollbar inside the rectangle instead of outside (remember we're taking over the entire client area).

Skipping the drawing of the hatched border or selectors.

Skipping the call to **IOleInPlaceSite::OnPosRectChange()**.

Doing nothing in **IOleObject::SetExtent()** calls.

For CIS/BINDSCRB, see the code in `Bindipfw.cpp`. For **OnRequestPositionChange()**, it does nothing:

```
void CDocObjectIPFrameWnd::OnRequestPositionChange(LPCRECT lpRect)
{
    CDocObjectServerDoc* pDoc = (CDocObjectServerDoc*)GetActiveDocument();
    ASSERT_VALID(pDoc);
    ASSERT(pDoc->IsKindOf(RUNTIME_CLASS(CDocObjectServerDoc)));

    // DocObjects don't need to generate OnPosRectChange calls, so we
    // just return if this is a DocObject.
    if (pDoc->IsDocObject())
        return;

    // The default behavior is to not affect the extent during the
    //  call to RequestPositionChange.  This results in consistent
    //  scaling behavior.
    pDoc->RequestPositionChange(lpRect);
}
```

See **RecalcLayout()** in `Bindipfw.cpp` for details of the detailed drawing adjustment coding. This code provide the skipping of hatched border draw and ensures that the scrollbars are drawn within the client rectangle.

Also, **IOleObject::SetExtent()** calls are ignored in **XDocOleObject::SetExtent()** in the Oleobjct.cpp file:

```
STDMETHODIMP CDocObjectServerDoc::XDocOleObject::SetExtent(
    DWORD dwDrawAspect, LPSIZEL lpsizel)
{
    METHOD_PROLOGUE_EX(CDocObjectServerDoc, DocOleObject)
    ASSERT_VALID(pThis);

   // DocObjects ignore SetExtent calls, so return E_FAIL
   if (pThis->IsDocObject())
      return E_FAIL;

   // Otherwise, just do the normal processing
   return pThis->m_xOleObject.SetExtent(dwDrawAspect, lpsizel);
}
```

Merge the Help Menus of Container and View

The help menus of both the container and the Active Document should be available to the user. This is accomplished via some fairly complex calculation and coding. Fortunately, the office binder support coding provided in Mfcbind.cpp performs it for us in the helper functions **MfcBinderMergeMenus()** and M**fcBinderUnMergeMenus()**. CIS makes use of this through the **CDocObjectIPFrameWnd::BuildSharedMenu()** and **CDocObjectIPFrameWnd::DestroySharedMenu()** functions in the Bindipfw.cpp file.

This complex, yet very regular operation could be easily absorbed into the MFC framework in a future version.

Adding New Registry Keys

To support Active Documents, the registry needs to be populated with new keys. These new keys include a **DocObject** key associated with the Active Document's CLSID:

HKEY_CLASSES_ROOT\<clsid>\DocObject=<flags>

The flags should be a 32 bit bit-mask, identical to the returned value from the Active Document's **IOleDocument::GetDocMiscStatus()** function. See the description earlier in this chapter for these flags.

Another key of vital importance is:

HKEY_CLASSES_ROOT\CLSID\<clsid>\DefaultExtension=<file extension>

For example, CIS has file extension **.CIS**.

Other keys that should be added if not already existing are:

HKEY_CLASSES_ROOT\CLSID\<clsid>\DocObject
HKEY_CLASSES_ROOT\CLSID\<clsid>\Printable

The **Printable** key should only be added if **IPrint** is supported.

The office binder helper function **MFCBinderUpdateRegistry()** in Mfcbind.cpp will perform all this for you. It should be called from the **InitInstance()** function in the **CWinApp** derived class.

Now that we have had BINDSCRB do almost all the tough parts for us, we must first add normal embedding and in-place editing support to CIS.

To add embedding and in-place editing support, we'll need to:

Add document serialization.
Add in-place activation support.

Adding Serialization Support

Adding serialization in CIS will allow us to save CIS documents to disk or compound document storage. This is a prerequisite for OLE embedding support.

Like many other applications, CIS doesn't naturally require documents to be saved in its normal operation. If you have an application, like CIS, which doesn't naturally support the File/Save... and File/Open... operations, you'll have to think of saving some operation states which will allow the user to pick up from where they left off. What we will be saving to the archive for CIS will be the most recently used Gopher locator (an ASCII string) . This locator will allow us to display the same selection of buttons when the file is reloaded.

First, we add the **CString m_curLocator** member variable to the **CCISOLE2Doc** class. This is the variable where we'll hold the most recently used Gopher locator. We initialize it in the constructor and update it every time we update the Gopher map:

```
BOOL CCISOLE2Doc::FillGopherMap(LPCTSTR Locator)
{
   // calling Locator must not be part of Map, otherwise
   // it will be deleted in the RemoveAll().  We'll make
   // it safe by storing a local copy
   CString myLocator = Locator;
   m_GopherMap.RemoveAll();

   CGopherItem * FirstItem = new CGopherItem;
   BOOL firstItemAvailable;

   m_curLocator = Locator;  // store the current locator

   // dual use, if locator not supplied, just grab the
   // default one from the specified host
   if ((NULL == Locator) || (0 == m_curLocator.GetLength()))
      firstItemAvailable = m_Gopher->GetFirstItem(*FirstItem);
   else
      firstItemAvailable = m_Gopher->GetFirstItem(*FirstItem,
        (LPCTSTR) myLocator);
...
```

Notice that we have also modified the check for a null value for the locator string. In fact, the locator may be pointing to a string with a **\0** as its first character. Only a check against the length will ensure that both cases are covered.

To add serialization support, we override the **Serialize()** function:

```
void CCISOLE2Doc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        ar << m_curLocator;
    }
    else
    {
        ar >> m_curLocator;
    }
}
```

This completes the serialization support. Next, we need to add in-place editing support.

Adding In-place Activation Support

The **`OnDraw()`** function of the **`CCISOLE2ServerItem`** is called when the embedded item is not active. This function draws to a metafile, which is used in drawing the inactive object. Since this is not strictly necessary for Active Document support (i.e. Active Documents are always in-place active), we'll do the minimum here.

```
BOOL CCISOLE2SrvrItem::OnDraw(CDC* pDC, CSize& rSize)
{
    CCISOLE2Doc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);


    //  All drawing takes place in the metafile device context
    CSize sizeDoc = pDoc->GetDocSize();

    pDC->SetMapMode(MM_ANISOTROPIC);

    sizeDoc.cy = -sizeDoc.cy;
    pDC->SetWindowExt(sizeDoc);
    pDC->SetWindowOrg(0,0);

    POSITION pos = pDoc->GetFirstViewPosition();
    CCISOLE2View * pView = (CCISOLE2View *) pDoc->GetNextView(pos);
    pDC->DPtoLP(&sizeDoc);
    CRect ab(0,0,sizeDoc.cx, sizeDoc.cy);
    ab.DeflateRect(3,3);
    CBrush aBrush;
    aBrush.CreateSolidBrush(RGB(255,0,0));
    pDC->FillRect( ab, &aBrush);

    return TRUE;
}
```

The **`OnGetExtent()`** function of the **`CCISOLE2SrvrItem`** class also needs to be overridden to return the item size in **`HIMETRIC`** units. Since our buttons will scale to whatever size the in-place window may be, we save the last known size of the window and return it to the container in this function.

```
BOOL CCISOLE2SrvrItem::OnGetExtent(DVASPECT dwDrawAspect, CSize& rSize)
{
    // Most applications, like this one, only handle drawing the
    // DVASPECT_CONTENT flag

    if (dwDrawAspect != DVASPECT_CONTENT)
        return CDocObjectServerItem::OnGetExtent(dwDrawAspect, rSize);

    // CCISOLE2SrvrItem::OnGetExtent is called to get the extent in
    //  HIMETRIC units of the entire item.  The default implementation
    //  here simply returns a hard-coded number of units.

    CCISOLE2Doc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);


    CSize sizeDoc = pDoc->GetDocSize();
    CClientDC dc(NULL);
    dc.SetMapMode(MM_ANISOTROPIC);

    dc.SetWindowExt(sizeDoc);
    dc.SetWindowOrg(0,0);

    rSize = sizeDoc;
    dc.DPtoHIMETRIC(&rSize);

    return TRUE;
}
```

The size of the document is stored in a **CSize** member variable, **m_DocSize**, in the **CCISOLE2Doc** class. It is initialized to 100 pixels-by-100 pixels in the constructor and refreshed each time the **OnSetItemRects()** call is made. This call is typically made indirectly by the container when the item is being resized.

```
void CCISOLE2Doc::OnSetItemRects(LPCRECT lpPosRect, LPCRECT lpClipRect)
{
    // call base class to change the size of the window
    CDocObjectServerDoc::OnSetItemRects(lpPosRect, lpClipRect);
    m_DocSize.cx = lpPosRect->right - lpPosRect->left;
    m_DocSize.cy = lpPosRect->bottom - lpPosRect->top;
}
```

If you have followed along with the modifications to the original CIS source code and have not yet copied the BINDSCRB files over, it will be a good time now to check and verify that the serialization is working and that embedding/in-place editing also works. For a container host, you can use either the CONTAIN sample container from the MFC samples, or any of the Office 95 applications. Select Insert/Object... and CISOLE document and watch as a mini-CIS appears embedded in the document.

The CIS buttons are still fully functional, even when they are embedded, and CIS will resize with the in-place frame. When CIS is deactivated, the buttons disappear from the client area. This is because we didn't implement the unnecessary metafile redraw for update of the server item. Reactivation of the embedded CIS object is achieved with a double click. This double click activation is very unnatural for CIS and makes it unsuitable as an embedded/in-place activated OLE server.

By now, you probably have a question on your mind.

Is that all we had to do to make CIS an Active Document server?

The answer is YES! The Binder support classes provided by the BINDSCRB sample does most of the tough work for us. When MFC absorbs it in a future version, all we'll have to do is to click a check box in AppWizard to get Active Document support.

Making Use of BINDSCRB Classes

If you're following along and have completed the modification of the original CIS code for OLE embedding/in-place activation, the next step is to add the new classes provided by the BINDSCRB sample software from the Visual C++ 4.1 CD-ROM. Copy the following files from the BINDSCRB sample to the modified CIS directory:

```
Binddcmt.cpp        Binddoc.cpp
Bindipfw.cpp        Binditem.cpp
Bindtarg.cpp        Bindview.cpp
Mfcbind.cpp         Oleobjct.cpp
Print.cpp           Binddoc.h
Bindipfw.h          Binditem.h
Mfcbind.h
```

Once the files are copied, add all the new `.cpp` files to the CIS project. From the developers studio's class view, select **CCISOLE2Doc** class, modify the base class to **CDocObjectServerDoc** instead of **COleServerDoc** (`Cisole2doc.h`). Select the **CCISOLE2SrvItem** class and modify the base class to **CDocObjectServerItem** , instead of **COleServerItem** (`Srvritem.h`). Select the **CInPlaceFrame** class and modify the base class to **CDocObjectIPFrameWnd** instead of **COleIPFrameWnd** (`Ipframe.h`).

Enabling the Doc Object

To continue the base class replacement, replace all references to **COleServerDoc** in `Cisole2doc.h` and

Cisole2doc.cpp to **CDocObjectServerDoc**. Replace all references to **CCISOLE2SrvItem** in Srvritem.h and Srvritem.cpp to **CDocObjectServerItem**. Replace all references to **COleIPFrameWnd** in Ipframe.h and Ipframe.cpp to **CDocObjectIPFrameWnd**. Finally, modify Cisole2doc.cpp, Srvritem.cpp and Ipframe.cpp to include Binddoc.h, Binditem.h and Bindipfw.h respectively. This completes the replacement of the base class.

The final step in converting CIS to an Active Document server involves adding a tag in the registry, telling the shell that CIS will support Active Document. You can do this by replacing the App Wizard-generated registry update line with a support function **MfcBinderUpdateRegistry()**. Do this in the **CCISOLE2App::InitInstance()** function in the Cisole2.cpp file.

```
BOOL CCISOLE2App::InitInstance()
{
...
   //BINDER:
   //    Binder objects have some special registry entries that
   //    MFC doesn't know about. Instead of calling
   //    COleTemplateServer::UpdateRegistry, call our special
   //    registration function
   MfcBinderUpdateRegistry(pDocTemplate, OAT_INPLACE_SERVER);
   //END_BINDER
...
}
```

That's it! Rebuild the CIS project, run it as a stand-alone application once to update the registry, and CIS is now ready to participate in the ActiveX party.

Testing with Doc Object Containers

We're now ready to test our new Active Document server, but first we must find some working Active Document containers. The two most widely available containers supporting full Active Document hosting are:

Office 95 Binder
Internet Explorer 3.0

Future versions of NCSA Mosaic and potentially other web browsers may also support Active Document hosting. For now, let's look at how CIS manages to plug-and-play with the above containers.

The Microsoft Office Binder

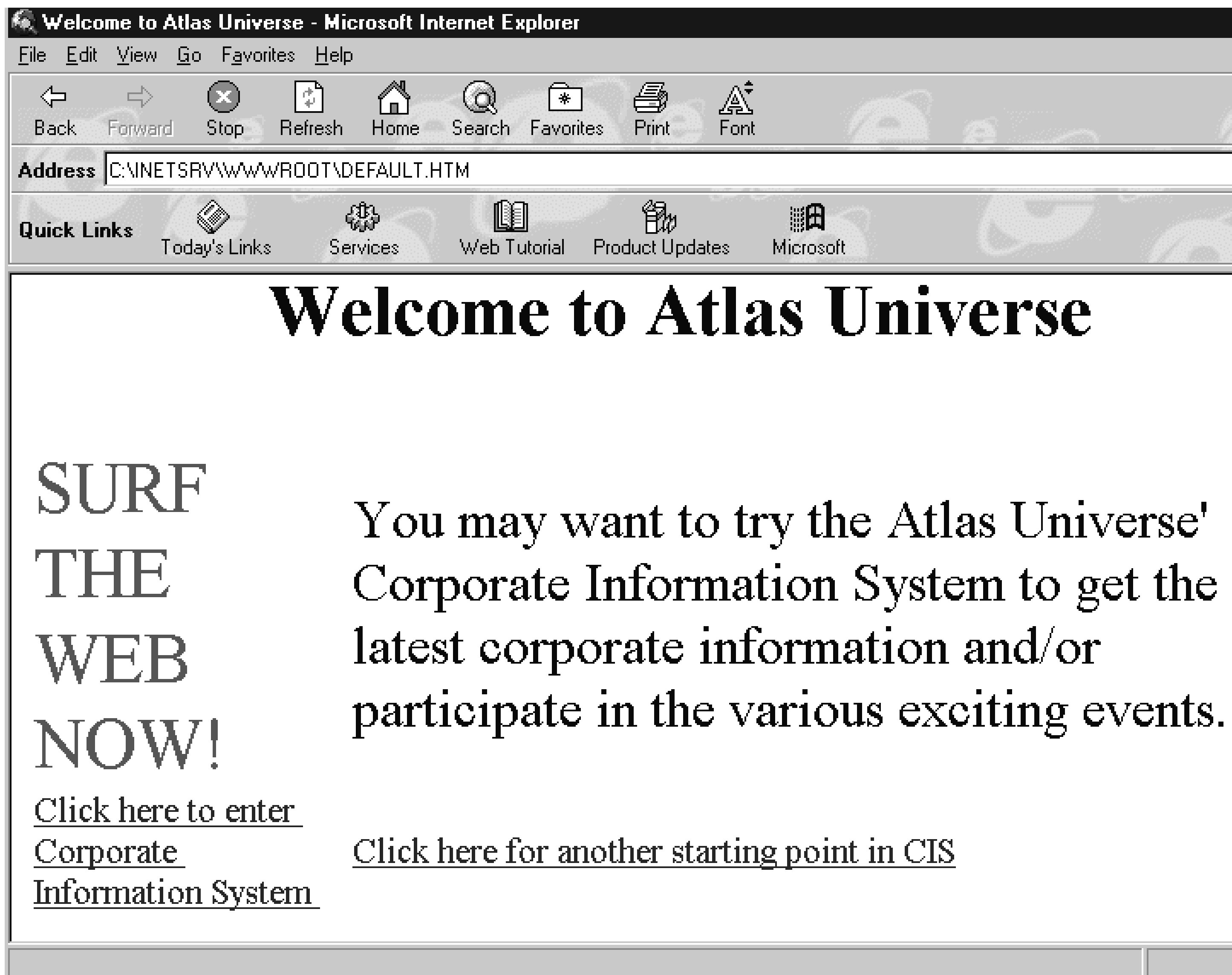
The Microsoft Office 95 binder is a Doc Object container which allows you to manipulate Office 95 documents in the same frame. You can create a binder consisting of a mixture of Word, Excel and Powerpoint documents. By clicking on the iconic representation of the document on the left frame, the document will be activated on the right half and you can use the native menus and toolbars to edit the document. The entire document can be saved as one big binder document. A binder can also be printed in one shot with running page numbers through all the subdocuments (as long as the **IPrint** interface is implemented by the Doc Object).

To add a CIS document to the binder, use the Section menu. Select either Add... or Add from File.... Add from File... will required a previously saved CIS file. If you use Add..., you'll have to select CISOLE object from the list of insertable objects. Once you have activated a CIS document, try adding an Excel and Word document. Try switching between them to see how the binder container switches between the Doc Objects.

The figure below shows CIS being hosted within the Office Binder. Also in the binder are a Word document and an Excel document.

Internet Explorer 3.0

Internet Explorer 3.0 is an Active Document container. The web browser functionality itself is provided by an MSHTML Active Document. It's easy for Internet Explorer to use another Active Document, like CIS. You can try out hosting of CIS within Internet Explorer. You can do this by typing in the name of a saved CIS file into the URL window (i.e. C:\TEST1.CIS), or enter it through a link in a standard HTML page.



Internet Explorer 3.0 will currently only run only on Windows NT 4.x or Windows 95. This means that we cannot test CIS on a single Windows NT machine using loopback unless it's running Windows NT 4.x. In my case, I've used a separate Windows 95 machine connected via LAN to the Windows NT development server.

As a matter of fact, the URL (**`file://C:\TEST1.CIS`** for the alpha version, or **`file:///C:\TEST1.CIS`** for the beta version) can be a URL hypertext link on a web page. When the user clicks on the hypertext link, CIS will become active within the Internet Explorer frame with the current location set to the one saved in the `Test.cis` file.

The backward and forward buttons of Internet Explorer will switch between the CIS 'page' and the recently traversed HTML web pages. Merged menus and the combined toolbar expose the 'return to start' button of CIS. Microsoft is currently in the process of defining the details of a hyperlinking interface. With this and the **`IOleCommandTarget`** interface, it will be possible to expose CIS pages as part of the navigation sequence. This will mean that the forward and backward buttons can also be used to traverse through a history list of CIS pages.

By interspersing web pages with links to existing Active Documents, legacy non-HTML documents can participate in Internet/intranet-based information publishing without any conversion. The user can navigate one 'navigation

space' consisting of the local machine, the local network, the local Intranet and the Internet. Users can search for information, store it, retrieve it, route it, send it and manipulate it, all within the same frame provided by a universal client. What we formerly called the browser, for all intents and purposes, has became the single, universal interface to the operating system.

OLE Server to Active Document Recap

In summary, we have performed the following procedure to convert an MFC application to an Active Document server:

- Add serialization support if not already existing.
- Add basic OLE embedding and in-place editing support, implementing minimal support for DocItem's metafile draw.
- Add Active Document base class files from the BINDSCRB project and inherit the application classes from these new base classes instead of standard MFC OLE.
- Replace the standard registry update function in the application's **`InitInstance()`** member function with **`MfcBinderUpdateRegistry()`**.

Microsoft intends to make this procedure even easier in Visual C++ 4.2 and beyond by providing Active Document support as an option when App Wizard generates the initial framework for an application.

Summary

In this chapter, we've discussed Microsoft's new Internet/intranet focus and have examined the technological components of its new ActiveX strategy. We have made a microscopic examination of the innards of the Active Document portion of the ActiveX client architecture and illustrated how it can facilitate a seamlessly integrated desktop-LAN-WAN-World experience for the user. We've also got our hands dirty by actually adding support for Active Document by enhancing the WinINET-based intranet application from the previous chapter. From this experience, we have seen how easy it is to add Active Document support to MFC-based applications.

To limit the scope of discussion within these two chapters, we had to make it extremely Microsoft-centric. Obviously, Microsoft don't conduct business in a vacuum. There are formidable opponents in the marketplace, including Netscape, IBM/Lotus, Novell and Sunsoft, each supplying their own set of Internet/intranet interfaces, object models and distributed computing infrastructure. With Microsoft at the helm, controlling the desktop computing marketplace, however, the ActiveX movement is guaranteed to have significant user support in the foreseeable future.

Consistent with my message for the WinINET chapter, the most important asset of a contemporary practicing software architect/developer/engineer is his/her familiarity with the building blocks currently available to solve problems in any specific problem domain. In this new wired world of distributed computing, many of the ActiveX building blocks described in this chapter will go on in history to become the fabric for many innovative applications.

What is WROX Press?

WROX Press is a computer book publisher which promotes a brand new concept - clear, jargon-free programming and database titles that fulfill your real demands. We publish for everyone, from the novice through to the experienced programmer. To ensure our books meet your needs, we carry out continuous research on all our titles. Through our dialog with you we can craft the book you really need.

We welcome suggestions and take all of them to heart - your input is paramount in creating the next great WROX title. Use the reply card inside the book or mail us at:

feedback@wrox.com
or
Compuserve 100063,2152

WROX Press Inc
2710 W. Touhy
Chicago
IL 60645
U.S.A.
Tel:(312) 465 3559
Fax: (312) 465 4063
On the World Wide Web:
http://www.wrox.com/