THE COMPLETE WINDOWS PROGRAMMING GUIDE

THE MFC TUTORIALS

The MFC stands for Microsoft Foundation Class. Here are some tutorials for MFC:

- The Complete MFC Guide (BIG GUIDE)
- Introduction to MFC with Visual C++
- New And cool in MFC
- MFC and Visual C++ 5

THE COMPLETE WINDOWS PROGRAMMING GUIDE

Advanced MFC Programming

Table of Contents

CHAPTER 5. Common Controls

CHAPTER 6. Dialog Box

CHAPTER 7. Common Dialog Boxes

CHAPTER 8. DC, Pen, Brush and Palette

CHAPTER 9. Font

CHAPTER 10. Bitmap

CHAPTER 12. Screen Capturing & Printing

CHAPTER 13. Adding Special Features to Application

CHAPTER 14. Views

CHAPTER 15. DDE

CHAPTER 16. Context Sensitive Help

THE COMPLETE WINDOWS
PROGRAMMING GUIDE

Introduction to MFC Programming with Visual C++ v5.x

Introduction to MFC

Visual C++ is much more than a compiler. It is a complete application development environment that, when used as intended, lets you fully exploit the object oriented nature of C++ to create professional Windows applications. In order to take advantage of these features, you need to understand the C++ programming language. If you have never used C++, please turn to the C++ tutorials in the C/C++ Tutorials page for an introduction. You must then understand the Microsoft Foundation Class (MFC) hierarchy. This class hierarchy encapsulates the user interface portion of the Windows API, and makes it significantly easier to create Windows applications in an object oriented way. This hierarchy is available for and compatible with all versions of Windows. The code you create in MFC is extremely portable.

These tutorials introduce the fundamental concepts and vocabulary behind MFC and event driven programming. In this tutorial you will enter, compile, and run a simple MFC program using Visual C++. Tutotial 2 provides a detailed explanation of the code used in Tutorial 1. Tutorial 3 discusses MFC controls and their customization. Tutorial 4 covers message maps, which let you handle events in MFC.

What is the Microsoft Foundations Class Library?

Let's say you want to create a Windows application. You might, for example, need to create a specialized text or drawing editor, or a program that finds files on a large hard disk, or an application that lets a user visualize the interrelationships in a big data set. Where do you begin?

A good starting place is the design of the user interface. First, decide what the user should be able to do with the program and then pick a set of user interface objects accordingly. The Windows user interface has a number of standard controls, such as buttons, menus, scroll bars, and lists, that are already familiar to Windows users. With this in mind, the programmer must choose a set of controls and decide how they should be arranged on screen. A time-honored procedure is to make a rough sketch of the proposed user interface (by tradition on a napkin or the back of an envelope) and play with the elements until they feel right. For small projects, or for the early prototyping phase of a larger project, this is sufficient.

The next step is to implement the code. When creating a program for any Windows platform, the programmer has two choices: C or C++. With C, the programmer codes at the level of the Windows Application Program Interface (API). This interface consists of a collection of hundreds of C functions described in the Window's API Reference books. For Window's NT, the API is typically referred to as the "Win32 API," to distinguish it from the original 16-bit API of lower-level Windows products like Windows 3.1.

Microsoft also provides a C++ library that sits on top of any of the Windows APIs and makes the programmer's job easier. Called the Microsoft Foundation Class library (MFC), this library's primary advantage is efficiency. It greatly reduces the amount of code that must be written to create a Windows program. It also provides all the advantages normally found in C++ programming, such as inheritance and encapsulation. MFC is portable, so that, for example, code created under Windows 3.1 can move to Windows NT or Windows 95 very easily. MFC is therefore the preferred method for developing Windows applications and will be used throughout these tutorials.

When you use MFC, you write code that creates the necessary user interface controls and customizes their appearance. You also write code that responds when the user manipulates these controls. For example, if the user clicks a button, you want to have code in place that responds appropriately. It is this sort of event-handling code that will form the bulk of any application. Once the application responds correctly to all of the available controls, it is finished.

You can see from this discussion that the creation of a Windows program is a straightforward process when using MFC. The goal of these tutorials is to fill in the details and to show the techniques you can use to create professional applications as quickly as possible. The Visual C++ application development environment is specifically tuned to MFC, so by learning MFC and Visual C++ together you can significantly increase your power as an application developer.

Windows Vocabulary

The vocabulary used to talk about user interface features and software development in Windows is basic but unique. Here we review a few definitions to make discussion easier for those who are new to the environment.

Windows applications use several standard user controls:

- Static text labels
- Push buttons
- List boxes
- Combo boxes (a more advanced form of list)
- Radio boxes

- Check boxes
- Editable text areas (single and multi-line)
- Scroll bars

You can create these controls either in code or through a "resource editor" that can create dialogs and the controls inside of them. In this set of tutorials we will examine how to create them in code. See the tutorials on the AppWizard and ClassWizard on the MFC Tutorials page for an introduction to the resource editor for dialogs.

Windows supports several types of application windows. A typical application will live inside a "frame window". A frame window is a fully featured main window that the user can re-size, minimize, maximize to fill the screen, and so on. Windows also supports two types of dialog boxes: modal and modeless. A modal dialog box, once on the screen, blocks input to the rest of the application until it is answered. A modeless dialog box can appear at the same time as the application and seems to "float above" it to keep from being overlaid.

Most simple Windows applications use a Single Document Interface, or SDI, frame. The Clock, PIF editor, and Notepad are examples of SDI applications. Windows also provides an organizing scheme called the Multiple Document Interface, or MDI for more complicated applications. The MDI system allows the user to view multiple documents at the same time within a single instance of an application. For example, a text editor might allow the user to open multiple files simultaneously. When implemented with MDI, the application presents a large application window that can hold multiple sub-windows, each containing a document. The single main menu is held by the main application window and it applies to the top-most window held within the MDI frame. Individual windows can be iconified or expanded as desired within the MDI frame, or the entire MDI frame can be minimized into a single icon on the desktop. The MDI interface gives the impression of a second desktop out on the desktop, and it goes a long way towards organizing and removing window clutter.

Each application that you create will use its own unique set of controls, its own menu structure, and its own dialog boxes. A great deal of the effort that goes into creating any good application interface lies in the choice and organization of these interface objects. Visual C++, along with its resource editors, makes the creation and customization of these interface objects extremely easy.

Event-driven Software and Vocabulary

All window-based GUIs contain the same basic elements and all operate in the same way. On screen the user sees a group of windows, each of which contains controls, icons, objects and such that are manipulated with the mouse or the keyboard. The interface objects seen by the user are the same from system to system: push buttons, scroll bars, icons, dialog boxes, pull down menus, etc. These interface objects all work the same way, although some have minor differences in their "look and feel." For example, scroll bars look slightly different as you move from Windows to the Mac to Motif, but they all do the same thing.

From a programmer's standpoint, the systems are all similar in concept, although they differ radically in their specifics. To create a GUI program, the programmer first puts all of the needed user interface controls into a window. For example, if the programmer is trying to create a simple program such as a Fahrenheit to Celsius converter, then the programmer selects user interface objects appropriate to the task and displays them on screen. In this example, the programmer might let the user enter a temperature in an editable text area, display the converted temperature in another un-editable text area, and let the user exit the program by clicking on a push-button labeled "quit".

As the user manipulates the application's controls, the program must respond appropriately. The responses are determined by the user's actions on the different controls using the mouse and the keyboard. Each user interface object on the screen will respond to events differently. For example, if the user clicks the Quit button, the button must update the screen appropriately, highlighting itself as necessary. Then the program must respond by quitting. Normally the button manages its appearance itself, and the program in some way receives a message from the button that says, "The quit button was pressed. Do something about it." The program responds by exiting.

Windows follows this same general pattern. In a typical application you will create a main window and place inside it different user interface controls. These controls are often referred to as child windows-each control is like a smaller and more specialized sub-window inside the main application window. As the application programmer, you manipulate the controls by sending messages via function calls, and they respond to user actions by sending messages back to your code.

If you have never done any "event-driven" programming, then all of this may seem foreign to you. However, the event-driven style of programming is easy to understand. The exact details depend on the system and the level at which you are interfacing with it, but the basic concepts are similar. In an event-driven interface, the application paints several (or many) user interface objects such as buttons, text areas, and menus onto the screen. Now the application waits-typically in a piece of code called an event loop-for the user to do something. The user can do anything to any of the objects on screen using either the mouse or the keyboard. The user might click one of the buttons, for example. The mouse click is called an event. Event driven systems define events for user actions such as mouse clicks and keystrokes, as well as for system activities such as screen updating.

At the lowest level of abstraction, you have to respond to each event in a fair amount of detail. This is the case when you are writing normal C code directly to the API. In such a scenario, you receive the mouse-click event in some sort of structure. Code in your event loop looks at different fields in the structure, determines which user interface object was affected, perhaps highlights the object in some way to give the user visual feedback, and then performs the appropriate action for that object and event. When there are many objects on the screen the application becomes very large. It can take quite a bit of code simply to figure out which object was clicked and what to do about it.

Fortunately, you can work at a much higher level of abstraction. In MFC, almost all these low-level implementation details are handled for you. If you want to place a user interface object on the screen, you create it with two lines of code. If the user clicks on a button, the button does everything needed to update its appearance on the screen and then calls a pre-arranged function in your program. This function contains the code that implements the appropriate action for the button. MFC handles all the details for you: You create the button and tell it about a specific handler function, and it calls your function when the user presses it. Tutorial 4 shows you how to handle events using message maps

An Example

One of the best ways to begin understanding the structure and style of a typical MFC program is to enter, compile, and run a small example. The listing below contains a simple "hello world" program. If this is the first time you've seen this sort of program, it probably will not make a lot of sense initially. Don't worry about that. We will examine the code in detail in the next tutorial. For now, the goal is to use the Visual C++ environment to create, compile and execute this simple program.

```
//hello.cpp

#include <afxwin.h>

// Declare the application class
class CHelloApp : public CWinApp
{
public:
  virtual BOOL InitInstance();
};

// Create an instance of the application class
CHelloApp HelloApp;

// Declare the main window class
class CHelloWindow : public CFrameWnd
{
  CStatic* cs;
public:
  CHelloWindow();
};

// The InitInstance function is called each
// time the application first executes.
BOOL CHelloApp::InitInstance()
{
  m_pMainWnd = new CHelloWindow();
  m_pMainWnd->ShowWindow(m_nCmdShow);
  m_pMainWnd->UpdateWindow();
  return TRUE;
}

// The constructor for the window class
CHelloWindow::CHelloWindow()
{
  // Create the window itself
  Create(NULL,
    "Hello World!",
    WS_OVERLAPPEDWINDOW,
    CRect(0,0,200,200));
  // Create a static label
  cs = new CStatic();
  cs->Create("hello world",
    WS_CHILD|WS_VISIBLE|SS_CENTER,
    CRect(50,80,150,150),
    this);
}
```

This small program does three things. First, it creates an "application object." Every MFC program you write will have a single application object that handles the initialization details of MFC and Windows. Next, the application creates a single window on the screen to act as the main application window. Finally, inside that window the application creates a single static text label containing the words "hello world". We will look at this program in detail in the next tutorial to gain a complete understanding of its structure.

The steps necessary to enter and compile this program are straightforward. If you have not yet installed Visual C++ on your machine, do so now. You will have the option of creating standard and custom installations. For the purposes of these tutorials a standard installation is suitable and after answering two or three simple questions the rest of the installation is quick and painless.

The compilation instructions supplied here apply specifically to Visual C++ version 5.x under Windows NT or Windows 95. If you are using Visual C++ version 1.5, 2.x, 4.x, or 6.x then you will want to see the tutorials for these versions on the MFC Tutorials page.

Start VC++ by double clicking on its icon in the Visual C++ group of the Program Manager. If you have just installed the product, you will see an empty window with a menu bar. If VC++ has been used before on this machine, it is possible for it to come up in several different states because VC++ remembers and automatically reopens the project and files in use the last time it exited. What we want right now is a state where it has no project or code loaded. If the program starts with a dialog that says it was unable to find a certain file, clear the dialog by clicking the "No" button. Go to the **Window** menu and select the **Close All** option if it is available. Go to the **File** menu and select the **Close** option if it is available to close any remaining windows. Now you are at the proper starting point. If you have just installed the package, you will see a window that looks something like this:

This screen can be rather intimidating the first time you see it. To eliminate some of the intimidation, click on the lower of the two "x" buttons that you see in the upper right hand corner of the screen if it is available. This action will let you close the "InfoViewer Topic" window. You may later get that window back if you desire by clicking on the little house button in the InfoViewer toolbar. If you want to get rid of the InfoViewer toolbar as well, you can drag it so it docks somewhere along the side of the window, or close it and later get it back by choosing the **Customize** option in the **Tools** menu.

What you see now is "normal". Along the top is the menu bar and several toolbars. Along the left side are all of the topics available from the on-line book collection (you might want to explore by double clicking on several of the items you see there - the collection of information found in the on-line books is gigantic). Along the bottom is a status window where various messages will be displayed.

Now what? What you would like to do is type in the above program, compile it and run it.

Before you start, switch to the File Manager (or the MS-DOS prompt) and make sure your drive has at least five megabytes of free space available. Then take the following steps.

Creating a Project and Compiling the Code

In order to compile any code in Visual C++, you have to create a *project*. With a very small program like this the project seems like overkill, but in any real program the project concept is quite useful. A project holds three different types of information:

1. It remembers all of the source code files that combine together to create one executable. In this simple example, the file HELLO.CPP will be the only source file, but in larger applications you often break the code up into several different files to make it easier to understand (and also to make it possible for several people to work on it simultaneously). The project maintains a list of the different source files and compiles all of them as necessary each time you want to create a new executable.
2. It remembers compiler and linker options particular to this specific application. For example, it remembers which libraries to link into the executable, whether or not you want to use pre-compiled headers, and so on.
3. It remembers what type of project you wish to build: a console application, a windows application, etc.

If you are familiar with makefiles, then it is easy to think of a project as a machine-generated makefile that has a very easy-to-understand user interface to manipulate it. For now we will create a very simple project file and use it to compile HELLO.CPP.

To create a new project for HELLO.CPP, choose the **New** option in the **File** menu. Under the **Projects** tab, highlight **Win32 Application**. In the Location field type an appropriate path name or click the Browse button. Type the word "hello" in for the project name, and you will see that word echoed in the Location field as well. Click the OK button. Visual C++ will create a new subdirectory named HELLO and place the project files named HELLO.OPT, HELLO.NCB, HELLO.DSP, and HELLO.DSW in that directory. If you quit and later want to reopen the project, double-click on HELLO.DSW.

The area along the left side of the screen will now change so that three tabs are available. The InfoView tab is still there, but there is now also a ClassView and a FileView tab. The ClassView tab will show you a list of all of the classes in your application and the FileView tab gives you a list of all of the files in the project.

Now it is time to type in the code for the program. In the **File** menu select the **New** option to create a new editor window. In the dialog that appears, make sure the **Files** tab is active and request a "Text File". Visual C++ comes with its own intelligent C++ editor, and you will use it to enter the program shown above. Type the code in the listing into the editor window. You will find that the editor automatically colors different pieces of text such as comments, key words, string literals, and so on. If you want to change the colors or turn the coloring off, go to the **Options** option in the **Tools** menu, choose the **Format** tab and select the **Source**

Windows option from the left hand list. If there is some aspect of the editor that displeases you, you may be able to change it using the **Editor** tab of the **Options** dialog.

After you have finished entering the code, save the file by selecting the **Save** option in the **File** menu. Save it to a file named HELLO.CPP in the new directory Visual C++ created.

Now choose the **Add To Project** option in the Project menu, and select **Files...** You will see a dialog that lets you select the source files that you want to include in your project. You can call up this dialog at any time to edit the project's files by selecting the **Add To Project** option. In this case, the HELLO.CPP file is the only source file needed, so double click on it.

In the area on the left side of the screen, click the FileView tab and double-click the folder icon labeled HELLO. You will see the file named HELLO.CPP. Click on the ClassView tab and double-click on the folder icon and you will see the classes in the application. You can remove a file from a project at any time by going to the FileView, clicking the file, and pressing the delete button.

*Finally, **you must now tell the project to use the MFC library**. If you omit this step the project will not link properly, and the error messages that the linker produces will not help one bit.* Choose the **Settings** option in the **Project** menu. Make sure that the **General** tab is selected in the tab at the top of the dialog that appears. In the **Microsoft Foundation Classes** combo box, choose the third option: "Use MFC in a Shared DLL." Then close the dialog.

Having created the project file and adjusted the settings, you are ready to compile the HELLO.CPP program. In the **Build** menu you will find three different compile options:

1. Compile HELLO.CPP (only available if the text window for HELLO.CPP has focus)
2. Build HELLO.EXE
3. Rebuild All

The first option simply compiles the source file listed and forms the object file for it. This option does not perform a link, so it is useful only for quickly compiling a file to check for errors. The second option compiles all of the source files in the project that have been modified since the last build, and then links them to form an executable. The third option recompiles all of the source files in the project and relinks them. It is a "compile and link from scratch" option that is useful after you change certain compiler options or move to a different platform.

In this case, choose the **Build HELLO.EXE** option in the **Build** menu to compile and link the code. Visual C++ will create a new subdirectory named Debug and place the executable named HELLO.EXE in that new subdirectory. This subdirectory holds all disposable (easily recreated) files generated by the compiler, so you can delete this directory when you run short on disk space without fear of losing anything important.

If you see compiler errors, simply double click on the error message in the output window. The editor will take you to that error. Compare your code against the code above and fix the problem. If you see a mass of linker errors, it probably means that you specified the project type incorrectly in the dialog used to create the project. You may want to simply delete your new directory and recreate it again following the instructions given above exactly.

To execute the program, choose the **Execute HELLO.EXE** option in the **Build** menu. A window appears with the words "hello world". The window itself has the usual decorations: a title bar, re-size areas, minimize and maximize buttons, and so on. Inside the window is a static label displaying the words "hello world". Note that the program is complete. You can move the window, re-size it, minimize it, and cover and uncover it with other windows. With a very small amount of code you have created a complete Window application. This is one of the many advantages of using MFC. All the details are handled elsewhere.

To terminate the program, click on its system menu (the small box to the left of the title bar) and select the Close option.

Conclusion

In this tutorial you have successfully compiled and executed your first program. You will use these same steps for each of the programs you create in the following tutorials. You will find that you can either create a separate directory for each project that you create, or you can create a single project file and then add and remove different source files.

In the next tutorial, we will examine this program in detail so you may gain a more complete understanding of its structure.

Part 2

A Simple MFC Program

In this tutorial we will examine a simple MFC program piece by piece to gain an understanding of its structure and conceptual framework. We will start by looking at MFC itself and then examine how MFC is used to create applications.

An Introduction to MFC

MFC is a large and extensive C++ class hierarchy that makes Windows application development significantly easier. MFC is compatible across the entire Windows family. As each new version of Windows comes out, MFC gets modified so that old code compiles and works under the new system. MFC also gets extended, adding new capabilities to the hierarchy and making it easier to create complete applications.

The advantage of using MFC and C++ - as opposed to directly accessing the Windows API from a C program-is that MFC already contains and encapsulates all the normal "boilerplate" code that all Windows programs written in C must contain. Programs written in MFC are therefore much smaller than equivalent C programs. On the other hand, MFC is a fairly thin covering over the C functions, so there is little or no performance penalty imposed by its use. It is also easy to customize things using the standard C calls when necessary since MFC does not modify or hide the basic structure of a Windows program.

The best part about using MFC is that it does all of the hard work for you. The hierarchy contains thousands and thousands of lines of correct, optimized and robust Windows code. Many of the member functions that you call invoke code that would have taken you weeks to write yourself. In this way MFC tremendously accelerates your project development cycle.

MFC is fairly large. For example, Version 4.0 of the hierarchy contains something like 200 different classes. Fortunately, you don't need to use all of them in a typical program. In fact, it is possible to create some fairly spectacular software using only ten or so of the different classes available in MFC. The hierarchy is broken into several different class categories which include (but is not limited to):

- Application Architecture
- Graphical Drawing and Drawing Objects
- File Services
- Exceptions
- Structures - Lists, Arrays, Maps
- Internet Services
- OLE 2
- Database
- General Purpose

We will concentrate on visual objects in these tutorials. The list below shows the portion of the class hierarchy that deals with application support and windows support.

- CObject
- CCmdTarget
- CWinThread
- CWinApp
- CWnd
- CFrameWnd
- CDialog

- CView
- CStatic
- CButton
- CListBox
- CComboBox
- CEdit
- CScrollBar

There are several things to notice in this list. First, most classes in MFC derive from a base class called **CObject**. This class contains data members and member functions that are common to most MFC classes. The second thing to notice is the simplicity of the list. The **CWinApp** class is used whenever you create an application and it is used only once in any program. The **CWnd** class collects all the common features found in windows, dialog boxes, and controls. The **CFrameWnd** class is derived from **CWnd** and implements a normal framed application window. **CDialog** handles the two normal flavors of dialogs: modeless and modal respectively. **CView** is used to give a user access to a document through a window. Finally, Windows supports six native control types: static text, editable text, push buttons, scroll bars, lists, and combo boxes (an extended form of list). Once you understand this fairly small number of pieces, you are well on your way to a complete understanding of MFC. The other classes in the MFC hierarchy implement other features such as memory management, document control, data base support, and so on.

To create a program in MFC, you either use its classes directly or, more commonly, you derive new classes from the existing classes. In the derived classes you create new member functions that allow instances of the class to behave properly in your application. You can see this derivation process in the simple program we used in Tutorial 1, which is described in greater detail below. Both **CHelloApp** and **CHelloWindow** are derived from existing MFC classes.

Designing a Program

Before discussing the code itself, it is worthwhile to briefly discuss the program design process under MFC. As an example, imagine that you want to create a program that displays the message "Hello World" to the user. This is obviously a very simple application but it still requires some thought.

A "hello world" application first needs to create a window on the screen that holds the words "hello world". It then needs to get the actual "hello world" words into that window. Three objects are required to accomplish this task:

1. An application object which initializes the application and hooks it to Windows. The application object handles all low-level event processing.
2. A window object that acts as the main application window.
3. A static text object which will hold the static text label "hello world".

Every program that you create in MFC will contain the first two objects. The third object is unique to this particular application. Each application will define its own set of user interface objects that display the application's output as well as gather input from the user.

Once you have completed the user interface design and decided on the controls necessary to implement the interface, you write the code to create the controls on the screen. You also write the code that handles the messages generated by these controls as they are manipulated by the user. In the case of a "hello world" application, only one user interface control is necessary. It holds the words "hello world". More realistic applications may have hundreds of controls arranged in the main window and dialog boxes.

It is important to note that there are actually two different ways to create user controls in a program. The method described here uses straight C++ code to create the controls. In a large application, however, this method becomes painful. Creating the controls for an application containing 50 or 100 dialogs using C++ code to do it would take an eon. Therefore, a second method uses *resource files* to create the controls with a graphical dialog editor. This method is much faster and works well on most dialogs.

Understanding the Code for "hello world"

The listing below shows the code for the simple "hello world" program that you entered, compiled and executed in Tutorial 1. Line numbers have been added to allow discussion of the code in the sections that follow. By walking through this program line by line, you can gain a good understanding of the way MFC is used to create simple applications.

If you have not done so already, please compile and execute the code below by following the instructions given in Tutorial 1.

```
1 //hello.cpp

2 #include <afxwin.h>

3 // Declare the application class
4 class CHelloApp : public CWinApp
5 {
6       public:
7               virtual BOOL InitInstance();
8 };
```

```
// Create an instance of the application class
CHelloApp HelloApp;

// Declare the main window class
class CHelloWindow : public CFrameWnd
{
    CStatic* cs;
    public:
    CHelloWindow();
};

// The InitInstance function is called each
// time the application first executes.
BOOL CHelloApp::InitInstance()
{
    m_pMainWnd = new CHelloWindow();
    m_pMainWnd->ShowWindow(m_nCmdShow);
    m_pMainWnd->UpdateWindow();
    return TRUE;
}

// The constructor for the window class
CHelloWindow::CHelloWindow()
{
    // Create the window itself
    Create(NULL,
        "Hello World!",
        WS_OVERLAPPEDWINDOW,
        CRect(0,0,200,200));
35      // Create a static label
36      cs = new CStatic();
37      cs->Create("hello world",
38              WS_CHILD|WS_VISIBLE|SS_CENTER,
39              CRect(50,80,150,150),
40              this);
41 }
```

Take a moment and look through this program. Get a feeling for the "lay of the land." The program consists of six small parts, each of which does something important.

The program first includes `afxwin.h` (line 2). This header file contains all the types, classes, functions, and variables used in MFC. It also includes other header files for such things as the Windows API libraries.

Lines 3 through 8 derive a new application class named **CHelloApp** from the standard **CWinApp** application class declared in MFC. The new class is created so the **InitInstance** member function in the **CWinApp** class can be overridden. **InitInstance** is a virtual function that is called as the application begins execution.

In Line 10, the code declares an instance of the application object as a global variable. This instance is important because it causes the program to execute. When the application is loaded into memory and begins running, the creation of that global variable causes the default constructor for the **CWinApp** class to execute. This constructor automatically calls the **InitInstance** function defined in lines 18 though 26.

In lines 11 through 17, the **CHelloWindow** class is derived from the **CFrameWnd** class declared in MFC. **CHelloWindow** acts as the application's window on the screen. A new class is created so that a new constructor, destructor, and data member can be implemented.

Lines 18 through 26 implement the **InitInstance** function. This function creates an instance of the **CHelloWindow** class, thereby causing the constructor for the class in Lines 27 through 41 to execute. It also gets the new window onto the screen.

Lines 27 through 41 implement the window's constructor. The constructor actually creates the window and then creates a static control inside it.

An interesting thing to notice in this program is that there is no **main** or **WinMain** function, and no apparent event loop. Yet we know from executing it in Tutorial 1 that it processed events. The window could be minimized and maximized, moved around, and so on. All this activity is hidden in the main application class **CWinApp** and we therefore don't have to worry about it-event handling is totally automatic and invisible in MFC.

The following sections describe the different pieces of this program in more detail. It is unlikely that all of this information will make complete sense to you right now: It's best to read through it to get your first exposure to the concepts. In Tutorial 3, where a number of specific examples are discussed, the different pieces will come together and begin to clarify themselves.

The Application Object

Every program that you create in MFC will contain a single application object that you derive from the **CWinApp** class. This object must be declared globally (line 10) and can exist only once in any given program.

An object derived from the **CWinApp** class handles initialization of the application, as well as the main event loop for the program. The **CWinApp** class has several data members, and a number of member functions. For now, almost all are unimportant. If you would like to browse through some of these functions however, search for **CWinApp** in the MFC help file by choosing the **Search** option in the **Help** menu and typing in "CWinApp". In the program above, we have overridden only one virtual function in **CWinApp**, that being the **InitInstance** function.

The purpose of the application object is to initialize and control your application. Because Windows allows multiple instances of the same application to run simultaneously, MFC breaks the initialization process into two parts and uses two functions-**InitApplication** and **InitInstance**-to handle it. Here we have used only the **InitInstance** function because of the simplicity of the application. It is called each time a new instance of the application is invoked. The code in Lines 3 through 8 creates a class called **CHelloApp** derived from **CWinApp**. It contains a new **InitInstance** function that overrides the existing function in **CWinApp** (which does nothing):

```
3 // Declare the application class

4 class CHelloApp : public CWinApp

5 {

6       public:

7               virtual BOOL InitInstance();

8 };
```

Inside the overridden **InitInstance** function at lines 18 through 26, the program creates and displays the window using **CHelloApp**'s data member named **m_pMainWnd**:

```
18 // The InitInstance function is called each
19 // time the application first executes.
20 BOOL CHelloApp::InitInstance()
21 {
22      m_pMainWnd = new CHelloWindow();
23      m_pMainWnd->ShowWindow(m_nCmdShow);
24      m_pMainWnd->UpdateWindow();
25      return TRUE;
26 }
```

The **InitInstance** function returns a TRUE value to indicate that initialization completed successfully. Had the function returned a FALSE value, the application would terminate immediately. We will see more details of the window initialization process in the next section.

When the application object is created at line 10, its data members (inherited from **CWinApp**) are automatically initialized. For example, **m_pszAppName**, **m_lpCmdLine**, and **m_nCmdShow** all contain appropriate values. See the MFC help file for more information. We'll see a use for one of these variables in a moment.

The Window Object

MFC defines two types of windows: 1) frame windows, which are fully functional windows that can be re-sized, minimized, and so on, and 2) dialog windows, which are not re-sizable. A frame window is typically used for the main application window of a program.

In the code shown in listing 2.1, a new class named **CHelloWindow** is derived from the **CFrameWnd** class in lines 11 through 17:

```
11 // Declare the main window class
12 class CHelloWindow : public CFrameWnd
13 {
14      CStatic* cs;
15      public:
16      CHelloWindow();
17 };
```

The derivation contains a new constructor, along with a data member that will point to the single user interface control used in the program. Each application that you create will have a unique set of controls residing in the main application window. Therefore, the derived class will have an overridden constructor that creates all the controls required in the main window. Typically this class will also have an overridden destructor to delete them when the window closes, but the destructor is not used here. In Tutorial 4, we will see that the derived window class will also declare a message handler to handle messages that these controls produce in response to user events.

Typically, any application you create will have a single main application window. The **CHelloApp** application class therefore defines a data member named **m_pMainWnd** that can point to this main window. To create the main window for this application, the **InitInstance** function (lines 18 through 26) creates an instance of **CHelloWindow** and uses **m_pMainWnd** to point to the new window. Our **CHelloWindow** object is created at line 22:

```
18 // The InitInstance function is called each
19 // time the application first executes.
20 BOOL CHelloApp::InitInstance()
21 {
22      m_pMainWnd = new CHelloWindow();
23      m_pMainWnd->ShowWindow(m_nCmdShow);
24      m_pMainWnd->UpdateWindow();
25      return TRUE;
26 }
```

Simply creating a frame window is not enough, however. Two other steps are required to make sure that the new window appears on screen correctly. First, the code must call the window's **ShowWindow** function to make the window appear on screen (line 23). Second, the program must call the **UpdateWindow** function to make sure that each control, and any drawing done in the interior of the window, is painted correctly onto the screen (line 24).

You may wonder where the **ShowWindow** and **UpdateWindow** functions are defined. For example, if you wanted to look them up to learn more about them, you might look in the MFC help file (use the **Search** option in the **Help** menu) at the **CFrameWnd** class description. **CFrameWnd** does not contain either of these member functions, however. It turns out that **CFrameWnd** inherits its behavior-as do all controls and windows in MFC-from the **CWnd** class (see figure 2.1). If you refer to **CWnd** in the MFC documentation, you will find that it is a huge class containing over 200 different functions. Obviously, you are not going to master this particular class in a couple of minutes, but among the many useful functions are **ShowWindow** and **UpdateWindow**.

Since we are on the subject, take a minute now to look up the **CWnd::ShowWindow** function in the MFC help file. You do this by clicking the help file's **Search** button and entering "ShowWindow". As an alternative, find the section describing the **CWnd** class using the **Search** button, and then find the **ShowWindow** function under the Update/Painting Functions in the class member list. Notice that **ShowWindow** accepts a single parameter, and that the parameter can be set to one of ten different values. We have set it to a data member held by **CHelloApp** in our program, **m_nCmdShow** (line 23). The **m_nCmdShow** variable is initialized based on conditions set by the user at application start-up. For example, the user may have started the application from the Program Manager and told the Program Manager to start the application in the minimized state by setting the check box in the application's properties dialog. The **m_nCmdShow** variable will be set to SW_SHOWMINIMIZED, and the application will start in an iconic state. The **m_nCmdShow** variable is a way for the outside world to communicate with the new application at start-up. If you would like to experiment, you can try replacing **m_nCmdShow** in the call to **ShowWindow** with the different constant values defined for **ShowWindow** . Recompile the program and see what they do.

Line 22 initializes the window. It allocates memory for it by calling the **new** function. At this point in the program's execution the constructor for the **CHelloWindow** is called. The constructor is called whenever an instance of the class is allocated. Inside the window's constructor, the window must create itself. It does this by calling the **Create** member function for the **CFrameWnd** class at line 31:

```
27 // The constructor for the window class
28 CHelloWindow::CHelloWindow()
29 {
30      // Create the window itself
31      Create(NULL,
32              "Hello World!",
33              WS_OVERLAPPEDWINDOW,
34              CRect(0,0,200,200));
```

Four parameters are passed to the create function. By looking in the MFC documentation you can see the different types. The initial NULL parameter indicates that a default class name be used. The second parameter is the title of the window that will appear in the title bar. The third parameter is the style attribute for the window. This example indicates that a normal, overlappable window should be created. Style attributes are covered in detail in Tutorial 3. The fourth parameter specifies that the window should be placed onto the screen with its upper left corner at point 0,0, and that the initial size of the window should be 200 by 200 pixels. If the value **rectDefault** is used as the fourth parameter instead, Windows will place and size the window automatically for you.

Since this is an extremely simple program, it creates a single static text control inside the window. In this particular example, the program uses a single static text label as its only control, and it is created at lines 35 through 40. More on this step in the next section.

The Static Text Control

The program derives the **CHelloWindow** class from the **CFrameWnd** class (lines 11 through 17). In doing so it declares a private data member of type **CStatic***, as well as a constructor.

As seen in the previous section, the **CHelloWindow** constructor does two things. First it creates the application's window by calling the **Create** function (line 31), and then it allocates and creates the control that belongs inside the window. In this case a single static label is used as the only control. Object creation is always a two-step process in MFC. First, the memory for the instance of the class is allocated, thereby calling the constructor to initialize any variables. Next, an explicit Create function is called to actually create the object on screen. The code allocates, constructs, and creates a single static text object using this two-step process at lines 36 through 40:

```
27 // The constructor for the window class
28 CHelloWindow::CHelloWindow()
29 {
30      // Create the window itself
31      Create(NULL,
32           "Hello World!",
33           WS_OVERLAPPEDWINDOW,
34           CRect(0,0,200,200));
35      // Create a static label
36      cs = new CStatic();
37      cs->Create("hello world",
38              WS_CHILD|WS_VISIBLE|SS_CENTER,
39              CRect(50,80,150,150),
40              this);
41 }
```

The constructor for the **CStatic** item is called when the memory for it is allocated, and then an explicit **Create** function is called to create the **CStatic** control's window. The parameters used in the **Create** function here are similar to those used for window creation at Line 31. The first parameter specifies the text to be displayed by the control. The second parameter specifies the style attributes. The style attributes are discussed in detail in the next tutorial but here we requested that the control be a child window (and therefore displayed within another window), that it should be visible, and that the text within the control should be centered. The third parameter determines the size and position of the static control. The fourth indicates the parent window for which this control is the child. Having created the static control, it will appear in the application's window and display the text specified.

Conclusion

In looking at this code for the first time, it will be unfamiliar and therefore potentially annoying. Don't worry about it. The only part in the entire program that matters from an application programmer's perspective is the **CStatic** creation code at lines 36 through 40. The rest you will type in once and then ignore. In the next tutorial you will come to a full understanding of what lines 36 through 40 do, and see a number of options that you have in customizing a **CStatic** control.

New and Cool in MFC 6.0

Changes to AppWizard, IE4 Controls, and a connection to DHTML make this rev worth learning By Scot Wingo

You may already be enjoying the productivity gains from great new features in Microsoft Visual C++ 6.0 like the IDE, editor, and compiler. Now that you've had a chance to recompile your current projects with the latest rev of Visual C++, it's time to look at what's new in the latest release of MFC, Version 6.0.

Fire It Up!

The fastest way to discover what's new in MFC 6.0 is to fire up our old friend, AppWizard, and see what's changed since Version 5.0. Go ahead and start AppWizard by selecting File/New/MFC AppWizard (EXE).

Figure 1: New AppWizard functionality

The first difference you'll notice occurs in step 1. Figure 1 shows a new option: "Document/View architecture support?" In previous versions, everything generated by AppWizard was automatically document/view-oriented, and you had to manually remove any document/view (and document template, etc.). Now if you're starting a project and don't care for document/view support, this option automatically omits document/view references from the generated code for you.

For this example, accept all of the defaults and go to step 4. Figure 2 shows step 4 of the AppWizard and another new option: "How do you want your toolbars to look?" with the "Normal" and "Internet Explorer ReBars" options. Choose Internet Explorer ReBars.

Figure 3: Project style option

At the top of Step 5, shown in Figure 3, you'll get another new question: "What style of project would you like?" The options are "MFC Standard" or "Windows Explorer." Choose the

Windows Explorer option.

Finally, Step 6 includes a new but hidden feature. Can you find it? If you select a view, you'll notice a new CView derivative: CHtmlView. Figure 4 shows where this new option lives.

Figure 4: CHtmlView

CHtmlView is fairly sophisticated, so we'll cover it in a separate section later. Accept the defaults in step 6 and compile and run the application. Figure 5 shows the application in action.

Figure 5: Working application

Notice that the application has a fancy new toolbar that "slides" and has grippers like the toolbar in Internet Explorer 4. Also notice that a splitter has automatically been created for us so that we can more easily create an Explorer-style application with a CTreeView in the left pane and a CListView in the right pane.

Let's learn more about the new ReBar control by exploring the code that AppWizard generated (so you can convert existing applications to use the new toolbar) and customize the default ReBar.

Anatomy of a ReBar

In addition to the "sliding" and gripper functionality, the ReBar also allows you to provide many more internal control types—such as drop-down menus—than are available in CToolBar. Before we explore these features let's look at the components of a ReBar.

Figure 6: ReBar elements

Figure 6 shows the various parts of a ReBar. Each internal toolbar in a ReBar is called a "band." The raised edge where the user slides the band is called a gripper. Each band can also have a label as illustrated.

MFC 6.0 provides two new classes that facilitate working with ReBars:

- CReBar—a high-level abstraction class that provides members for adding CToolBar and CDialogBar classes to ReBars as bands. CReBar also handles communication (such as message notifications) between the underlying control and the MFC framework.
- CReBarCtrl—a low-level wrapper class that wraps the IE ReBar control. This class provides numerous members for creating and manipulating ReBars, but doesn't provide the niceties found in CReBar. Most MFC applications use CReBar and call the member function GetReBarCtrl( ), which returns a CReBarCtrl pointer to gain access to the lower-level control on an as-needed basis.

Working with CReBar

If you open the MainFrm.h header file generated earlier by AppWizard, you'll see the code below which declares the CReBar data member, m_ndReBar.

```
protected:  // control bar embedded members
        CStatusBar  m_wndStatusBar;
        CToolBar    m_wndToolBar;
        CReBar      m_wndReBar;
        CDialogBar  m_wndDlgBar;
```

In the MainFrm.cpp file, you can see the code that adds the toolbar and dialog bar to the CReBar:

```
        if (!m_wndReBar.Create(this) ||
                !m_wndReBar.AddBar(&m_wndToolBar) ||
                !m_wndReBar.AddBar(&m_wndDlgBar))
        {
                TRACE0("Failed to create rebar\n");
                return -1;      // fail to create
        }
```

Let's customize the "TODO: layout dialog bar" band from the application we generated earlier with AppWizard.

First, open up the Visual C++ resource editor. Under the dialog heading you'll find a dialog resource (for the dialog bar) with ID IDR_MAINFRAME, which contains "TODO: layout dialog bar." Let's follow AppWizard's suggestion and put some real controls into the dialog bar. First, delete the static control with the "TODO" text in it. Next, place a combo box in the dialog bar and enter some default data items: "one," "two," "buckle," "my," "shoe"! Now place a button and change the button's text to "Increment." Next, place a progress bar with the default properties and place another button with the text "Decrement." When you're done laying out the dialog bar it should look similar to Figure 7.

Figure 7: Adding controls to the dialog bar

Before we can program the handlers for the "Increment" and "Decrement" buttons, we need to attach the dialog bar to a class using ClassWizard. While in the resource editor, bring up ClassWizard by double clicking on the Increment button. Choose Select an existing class. We choose this option because we want our dialog resource to be a band in the toolbar, not a separate dialog class. Choose CMainFrame from the list and select OK. ClassWizard prompts you with one last dialog. Select Yes and exit ClassWizard. You have successfully associated the IDR_MAINFRAME dialog bar with the CMainFrame class!

Finally, let's add some behavior to the dialog bar. Bring up the IDR_MAINFRAME dialog resource in the resource editor and double click on the Increment button. ClassWizard automatically creates an OnButton1( ) handler for you—accept the default name for this function. Enter this code for the OnButton1( ) function:

```
void CMainFrame::OnButton1()

{

      CProgressCtrl * pProgress =

            (CProgressCtrl*)m_wndDlg Bar.GetDlgItem

                                          (IDC_PROGRESS1);

      pProgress->StepIt();

}
```

The OnButton1( ) handler first gets a pointer to the progress control and then calls StepIt( ) to increment the progress control.

Now we need to add similar code to the decrement handler. Double click on the Decrement button in the resource editor, and ClassWizard will automatically create an OnButton2( ) handler. Add the following code to the OnButton2( ) member function:

```
void CMainFrame::OnButton2()

{

CProgressCtrl * pProgress =

        (CProgressCtrl*)m_wndDlgBar.GetDlgItem

                                        (IDC_PROGRESS1);


      int nCurrentPos = pProgress->GetPos();


      pProgress->SetPos(nCurrentPos-10);

}
```

Figure 8: Application results

Now you’re ready to compile and run the application. Figure 8 shows the results. This example is just the tip of the ReBar functionality iceberg, but it should give you an idea of how to get started and what it’s like to work with the new class. The Visual C++ on-line documentation is a great place to learn more about ReBar’s capabilities.

Actually, the ReBar is one of several new controls that are part of the Internet Explorer SDK now supported in MFC 6.0. Let’s take a look at some of the other new controls that are supported.

IE 4 Controls

When Microsoft released Internet Explorer 4, they included a new and improved version of the COMCTL32.DLL, which houses the Windows Common Controls. Because this update to the common controls wasn’t part of an operating system release, Microsoft calls the update the “Internet Explorer 4 Common Controls.” It updates all of the existing controls and adds a variety of advanced new controls. Visual C++ 6.0 and MFC 6.0 include a great deal of support for these new controls.

Figure 9 shows a dialog with each of the new IE4 controls. You can refer to it as you read the descriptions of the controls that follow.

Figure 9: New IE4 controls

Date and Time Picker

Before the date and time picker was provided with the IE4 controls, to enter a date, you had to use a third-party control, or subclass an MFC edit control to do significant data validation to ensure the entered date was valid. Fortunately, the new date and time picker provides an advanced control that prompts the user for a date or time while offering the developer a wide variety of styles and options. For example, dates can be displayed in short (8/14/68) or long (August 14, 1968) formats. A time mode lets the user enter a time using the familiar hour:minute:second AM/PM format.

The control also lets you decide if you want the user to select the date via in-place editing, a pop-down calendar, or a spinner. A variety of selection options are available, including single and multi (a range of dates) selections and the ability to turn on and off the red-ink “circling” of the current date. The control even has a mode that lets the user select “no date” by checking a check box. In Figure 9, the first four controls on the left illustrate the variety of configurations available with this control.

The new MFC 6.0 CDateTimeCtrl class provides the MFC interface to the IE4 date and time picker common control. This class offers a variety of notifications that enhance the programmability of the control. CDateTimeCtrl provides member functions for dealing with either CTime or COleDateTime time structures.

You set the date and time in a CDateTimeCtrl using the SetTime( ) member function and retrieve the date and time via the GetTime( ) function. You can create custom formats using the SetFormat( ) member function and change a variety of other configurations using the CDateTimeCtrl interface.

Month Calendar

The large display at the bottom left of Figure 9 is a month calendar. Like the date and time picker, the month calendar lets the user choose a date; but the month calendar can also be used to implement a small “Personal Information Manager (PIM)” in your applications. You can use the month calendar to show from one to 12 months. Figure 9 uses the month calendar to show only two months.

The month calendar supports single or multiple selection and allows you to display a variety of different options such as numbered months and circled days. Notifications for the control let you specify which dates are bold. It’s up to you to decide what this represents. For example, you could use the bold feature to indicate holidays, appointments, or invalid dates. The MFC 6.0 CMonthCalCtrl class provides the implementation of this control.

To initialize CMonthCal-Ctrl, you can call the SetToday() member function. CMonthCalCtrl provides members that deal with both CTime and COleDateTime.

Internet Protocol Address Control

If you write an application that uses any form of Internet or TCP/IP functionality, you may need to prompt the user for an Internet Protocol (IP) address. The IE4 Common Controls include an IP address control as shown in the top right of Figure 9. In addition to allowing the user to enter a four-byte IP address, this control performs automatic validation of the entered IP address. CIPAddressCtrl provides MFC support for the IP address control.

Figure 10: IP address

An IP address consists of four "fields" as shown in Figure 10. The fields are numbered from left to right. To initialize an IP address control, call the SetAddress( ) member function in your OnInitDialog( ) function. SetAddress( ) takes a DWORD, with each byte in the DWORD representing one of the fields. In your message handlers, you can call the GetAddress( ) member function to retrieve a DWORD or a series of bytes to retrieve the various values of the four IP address fields.

Extended Combo Box

The "old fashioned" combo box was developed in the early days of Windows, and its age and inflexible design have been the source of a great deal of developer confusion. With the IE4 controls, Microsoft released a much more flexible version of the combo box, called the "extended combo box."

The extended combo box gives you much simpler access and control over the edit control portion of the combo box. In addition, the extended combo box lets you attach an image list to the items in the combo box so you can easily display graphics in it. (Remember the old days when you had to use owner-drawn combo boxes?) Each item in the extended combo box can be associated with a selected image, an unselected image, and an overlay image. These three images can be used to provide a variety of graphical displays in the combo box. The bottom two combo boxes in Figure 9 are both extended combo boxes. The MFC CComboBoxEx class provides comprehensive extended combo box support.

CComboBoxEx can be attached to a CImageList that will automatically display graphics next to the text in the extended combo box. If you're already familiar with CComboBox, CComboBoxEx may cause some confusion; instead of containing strings, the extended combo box contains items of type COMBOBOXEXITEM. COMBOBOXEXITEM is a structure that consists of the following fields:

- UINT mask—A set of bit flags that specify which operations are to be performed using the structure. For example, set the CBEIF_IMAGE flag if the iImage field is to be set or retrieved in an operation.
- int iItem—Extended combo box item number. Like the older style combo box, the extended combo box uses zero-based indexing.
- LPTSTR pszText—Text of the item.
- int cchTextMax—Length of the buffer available in pszText.
- int iImage—Zero-based index into an associated image list.
- int iSelectedImage—Index of the image in the image list to be used to represent the “selected” state.
- int iOverlay—Index of the image in the image list to be used to overlay the current image.
- int iIndent—Number of 10-pixel indentation spaces.
- LPARAM lParam—32-bit parameter for the item.

See Bill Wagner’s article, “Build a Browser in an Afternoon Using Microsoft’s Web Browser” at www.vcdj.com/vcdj/jul98 /browser1.asp for more more information.

The last and perhaps biggest new feature in MFC 6.0 is support for Dynamic HTML in the CHTMLView class.

DHTML Support

Dynamic HTML (DHTML) is a new and exciting feature introduced as part of IE4. It provides serious benefits that could ultimately change the way we think about developing Windows applications. What’s the buzz about?

It all starts with the IE “HTML display engine,” sometimes called “Trident” in Microsoft literature. As part of the design of IE4, Microsoft made Trident its own COM component, exposing many of its internal objects used for displaying HTML pages in IE. This feature allows you to traverse the portions of an HTML page in script or code as if the HTML page was a data structure. Gone are the days of having to parse HTML or write grotesque CGI scripts to get to data in a form. The real power in DHTML doesn’t lie in the ability to access the HTML objects, but to actually change and manipulate the HTML page on-the-fly—thus, the name Dynamic HTML.

Once you understand the concept of DHTML, a million possible applications come to mind. For the Webmaster, this means that much of the logic that manipulates a page can live in scripts that are downloaded to the client. For the C++ developer, it means you can embed DHTML in your applications and use it as an embedded Web client or as a super-flexible, dynamic “form” that your application can change on-the-fly. For an example of how to leverage Web views in your applications, try some of the new Internet-enabled applications like Quicken 99 or Microsoft Money 99.

DHTML is so powerful and extensive that it would take a separate book (like Inside Dynamic HTML by Scott Isaacs) to fill you in on all of the copious details. For example, to really leverage DHTML, you need to understand the many possible elements of an HTML page, such as forms, lists, and stylesheets. Instead of covering all of the aspects of DHTML, I'll briefly introduce you to the DHTML object model and show you how to work with it using MFC. This is all made possible by the excellent DHTML support introduced in Visual C++ 6.0.

The DHTML Object Model

If you've been heads-down on a Visual C++ project and haven't had time to peek at HTML, it's an ASCII mark-up language format. Here's a very basic look at a simplistic HTML page:

```
<html>

<head>

<title>

This is an example of a very basic HTML page!

</title>

</head>

<body>

<h1>This is some text with H1!

</h1>

<h3>

This is some text with H3!

</h3>

</body>

</html>
```

This basic HTML "document" is composed of several "elements." The head (or header) contains the title: "This is an example of a very basic HTML page!" Next, the body of the document contains two elements. The first element has a style of heading 1 (h1) and reads:

"This is some text with H1!" The last element includes text with style heading 3 (h3) that reads: "This is some text with H3!"

Figure 11: DHTML object model hierarchy
Graphic from Inside Dynamic HTML by Scott Isaacs. Reproduced by permission of Microsoft Press. All rights reserved.

When IE loads this HTML page, it creates an internal representation that you can traverse, read, and manipulate through the DHTML object model. Figure 11 shows the basic hierarchy of the DHTML object model.

At the root of the object model is the window object. This object can be used from a script to do something like pop-up a dialog box. Here’s an example of some script that accesses the window object:

```
<SCRIPT LANGUAGE="JavaScript">

function about()

{

window.showModalDialog("about.htm","",

      "dialogWidth:25em;dialogHeight13em")

}

</SCRIPT>
```

When the about script is called, it in turn calls the showModalDialog( ) function in the window DHTML object to display a dialog. This example also illustrates how scripts access the object model—through globally accessible objects that map directly to the corresponding object in the DHTML object model.

The window object has several “sub” objects that further allow you to manipulate portions of IE4. The document object is what you’ll spend most of your time on when writing DHTML code, because it gives you programmatic access to the various elements of the HTML document currently loaded. The following script shows how to create basic dynamic content that changes the document object:

```
<HTML>

<HEAD>

<TITLE>Welcome!</TITLE>

<SCRIPT LANGUAGE="JScript">

function changeMe() {

        document.all.MyHeading.outerHTML = "<H1

                ID=MyHeading>Dynamic HTML is magic!</H1>";

        document.all.MyHeading.style.color = "green";

        document.all.MyText.innerText = "Presto Change-o! ";

        document.all.MyText.align = "center";

        document.body.insertAdjacentHTML("BeforeEnd",

                "<P ALIGN=\"center\">Open Sesame!</P>");

}

</SCRIPT>

<BODY onclick="changeMe()">

<H3 ID=MyHeading> Dynamic HTML demo!</H3>

<P ID=MyText>Click anywhere to see the power of DHTML!</P>

</BODY>

</HTML>
```

This script changes the MyHeading and MyText objects in the HTML documents on-the-fly. Not only does it change the text, but it also changes attributes of the elements such as the color and alignment. This script is included with this article on www.vcdj.com.

Before we further decompose the DHTML object model, you need to understand the DHTML concept of a “collection.” Collections in DHTML are logically equivalent to C++ data structures such as linked lists. In fact, access to the DHTML object model is largely performed by iterating through collections searching for a particular HTML element and then potentially iterating through another sub-collection to get to yet another element. Elements contain several methods such as “contains” and “length,” which you use to traverse through the elements.

For example, one of the sub-elements of the document object is a collection called “all” that contains all of the document’s elements. In fact, most of the sub-objects of the document object are collections. The following script shows how to iterate through the “all” collection and lists the various items of a document.

```
<HTML>

<HEAD><TITLE>Iterating through the all collection.</TITLE>

<SCRIPT LANGUAGE="JScript">

function listAllElements() {

    var tag_names = "";

    for (i=0; i<document.all.length; i++)

        tag_names = tag_names +

                        document.all(i).tagName + " ";

        alert("This document contains: " + tag_names);

}

</SCRIPT>

</HEAD>

<BODY onload="listAllElements()">

<H1>DHTML Rocks!</H1>

<P>This document is <B>very</B> short.

</BODY>

</HTML>
```

Notice how easy it is to retrieve items with script (using parentheses similar to how we access an array in C++)? Also notice that each element in an HTML document has properties such as “tagName” that allow you to search programmatically for various elements. For example, if you wanted to write a script that filtered out all bold items, you would scan the all collection for an element with tagName “B.”

Now that we’ve covered the basics of the DHTML object model and how to access them through scripts from the Web master’s perspective, let’s look at how MFC 6.0 lets us work with DHTML from an application developer’s perspective.

MFC 6.0 and the DHTML Connection

MFC 6.0’s CHTMLView gives you complete access to the DHTML object model. However, access to the object model from languages like C++ is done through OLE Automation (IDispatch) and in many cases isn't as cut-and-dried as some of the scripts we looked at earlier. The DHTML object model gets exposed through a set of COM objects with the prefix "IHTML" (as in IHTMLDoc-ument, IHTMLWindow, IHTMLElement, and IHTMLBodyElement). In C++, once you've obtained the document interface, you can use any of the IHTMLDocument2 interfaces to obtain or modify the document's properties.

You can access the all collection by calling the IHTMLDoc-ument2::get_all( ) method. This method returns an IHTML-ElementCollection collection interface that contains all the elements in the document. You can then iterate through the collection using the IHTMLElementCollection::item( ) method (similar to the parentheses in the scripts above). The IHTMLElementCollection::item( ) method supplies you with an IDispatch pointer on which you can call QueryInterface, requesting the IID_IHTMLElement interface. This gives you an IHTMLElement interface pointer to get or set information for the HTML element.

Most elements also provide a specific interface for working with that element type. These element-specific interface names take the format of IHTMLXXXXElement, where XXXX is the name of the element (as in IHTMLBodyElement). You must call QueryInterface on the IHTMLElement object to request the element-specific interface you need. If this sounds confusing, it can be! But don't worry, we'll go through an example of how to use DHTML with MFC.

Where the Connection Begins

MFC's support for DHTML starts with a new CView derivative, CHtmlView, allowing you to embed an HTML view inside of frame windows or splitter windows. With some DHTML work, CHtmlView can act as a dynamic form. Let's use it to generate a simple application and add some DHTML manipulation code.

First, create a simple MDI application with a CHtmlView as the view. In Visual C++, select File/New/Projects and choose MFC AppWizard (EXE). Accept all the defaults up to step 6. In step 6 choose CHtmlView as the base class.

If you look in your ProjectNameView.cpp file, you'll see this line in the CProjectNameView::OnInitialUpdate( ) function:

```
        Navigate2(_T("http://www.microsoft.com/visualc/"),NULL,NULL);
```

Figure 12: Sample application

You can edit this line to have the application load a different URL from the Visual C++ page, or a local page. Compile and run this application to see the basic CHtmlView in action.

Figure 12 shows the application running.

You can obtain on www.vcdj.com a more comprehensive DHTML sample that creates a CHtmlView and a CListView separated by a splitter. It then uses DHTML to enumerate the HTML elements in the CHtmlView and displays the results in the CListView. The end result is a DHTML explorer that you can use to view the DHTML object model of any HTML file! The sample includes a README.TXT file that describes the steps used to generate the application. Figure 13 shows the MFC 6.0 sample in action.

Figure 13: MFC 6.0 sample

For more information

I hope this introduction to DHTML has started you thinking about some ways you can use this exciting new technology in your MFC-based applications. The possibilities are endless: applications that update from the Internet, completely dynamic applications, and countless others.

THE COMPLETE WINDOWS PROGRAMMING GUIDE

PART 1 2 3

Introduction to MFC Programming with Visual C++ v5.x

by Marshall Brain

A Simple MFC Program

In this tutorial we will examine a simple MFC program piece by piece to gain an understanding of its structure and conceptual framework. We will start by looking at MFC itself and then examine how MFC is used to create applications.

An Introduction to MFC

MFC is a large and extensive C++ class hierarchy that makes Windows application development significantly easier. MFC is compatible across the entire Windows family. As each new version of Windows comes out, MFC gets modified so that old code compiles and works under the new system. MFC also gets extended, adding new capabilities to the hierarchy and making it easier to create complete applications.

The advantage of using MFC and C++ - as opposed to directly accessing the Windows API from a C program-is that MFC already contains and encapsulates all the normal "boilerplate" code that all Windows programs written in C must contain. Programs written in MFC are therefore much smaller than equivalent C programs. On the other hand, MFC is a fairly thin covering over the C functions, so there is little or no performance penalty imposed by its use. It is also easy to customize things using the standard C calls when necessary since MFC does not modify or hide the basic structure of a Windows program.

The best part about using MFC is that it does all of the hard work for you. The hierarchy contains thousands and thousands of lines of correct, optimized and robust Windows code. Many of the member functions that you call invoke code that would have taken you weeks to write yourself. In this way MFC tremendously accelerates your project development cycle.

MFC is fairly large. For example, Version 4.0 of the hierarchy contains something like 200 different classes. Fortunately, you don't need to use all of them in a typical program. In fact, it is possible to create some fairly spectacular software using only ten or so of the different classes available in MFC. The hierarchy is broken into several different class categories which include (but is not limited to):

- Application Architecture
- Graphical Drawing and Drawing Objects
- File Services
- Exceptions
- Structures - Lists, Arrays, Maps
- Internet Services
- OLE 2
- Database
- General Purpose

We will concentrate on visual objects in these tutorials. The list below shows the portion of the class hierarchy that deals with application support and windows support.

- CObject
- CCmdTarget
- CWinThread
- CWinApp
- CWnd
- CFrameWnd
- CDialog
- CView
- CStatic
- CButton
- CListBox
- CComboBox
- CEdit
- CScrollBar

There are several things to notice in this list. First, most classes in MFC derive

Visualizing MFC

One of the most frusterating things when you are first learning MFC is the "Where am I?" feeling you get. MFC has *hundreds* of classes. A good way to get around this feeling is to use a class hierarchy visualization tool like CodeVizor. With CodeVizor you can drag the source code for MFC into the CodeVizor tool and in about 30 seconds have a beautiful, clickable (and printable!) class hierarchy chart. Get CodeVizor and see how much easier undestanding MFC becomes!

from a base class called **CObject**. This class contains data members and member functions that are common to most MFC classes. The second thing to notice is the simplicity of the list. The **CWinApp** class is used whenever you create an application and it is used only once in any program. The **CWnd** class collects all the common features found in windows, dialog boxes, and controls. The **CFrameWnd** class is derived from **CWnd** and implements a normal framed application window. **CDialog** handles the two normal flavors of dialogs: modeless and modal respectively. **CView** is used to give a user access to a document through a window. Finally, Windows supports six native control types: static text, editable text, push buttons, scroll bars, lists, and combo boxes (an extended form of list). Once you understand this fairly small number of pieces, you are well on your way to a complete understanding of MFC. The other classes in the MFC hierarchy implement other features such as memory management, document control, data base support, and so on.

To create a program in MFC, you either use its classes directly or, more commonly, you derive new classes from the existing classes. In the derived classes you create new member functions that allow instances of the class to behave properly in your application. You can see this derivation process in the simple program we used in Tutorial 1, which is described in greater detail below. Both **CHelloApp** and **CHelloWindow** are derived from existing MFC classes.

Designing a Program

Before discussing the code itself, it is worthwhile to briefly discuss the program design process under MFC. As an example, imagine that you want to create a program that displays the message "Hello World" to the user. This is obviously a very simple application but it still requires some thought.

A "hello world" application first needs to create a window on the screen that holds the words "hello world". It then needs to get the actual "hello world" words into that window. Three objects are required to accomplish this task:

1. An application object which initializes the application and hooks it to Windows. The application object handles all low-level event processing.
2. A window object that acts as the main application window.
3. A static text object which will hold the static text label "hello world".

Every program that you create in MFC will contain the first two objects. The third object is unique to this particular application. Each application will define its own set of user interface objects that display the application's output as well as gather input from the user.

Once you have completed the user interface design and decided on the controls necessary to implement the interface, you write the code to create the controls on the screen. You also write the code that handles the messages generated by these controls as they are manipulated by the user. In the case of a "hello world" application, only one user interface control is necessary. It holds the words "hello world". More realistic applications may have hundreds of controls arranged in the main window and dialog boxes.

It is important to note that there are actually two different ways to create user controls in a program. The method described here uses straight C++ code to create the controls. In a large application, however, this method becomes painful. Creating the controls for an application containing 50 or 100 dialogs using C++ code to do it would take an eon. Therefore, a second method uses *resource files* to create the controls with a graphical dialog editor. This method is much faster and works well on most dialogs.

Understanding the Code for "hello world"

The listing below shows the code for the simple "hello world" program that you entered, compiled and executed in Tutorial 1. Line numbers have been added to allow discussion of the code in the sections that follow. By walking through this program line by line, you can gain a good understanding of the way MFC is used to create simple applications.

If you have not done so already, please compile and execute the code below by following the instructions given in Tutorial 1.

```
1 //hello.cpp 2 #include <afxwin.h> 3 // Declare the application class 4 class CHelloApp :
public CWinApp 5 { 6 public: 7 virtual BOOL InitInstance(); 8 }; 9 // Create an instance of
the application class 10 CHelloApp HelloApp; 11 // Declare the main window class 12 class
CHelloWindow : public CFrameWnd 13 { 14 CStatic* cs; 15 public: 16 CHelloWindow(); 17
}; 18 // The InitInstance function is called each 19 // time the application first executes.
20 BOOL CHelloApp::InitInstance() 21 { 22 m_pMainWnd = new CHelloWindow(); 23
m_pMainWnd->ShowWindow(m_nCmdShow); 24 m_pMainWnd->UpdateWindow(); 25
return TRUE; 26 } 27 // The constructor for the window class 28
CHelloWindow::CHelloWindow() 29 { 30 // Create the window itself 31 Create(NULL, 32
"Hello World!", 33 WS_OVERLAPPEDWINDOW, 34 CRect(0,0,200,200)); 35 // Create a
static label 36 cs = new CStatic(); 37 cs->Create("hello world", 38
WS_CHILD|WS_VISIBLE|SS_CENTER, 39 CRect(50,80,150,150), 40 this); 41 }
```

Take a moment and look through this program. Get a feeling for the "lay of the land." The program consists of six small parts, each of which does something important.

The program first includes `afxwin.h` (line 2). This header file contains all the types, classes, functions, and variables used in MFC. It also includes other header files for such things as the Windows API libraries.

Lines 3 through 8 derive a new application class named **CHelloApp** from the standard **CWinApp** application class declared in MFC. The new class is created so the **InitInstance** member function in the **CWinApp** class can be overridden. **InitInstance** is a virtual function that is called as the application begins execution.

In Line 10, the code declares an instance of the application object as a global variable. This instance is important because it causes the program to execute. When the application is loaded into memory and begins running, the creation of that global variable causes the default constructor for the **CWinApp** class to execute. This constructor automatically calls the **InitInstance** function defined in lines 18 though 26.

In lines 11 through 17, the **CHelloWindow** class is derived from the **CFrameWnd** class declared in MFC. **CHelloWindow** acts as the application's window on the screen. A new class is created so that a new constructor, destructor, and data member can be implemented.

Lines 18 through 26 implement the **InitInstance** function. This function creates an instance of the **CHelloWindow** class, thereby causing the constructor for the class in Lines 27 through 41 to execute. It also gets the new window onto the screen.

Lines 27 through 41 implement the window's constructor. The constructor actually creates the window and then creates a static control inside it.

An interesting thing to notice in this program is that there is no **main** or **WinMain** function, and no apparent event loop. Yet we know from executing it in Tutorial 1 that it processed events. The window could be minimized and maximized, moved around, and so on. All this activity is hidden in the main application class **CWinApp** and we therefore don't have to worry about it-event handling is totally automatic and invisible in MFC.

The following sections describe the different pieces of this program in more detail. It is unlikely that all of this information will make complete sense to you right now: It's best to read through it to get your first exposure to the concepts. In Tutorial 3, where a number of specific examples are discussed, the different pieces will come together and begin to clarify themselves.

The Application Object

Every program that you create in MFC will contain a single application object that you derive from the **CWinApp** class. This object must be declared globally (line 10) and can exist only once in any given program.

An object derived from the **CWinApp** class handles initialization of the application, as well as the main event loop for the program. The **CWinApp** class has several data members, and a number of member functions. For now, almost all are unimportant. If you would like to browse through some of these functions however, search for **CWinApp** in the MFC help file by choosing the **Search** option in the **Help** menu and typing in "CWinApp". In the program above, we have overridden only one virtual function in **CWinApp**, that being the **InitInstance** function.

The purpose of the application object is to initialize and control your application. Because Windows allows multiple instances of the same application to run simultaneously, MFC breaks the initialization process into two parts and uses two functions-**InitApplication** and **InitInstance**-to handle it. Here we have used only the **InitInstance** function because of the simplicity of the application. It is called each time a new instance of the application is invoked. The code in Lines 3 through 8 creates a class called **CHelloApp** derived from **CWinApp**. It contains a new **InitInstance** function that overrides the existing function in **CWinApp** (which does nothing):

```
3 // Declare the application class 4 class CHelloApp : public CWinApp 5 { 6 public: 7
virtual BOOL InitInstance(); 8 };
```

Inside the overridden **InitInstance** function at lines 18 through 26, the program creates and displays the window using **CHelloApp**'s data member named **m_pMainWnd**:

```
18 // The InitInstance function is called each 19 // time the application first executes. 20
BOOL CHelloApp::InitInstance() 21 { 22 m_pMainWnd = new CHelloWindow(); 23
m_pMainWnd->ShowWindow(m_nCmdShow); 24 m_pMainWnd->UpdateWindow(); 25
return TRUE; 26 }
```

The **InitInstance** function returns a TRUE value to indicate that initialization completed successfully. Had the function returned a FALSE value, the application would terminate immediately. We will see more details of the window initialization process in the next section.

When the application object is created at line 10, its data members (inherited from **CWinApp**) are automatically initialized. For example, **m_pszAppName**, **m_lpCmdLine**, and **m_nCmdShow** all contain appropriate values. See the MFC help file for more information. We'll see a use for one of these variables in a moment.

The Window Object

MFC defines two types of windows: 1) frame windows, which are fully functional windows that can be re-sized, minimized, and so on, and 2) dialog windows, which are not re-sizable. A frame window is typically used for the main application window of a program.

In the code shown in listing 2.1, a new class named **CHelloWindow** is derived from the **CFrameWnd** class in lines 11 through 17:

```
11 // Declare the main window class 12 class CHelloWindow : public CFrameWnd 13 { 14
CStatic* cs; 15 public: 16 CHelloWindow(); 17 };
```

The derivation contains a new constructor, along with a data member that will point to the single user interface control used in the program. Each application that you create will have a unique set of controls residing in the main application window. Therefore, the derived class will have an overridden constructor that creates all the controls required in the main window. Typically this class will also have an overridden destructor to delete them when the window closes, but the destructor is not used here. In Tutorial 4, we will see that the derived window class will also declare a message handler to handle messages that these controls produce in response to user events.

Typically, any application you create will have a single main application window. The **CHelloApp** application class therefore defines a data member named **m_pMainWnd** that can point to this main window. To create the main window for this application, the **InitInstance** function (lines 18 through 26) creates an instance of **CHelloWindow** and uses **m_pMainWnd** to point to the new window. Our **CHelloWindow** object is created at line 22:

```
18 // The InitInstance function is called each 19 // time the application first executes. 20
BOOL CHelloApp::InitInstance() 21 { 22 m_pMainWnd = new CHelloWindow(); 23
m_pMainWnd->ShowWindow(m_nCmdShow); 24 m_pMainWnd->UpdateWindow(); 25
return TRUE; 26 }
```

Simply creating a frame window is not enough, however. Two other steps are required to make sure that the new window appears on screen correctly. First, the code must call the window's **ShowWindow** function to make the window appear on screen (line 23). Second, the program must call the **UpdateWindow** function to make sure that each control, and any drawing done in the interior of the window, is painted correctly onto the screen (line 24).

You may wonder where the **ShowWindow** and **UpdateWindow** functions are defined. For example, if you wanted to look them up to learn more about them, you might look in the MFC help file (use the **Search** option in the **Help** menu) at the **CFrameWnd** class description. **CFrameWnd** does not contain either of these member functions, however. It turns out that **CFrameWnd** inherits its behavior-as do all controls and windows in MFC-from the **CWnd** class (see figure 2.1). If you refer to **CWnd** in the MFC documentation, you will find that it is a huge class containing over 200 different functions. Obviously, you are not going to master this particular class in a couple of minutes, but among the many useful functions are **ShowWindow** and **UpdateWindow**.

Since we are on the subject, take a minute now to look up the **CWnd::ShowWindow** function in the MFC help file. You do this by clicking the help file's **Search** button and entering "ShowWindow". As an alternative, find the section describing the **CWnd** class using the **Search** button, and then find the **ShowWindow** function under the Update/Painting Functions in the class member list. Notice that **ShowWindow** accepts a single parameter, and that the parameter can be set to one of ten different values. We have set it to a data member held by **CHelloApp** in our program, **m_nCmdShow** (line 23). The **m_nCmdShow** variable is initialized based on conditions set by the user at application start-up. For example, the user may have started the application from the Program Manager and told the Program Manager to start the application in the minimized state by setting the check box in the application's properties dialog. The **m_nCmdShow** variable will be set to SW_SHOWMINIMIZED, and the application will start in an iconic state. The **m_nCmdShow** variable is a way for the outside world to communicate with the new application at start-up. If you would like to experiment, you can try replacing **m_nCmdShow** in the call to **ShowWindow** with the different constant values defined for **ShowWindow** . Recompile the program and see what they do.

Line 22 initializes the window. It allocates memory for it by calling the **new** function. At this point in the program's execution the constructor for the **CHelloWindow** is called. The constructor is called whenever an instance of the class is allocated. Inside the window's constructor, the window must create itself. It does this by calling the **Create** member function for the **CFrameWnd** class at line 31:

```
27 // The constructor for the window class 28 CHelloWindow::CHelloWindow() 29 { 30 //
Create the window itself 31 Create(NULL, 32 "Hello World!", 33
WS_OVERLAPPEDWINDOW, 34 CRect(0,0,200,200));
```

Four parameters are passed to the create function. By looking in the MFC documentation you can see the different types. The initial NULL parameter indicates that a default class name be used. The second parameter is the title of the window that will appear in the title bar. The third parameter is the style attribute for the window. This example indicates that a normal, overlappable window should be created. Style attributes are covered in detail in Tutorial 3. The fourth parameter specifies that the window should be placed onto the screen with its upper left corner at point 0,0, and that the initial size of the window should be 200 by 200 pixels. If the value **rectDefault** is used as the fourth parameter instead, Windows will place and size the window automatically for you.

Since this is an extremely simple program, it creates a single static text control inside the window. In this particular example, the program uses a single static text label as its only control, and it is created at lines 35 through 40. More on this step in the next section.

The Static Text Control

The program derives the **CHelloWindow** class from the **CFrameWnd** class (lines 11 through 17). In doing so it declares a private data member of type **CStatic***, as well as a constructor.

As seen in the previous section, the **CHelloWindow** constructor does two things. First it creates the application's window by calling the **Create** function (line 31), and then it allocates and creates the control that belongs inside the window. In this case a single static label is used as the only control. Object creation is always a two-step process in MFC. First, the memory for the instance of the class is allocated, thereby calling the constructor to initialize any variables. Next, an explicit Create function is called to actually create the object on screen. The code allocates, constructs, and creates a single static text object using this two-step process at lines 36 through 40:

```
27 // The constructor for the window class 28 CHelloWindow::CHelloWindow() 29 { 30 //
Create the window itself 31 Create(NULL, 32 "Hello World!", 33
WS_OVERLAPPEDWINDOW, 34 CRect(0,0,200,200)); 35 // Create a static label 36 cs =
new CStatic(); 37 cs->Create("hello world", 38 WS_CHILD|WS_VISIBLE|SS_CENTER, 39
CRect(50,80,150,150), 40 this); 41 }
```

The constructor for the **CStatic** item is called when the memory for it is allocated, and then an explicit **Create** function is called to create the **CStatic** control's window. The parameters used in the **Create** function here are similar to those used for window creation at Line 31. The first parameter specifies the text to be displayed by the control. The second parameter specifies the style attributes. The style attributes are discussed in detail in the next tutorial but here we requested that the control be a child window (and therefore displayed within another window), that it should be visible, and that the text within the control should be centered. The third parameter determines the size and position of the static control. The fourth indicates the parent window for which this control is the child. Having created the static control, it will appear in the application's window and display the text specified.

Conclusion

In looking at this code for the first time, it will be unfamiliar and therefore potentially annoying. Don't worry about it. The only part in the entire program that matters from an application programmer's perspective is the **CStatic** creation code at lines 36 through 40. The rest you will type in once and then ignore. In the next tutorial you will come to a full understanding of what lines 36 through 40 do, and see a number of options that you have in customizing a **CStatic** control.

PART 1 2 3

Introduction to MFC Programming with Visual C++ v5.x

by Marshall Brain

MFC Styles

Controls are the user interface objects used to create interfaces for Windows applications. Most Windows applications and dialog boxes that you see are nothing but a collection of controls arranged in a way that appropriately implements the functionality of the program. In order to build effective applications, you must completely understand how to use the controls available in Windows. There are only six basic controls-**CStatic**, **CButton** , **CEdit**, **CList**, **CComboBox**, and **CScrollBar** -along with some minor variations (also note that Windows 95 added a collection of about 15 enhanced controls as well). You need to understand what each control can do, how you can tune its appearance and behavior, and how to make the controls respond appropriately to user events. By combining this knowledge with an understanding of menus and dialogs you gain the ability to create any Windows application that you can imagine. You can create controls either programatically as shown in this tutorial, or through resource files using the dialog resource editor. While the dialog editor is much more convenient, it is extremely useful to have a general understanding of controls that you gain by working with them programatically as shown here and in the next tutorial.

The simplest of the controls, **CStatic**, displays static text. The **CStatic** class has no data members and only a few member functions: the constructor, the **Create** function for getting and setting icons on static controls, and several others. It does not respond to user events. Because of its simplicity, it is a good place to start learning about Windows controls.

In this tutorial we will look at the **CStatic** class to understand how controls can be modified and customized. In the following tutorial, we examine the **CButton** and **CScrollBar** classes to gain an understanding of event handling. Once you understand all of the controls and classes, you are ready to build complete applications.

The Basics

A **CStatic** class in MFC displays static text messages to the user. These messages can serve purely informational purposes (for example, text in a message dialog that describes an error), or they can serve as small labels that identify other controls. Pull open a File Open dialog in any Windows application and you will find six text labels. Five of the labels identify the lists, text area, and check box and do not ever change. The sixth displays the current directory and changes each time the current directory changes.

CStatic objects have several other display formats. By changing the *style* of a label it can display itself as a solid rectangle, as a border, or as an icon. The rectangular solid and frame forms of the **CStatic** class allow you to visually group related interface elements and to add separators between controls.

A **CStatic** control is always a child window to some parent window. Typically, the parent window is a main window for an application or a dialog box. You create the static control, as discussed in Tutorial 2, with two lines of code:

```
CStatic *cs; ... cs = new CStatic(); cs->Create("hello world",
WS_CHILD|WS_VISIBLE|SS_CENTER, CRect(50,80, 150, 150), this);
```

This two-line creation style is typical of all controls created using MFC. The call to **new** allocates memory for an instance of the **CStatic** class and, in the process, calls the constructor for the class. The constructor performs any initialization needed by the class. The **Create** function creates the control at the Windows level and puts it on the screen.

The **Create** function accepts up to five parameters, as described in the MFC help file. Choose the **Search** option in the **Help** menu of Visual C++ and then enter **Create** so that you can select **CStatic::Create** from the list. Alternatively, enter **CStatic** in the search dialog and then click the **Members** button on its overview page.

Most of these values are self-explanatory. The **lpszText** parameter specifies the text displayed by the label. The **rect** parameter controls the position, size, and shape of the text when it's displayed in its parent window. The upper left corner of the text is determined by the upper left corner of the **rect** parameter and its bounding rectangle is determined by the width and height of the rect parameter. The **pParentWnd** parameter indicates the parent of the **CStatic** control. The control will appear in the parent window, and the position of the control will be relative to the upper left corner of the client area of the parent. The **nID** parameter is an integer value used as a control ID by certain functions in the API. We'll see examples of this parameter in the next tutorial.

The **dwStyle** parameter is the most important parameter. It controls the appearance and behavior of the control. The following sections describe this parameter in detail.

CStatic Styles

All controls have a variety of display *styles*. Styles are determined at creation using the **dwStyle** parameter passed to the **Create** function. The style parameter is a bit mask that you build by or-ing together different mask constants. The constants available to a **CStatic** control can be found in the MFC help file (Find the page for the **CStatic::Create** function as described in the previous section, and click on the **Static Control Styles** link near the top of the page) and are also briefly described below:

Valid styles for the CStatic class -

Styles inherited from CWnd:

- WS_CHILD Mandatory for CStatic.
- WS_VISIBLE The control should be visible to the user.
- WS_DISABLED The control should reject user events.
- WS_BORDER The control's text is framed by a border.

Styles native to CStatic:

- SS_BLACKFRAME The control displays itself as a rectangular border. Color is the same as window frames.
- SS_BLACKRECT The control displays itself as a filled rectangle. Color is the same as window frames.
- SS_CENTER The text is center justified.
- SS_GRAYFRAME The control displays itself as a rectangular border. Color is the same as the desktop.
- SS_GRAYRECT The control displays itself as a filled rectangle. Color is the same as the desktop.
- SS_ICON The control displays itself as an icon. The text string is used as the name of the icon in a resource file. The rect parameter controls only positioning.

- SS_LEFT The text displayed is left justified. Extra text is word-wrapped.
- SS_LEFTNOWORDWRAP The text is left justified, but extra text is clipped.
- SS_NOPREFIX "&" characters in the text string indicate accelerator prefixes unless this attribute is used.
- SS_RIGHT The text displayed is right justified. Extra text is word-wrapped.
- SS_SIMPLE A single line of text is displayed left justified. Any CTLCOLOR messages must be ignored by the parent.
- SS_USERITEM User-defined item.
- SS_WHITEFRAME The control displays itself as a rectangular border. Color is the same as window backgrounds.
- SS_WHITERECT The control displays itself as a filled rectangle. Color is the same as window backgrounds.

These constants come from two different sources. The "SS" (Static Style) constants apply only to **CStatic** controls. The "WS" (Window Style) constants apply to all windows and are therefore defined in the **CWnd** object from which **CStatic** inherits its behavior. There are many other "WS" style constants defined in **CWnd**. They can be found by looking up the **CWnd::Create** function in the MFC documentation. The four above are the only ones that apply to a **CStatic**object.

A **CStatic** object will always have at least two style constants or-ed together: WS_CHILD and WS_VISIBLE. The control is not created unless it is the child of another window, and it will be invisible unless WS_VISIBLE is specified. WS_DISABLED controls the label's response to events and, since a label has no sensitivity to events such as keystrokes or mouse clicks anyway, specifically disabling it is redundant.

All the other style attributes are optional and control the appearance of the label. By modifying the style attributes passed to the **CStatic::Create** function, you control how the static object appears on screen. You can learn quite a bit about the different styles by using style attributes to modify the text appearance of the **CStatic** object, as discussed in the next section.

CStatic Text Appearance

The code shown below is useful for understanding the behavior of the **CStatic** object. It is similar to the listing discussed in Tutorial 2, but it modifies the creation of the **CStatic** object slightly. Please turn to Tutorial 1 for instructions on entering and compiling this code.

```
//static1.cpp #include <afxwin.h> // Declare the application class class CTestApp : public
CWinApp { public: virtual BOOL InitInstance(); }; // Create an instance of the application
```

```
class CTestApp TestApp; // Declare the main window class class CTestWindow : public CFrameWnd { CStatic* cs; public: CTestWindow(); }; // The InitInstance function is called // once when the application first executes BOOL CTestApp::InitInstance() { m_pMainWnd = new CTestWindow(); m_pMainWnd->ShowWindow(m_nCmdShow); m_pMainWnd->UpdateWindow(); return TRUE; } // The constructor for the window class CTestWindow::CTestWindow() { CRect r; // Create the window itself Create(NULL, "CStatic Tests", WS_OVERLAPPEDWINDOW, CRect(0,0,200,200)); // Get the size of the client rectangle GetClientRect(&r); r.InflateRect(-20,-20); // Create a static label cs = new CStatic(); cs->Create("hello world", WS_CHILD|WS_VISIBLE|WS_BORDER|SS_CENTER, r, this); }
```

The code of interest in listing 3.1 is in the function for the window constructor, which is repeated below with line numbers:

```
CTestWindow::CTestWindow() { CRect r; // Create the window itself 1 Create(NULL, "CStatic Tests", WS_OVERLAPPEDWINDOW, CRect(0,0,200,200)); // Get the size of the client rectangle 2 GetClientRect(&r); 3 r.InflateRect(-20,-20); // Create a static label 4 cs = new CStatic(); 5 cs->Create("hello world", WS_CHILD|WS_VISIBLE|WS_BORDER|SS_CENTER, r, this); }
```

The function first calls the **CTestWindow::Create** function for the window at line 1. This is the **Create** function for the **CFrameWnd** object, since **CTestWindow** inherits its behavior from **CFrameWnd**. The code in line 1 specifies that the window should have a size of 200 by 200 pixels and that the upper left corner of the window should be initially placed at location 0,0 on the screen. The constant **rectDefault** can replace the **CRect** parameter if desired.

At line 2, the code calls **CTestWindow::GetClientRect**, passing it the parameter **&r**. The **GetClientRect** function is inherited from the **CWnd** class (see the side-bar for search strategies to use when trying to look up functions in the Microsoft documentation). The variable **r** is of type **CRect** and is declared as a local variable at the beginning of the function.

Two questions arise here in trying to understand this code: 1) What does the **GetClientRect** function do? and 2) What does a **CRect** variable do? Let's start with question 1. When you look up the **CWnd::GetClientRect** function in the MFC documentation you find it returns a structure of type **CRect** that contains the size of the client rectangle of the particular window. It stores the structure at the address passed in as a parameter, in this case **&r**. That address should point to a location of type **CRect**. The **CRect** type is a class defined in MFC. It is a convenience class used to manage rectangles. If you look up the class in the MFC documentation, you will find that it defines over 30 member functions and operators to manipulate rectangles.

In our case, we want to center the words "Hello World" in the window. Therefore, we use **GetClientRect** to get the rectangle coordinates for the client area. In line 3 we then call **CRect::InflateRect**, which symmetrically increases or decreases the size of a rectangle (see also CRect::DeflateRect). Here we have decreased the rectangle by 20 pixels on all sides. Had we not, the border surrounding the label would have blended into the window frame, and we would not be able to see it.

The actual **CStatic** label is created in lines 4 and 5. The style attributes specify that the words displayed by the label should be centered and surrounded by a border. The size and position of the border is determined by the **CRect** parameter **r** .

By modifying the different style attributes you can gain an understanding of the different capabilities of the **CStatic** Object. For example, the code below contains a replacement for the **CTestWindow** constructor function in the first listing.

```
CTestWindow::CTestWindow() { CRect r; // Create the window itself Create(NULL,
"CStatic Tests", WS_OVERLAPPEDWINDOW, CRect(0,0,200,200)); // Get the size of the
client rectangle GetClientRect(&r); r.InflateRect(-20,-20); // Create a static label cs = new
CStatic(); cs->Create("Now is the time for all good men to \ come to the aid of their
country", WS_CHILD|WS_VISIBLE|WS_BORDER|SS_CENTER, r, this); }
```

The code above is identical to the previous except the text string is much longer. As you can see when you run the code, the **CStatic** object has wrapped the text within the specified bounding rectangle and centered each line individually.

If the bounding rectangle is too small to contain all the lines of text, then the text is clipped as needed to make it fit the available space. This feature of the **CStatic** object can be demonstrated by decreasing the size of the rectangle or increasing the length of the string.

In all the code we have seen so far, the style SS_CENTER has been used to center the text. The **CStatic** object also allows for left or right justification. Left justification is created by replacing the SS_CENTER attribute with an SS_LEFT attribute. Right justification aligns the words to the right margin rather than the left and is specified with the SS_RIGHT attribute.

One other text attribute is available. It turns off the word wrap feature and is used often for simple labels that identify other controls (see figure 3.1 for an example). The SS_LEFTNOWORDWRAP style forces left justification and causes no wrapping to take place.

Rectangular Display Modes for CStatic

The **CStatic** object also supports two different rectangular display modes: solid filled rectangles and frames. You normally use these two styles to visually group other controls within a window. For example, you might place a black rectangular frame in a window to collect together several related editable areas. You can choose from six different styles when creating these rectangles: SS_BLACKFRAME, SS_BLACKRECT, SS_GRAYFRAME, SS_GRAYRECT, SS_WHITEFRAME, and SS_WHITERECT. The RECT form is a filled rectangle, while the FRAME form is a border. The color names are a little misleading-for example, SS_WHITERECT displays a rectangle of the same color as the window background. Although this color defaults to white, the user can change it with the Control Panel and the rectangle may not be actually white on some machines.

When a rectangle or frame attribute is specified, the **CStatic** 's text string is ignored. Typically an empty string is passed. Try using several of these styles in the previous code and observe the result.

Fonts

You can change the font of a **CStatic** object by creating a **CFont** object. Doing so demonstrates how one MFC class can interact with another in certain cases to modify behavior of a control. The **CFont** class in MFC holds a single instance of a particular Windows font. For example, one instance of the **CFont** class might hold a Times font at 18 points while another might hold a Courier font at 10 points. You can modify the font used by a static label by calling the **SetFont** function that **CStatic** inherits from **CWnd**. The code below shows the code required to implement fonts.

```
CTestWindow::CTestWindow() { CRect r; // Create the window itself Create(NULL,
"CStatic Tests", WS_OVERLAPPEDWINDOW, CRect(0,0,200,200)); // Get the size of the
client rectangle GetClientRect(&r); r.InflateRect(-20,-20); // Create a static label cs = new
CStatic(); cs->Create("Hello World", WS_CHILD|WS_VISIBLE|WS_BORDER|SS_CENTER,
r, this); // Create a new 36 point Arial font font = new CFont; font->CreateFont(36,0,0,0,700,0,0,0, ANSI_CHARSET,OUT_DEFAULT_PRECIS,
CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY, DEFAULT_PITCH|FF_DONTCARE, "arial"); //
Cause the label to use the new font cs->SetFont(font); }
```

The code above starts by creating the window and the **CStatic** object as usual. The code then creates an object of type **CFont**. The font variable should be declared as a data member in the **CTestWindow** class with the line "CFont *font". The **CFont::CreateFont** function has 15 parameters (see the MFC help file), but only three matter in most cases. For example, the 36 specifies the size of the font in points, the 700 specifies the density of the font (400 is "normal," 700 is "bold," and values can range from 1 to 1000. The constants FW_NORMAL and FW_BOLD have the same meanings. See the FW constants in the API help file), and the word "arial" names the font to use. Windows typically ships with five True Type fonts (Arial, Courier New, Symbol, Times New Roman, and Wingdings), and by sticking to one of these you can be fairly certain that the font will exist on just about any machine. If you specify a font name that is unknown to the system, then the **CFont** class will choose the default font seen in all the other examples used in this tutorial.

For more information on the **CFont** class see the MFC documentation. There is also a good overview on fonts in the API on-line help file. Search for "Fonts and Text Overview."

The **SetFont** function comes from the **CWnd** class. It sets the font of a window, in this case the **CStatic** child window. One question you may have at this point is, "How do I know which functions available in **CWnd** apply to the **CStatic** class?" You learn this by experience. Take half an hour one day and read through all the functions in **CWnd** . You will learn quite a bit and you should find many functions that allow you to customize controls. We will see other **Set** functions found in the **CWnd** class in the next tutorial.

Conclusion

In this tutorial we looked at the many different capabilities of the **CStatic** object. We left out some of the **Set** functions inherited from the **CWnd** class so they can be discussed in Tutorial 4 where they are more appropriate.

Looking up functions in the Microsoft Documentation

In Visual C++ Version 5.x, looking up functions that you are unfamiliar with is very simple. All of the MFC, SDK, Windows API, and C/C++ standard library functions have all been integrated into the same help system. If you are uncertain of where a function is defined or what syntax it uses, just use the **Search** option in the Help menu. All occurrences of the function are returned and you may look through them to select the help for the specific function that you desire.

Compiling multiple executables

This tutorial contains several different example programs. There are two different ways for you to compile and run them. The first way is to place each different program into its own directory and then create a new project for each one. Using this technique, you can compile each program separately and work with each executeable simultaneously or independently. The disadvantage of this approach is the amount of disk space it consumes.

The second approach involves creating a single directory that contains all of the executables from this tutorial. You then create a single project file in that directory. To compile each program, you can edit the project and change its source file. When you rebuild the project, the new executable reflects the source file that you chose. This arrangement minimizes disk consumption, and is generally preferred.

PART 1 2 3

Introduction to MFC Programming with Visual C++ v5.x

by Marshall Brain

Message Maps

Any user interface object that an application places in a window has two controllable features: 1) its appearance, and 2) its behavior when responding to events. In the last tutorial you gained an understanding of the **CStatic** control and saw how you can use style attributes to customize the appearance of user interface objects. These concepts apply to all the different control classes available in MFC.

In this tutorial we will examine the **CButton** control to gain an understanding of message maps and simple event handling. We'll then look at the **CScrollBar** control to see a somewhat more involved example.

Understanding Message Maps

As discussed in Tutorial 2, MFC programs do not contain a main function or event loop. All of the event handling happens "behind the scenes" in C++ code that is part of the **CWinApp** class. Because it is hidden, we need a way to tell the invisible event loop to notify us about events of interest to the application. This is done with a mechanism called a *message map*. The message map identifies interesting events and then indicates functions to call in response to those events.

For example, say you want to write a program that will quit whenever the user presses a button labeled "Quit." In the program you place code to specify the button's creation: you indicate where the button goes, what it says, etc. Next, you create a message map for the parent of the button-whenever a user clicks the button, it tries to send a message to its parent. By installing a message map for the parent window you create a mechanism to intercept and use the button's messages. The message map will request that MFC call a specific function whenever a specific button event occurs. In this case, a click on the quit button is the event of interest. You then put the code for quitting the application in the indicated function.

MFC does the rest. When the program executes and the user clicks the Quit button, the button will highlight itself as expected. MFC then automatically calls the right function and the program terminates. With just a few lines of code your program becomes sensitive to user events.

The CButton Class

The **CStatic** control discussed in Tutorial 3 is unique in that it cannot respond to user events. No amount of clicking, typing, or dragging will do anything to a **CStatic** control because it ignores the user completely. However, The **CStatic** class is an anomaly. All of the other controls available in Windows respond to user events in two ways. First, they update their appearance automatically when the user manipulates them (e.g., when the user clicks on a button it highlights itself to give the user visual feedback). Second, each different control tries to send messages to your code so the program can respond to the user as needed. For example, a button sends a *command message* whenever it gets clicked. If you write code to receive the messages, then your code can respond to user events.

To gain an understanding of this process, we will start with the **CButton** control. The code below demonstrates the creation of a button.

```
// button1.cpp #include <afxwin.h> #define IDB_BUTTON 100 // Declare the application
class class CButtonApp : public CWinApp { public: virtual BOOL InitInstance(); }; //
Create an instance of the application class CButtonApp ButtonApp; // Declare the main
window class class CButtonWindow : public CFrameWnd { CButton *button; public:
CButtonWindow(); }; // The InitInstance function is called once // when the application
first executes BOOL CButtonApp::InitInstance() { m_pMainWnd = new CButtonWindow();
m_pMainWnd->ShowWindow(m_nCmdShow); m_pMainWnd->UpdateWindow(); return
TRUE; } // The constructor for the window class CButtonWindow::CButtonWindow() {
CRect r; // Create the window itself Create(NULL, "CButton Tests",
WS_OVERLAPPEDWINDOW, CRect(0,0,200,200)); // Get the size of the client rectangle
GetClientRect(&r); r.InflateRect(-20,-20); // Create a button button = new CButton();
button->Create("Push me", WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON, r, this,
IDB_BUTTON); }
```

The code above is nearly identical to the code discussed in previous tutorials. The **Create** function for the **CButton** class, as seen in the MFC help file, accepts five parameters. The first four are exactly the same as those found in the **CStatic** class. The fifth parameter indicates the resource ID for the button. The resource ID is a unique integer value used to identify the button in the message map. A constant value IDB_BUTTON has been defined at the top of the program for this value. The "IDB_" is arbitrary, but here indicates that the constant is an ID value for a Button. It is given a value of 100 because values less than 100 are reserved for system-defined IDs. You can use any value above 99.

The style attributes available for the **CButton** class are different from those for the **CStatic** class. Eleven different "BS" ("Button Style") constants are defined. A complete list of "BS" constants can be found using **Search** on CButton and selecting the "button style" link. Here we have used the BS_PUSHBUTTON style for the button, indicating that we want this button to display itself as a normal push-button. We have also used two familiar "WS" attributes: WS_CHILD and WS_VISIBLE. We will examine some of the other styles in later sections.

When you run the code, you will notice that the button responds to user events. That is, it highlights as you would expect. It does nothing else because we haven't told it what to do. We need to wire in a message map to make the button do something interesting.

Creating a Message Map

The code below contains a message map as well as a new function that handles the button click (so the program beeps when the user clicks on the button). It is simply an extension of the prior code.

```
// button2.cpp #include <afxwin.h> #define IDB_BUTTON 100 // Declare the application
class class CButtonApp : public CWinApp { public: virtual BOOL InitInstance(); }; //
Create an instance of the application class CButtonApp ButtonApp; // Declare the main
window class class CButtonWindow : public CFrameWnd { CButton *button; public:
CButtonWindow(); afx_msg void HandleButton(); DECLARE_MESSAGE_MAP() }; // The
message handler function void CButtonWindow::HandleButton() { MessageBeep(-1); } //
The message map BEGIN_MESSAGE_MAP(CButtonWindow, CFrameWnd)
ON_BN_CLICKED(IDB_BUTTON, HandleButton) END_MESSAGE_MAP() // The InitInstance
function is called once // when the application first executes BOOL
CButtonApp::InitInstance() { m_pMainWnd = new CButtonWindow(); m_pMainWnd-
>ShowWindow(m_nCmdShow); m_pMainWnd->UpdateWindow(); return TRUE; } // The
constructor for the window class CButtonWindow::CButtonWindow() { CRect r; // Create
the window itself Create(NULL, "CButton Tests", WS_OVERLAPPEDWINDOW,
CRect(0,0,200,200)); // Get the size of the client rectangle GetClientRect(&r);
r.InflateRect(-20,-20); // Create a button button = new CButton(); button->Create("Push
me", WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON, r, this, IDB_BUTTON); }
```

Three modifications have been made to the code:

1. The class declaration for **CButtonWindow** now contains a new member

function as well as a macro that indicates a message map is defined for the class. The **HandleButton** function, which is identified as a message handler by the use of the **afx_msg** tag, is a normal C++ function. There are some special constraints on this function which we will discuss shortly (e.g., it must be **void** and it cannot accept any parameters). The DECLARE_MESSAGE_MAP macro makes the creation of a message map possible. *Both the function and the macro must be public*.

2. The **HandleButton** function is created in the same way as any member function. In this function, we called the **MessageBeep** function available from the Windows API.
3. Special MFC macros create a message map. In the code, you can see that the BEGIN_MESSAGE_MAP macro accepts two parameters. The first is the name of the specific class to which the message map applies. The second is the base class from which the specific class is derived. It is followed by an ON_BN_CLICKED macro that accepts two parameters: The ID of the control and the function to call whenever that ID sends a command message. Finally, the message map ends with the END_MESSAGE_MAP macro.

When a user clicks the button, it sends a command message containing its ID to its parent, which is the window containing the button. That is default behavior for a button, and that is why this code works. The button sends the message to its parent because it is a child window. The parent window intercepts this message and uses the message map to determine the function to call. MFC handles the routing, and whenever the specified message is seen, the indicated function gets called. The program beeps whenever the user clicks the button.

The ON_BN_CLICKED message is the only interesting message sent by an instance of the **CButton** class. It is equivilent to the ON_COMMAND message in the **CWnd** class, and is simply a convenient synonym for it.

Sizing Messages

In the code above, the application's window, which is derived from the **CFrameWnd** class, recognized the button-click message generated by the button and responded to it because of its message map. The ON_BN_CLICKED macro added into the message map (search for the **CButton** overview as well as the the ON_COMMAND macro in the MFC help file) specifies the ID of the button and the function that the window should call when it receives a command message from that button. Since the button automatically sends to its parent its ID in a command message whenever the user clicks it, this arrangement allows the code to handle button events properly.

The frame window that acts as the main window for this application is also capable of sending messages itself. There are about 100 different messages available, all inherited from the **CWnd** class. By browsing through the member functions for the **CWnd** class in MFC help file you can see what all of these messages are. Look for any member function beginning with the word "On".

You may have noticed that all of the code demonstrated so far does not handle re-sizing very well. When the window re-sizes, the frame of the window adjusts accordingly but the contents stay where they were placed originally. It is possible to make resized windows respond more attractively by recognizing resizing events. One of the messages that is sent by any window is a sizing message. The message is generated whenever the window changes shape. We can use this message to control the size of child windows inside the frame, as shown below:

```
// button3.cpp #include <afxwin.h> #define IDB_BUTTON 100 // Declare the application
class class CButtonApp : public CWinApp { public: virtual BOOL InitInstance(); }; //
Create an instance of the application class CButtonApp ButtonApp; // Declare the main
window class class CButtonWindow : public CFrameWnd { CButton *button; public:
CButtonWindow(); afx_msg void HandleButton(); afx_msg void OnSize(UINT, int, int);
DECLARE_MESSAGE_MAP() }; // A message handler function void
CButtonWindow::HandleButton() { MessageBeep(-1); } // A message handler function
void CButtonWindow::OnSize(UINT nType, int cx, int cy) { CRect r; GetClientRect(&r);
r.InflateRect(-20,-20); button->MoveWindow(r); } // The message map
BEGIN_MESSAGE_MAP(CButtonWindow, CFrameWnd) ON_BN_CLICKED(IDB_BUTTON,
HandleButton) ON_WM_SIZE() END_MESSAGE_MAP() // The InitInstance function is called
once // when the application first executes BOOL CButtonApp::InitInstance() {
m_pMainWnd = new CButtonWindow(); m_pMainWnd->ShowWindow(m_nCmdShow);
m_pMainWnd->UpdateWindow(); return TRUE; } // The constructor for the window class
CButtonWindow::CButtonWindow() { CRect r; // Create the window itself Create(NULL,
"CButton Tests", WS_OVERLAPPEDWINDOW, CRect(0,0,200,200)); // Get the size of the
client rectangle GetClientRect(&r); r.InflateRect(-20,-20); // Create a button button = new
CButton(); button->Create("Push me", WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON, r, this,
IDB_BUTTON); }
```

To understand this code, start by looking in the message map for the window. There you will find the entry ON_WM_SIZE. This entry indicates that the message map is sensitive to sizing messages coming from the **CButtonWindow** object. Sizing messages are generated on this window whenever the user re-sizes it. The messages come to the window itself (rather than being sent to a parent as the ON_COMMAND message is by the button) because the frame window is not a child.

Notice also that the ON_WM_SIZE entry in the message map has no parameters. As you can see in the MFC documentation under the **CWnd** class, *it is understood that the ON_WM_SIZE entry in the message map will always call a function named* ***OnSize*** *, and that function must accept the* *three parameters shown* . The **OnSize** function must be a member function of the class owning the message map, and the function must be declared in the class as an **afx_msg** function (as shown in the definition of the **CButtonWindow** class).

If you look in the MFC documentation there are almost 100 functions named "On..." in the **CWnd** class. **CWnd::OnSize** is one of them. All these functions have a corresponding tag in the message map with the form ON_WM_. For example, ON_WM_SIZE corresponds to **OnSize**. None of the ON_WM_ entries in the message map accept parameters like ON_BN_CLICKED does. The parameters are assumed and automatically passed to the corresponding "On..." function like **OnSize**.

To repeat, because it is important: The **OnSize** function always corresponds to the ON_WM_SIZE entry in the message map. You must name the handler function **OnSize**, and it must accept the three parameters shown in the listing. You can find the specific parameter requirements of any **On...** function by looking up that function in the MFC help file. You can look the function up directly by typing **OnSize** into the search window, or you can find it as a member function of the **CWnd** class.

Inside the **OnSize** function itself in the code above, three lines of code modify the size of the button held in the window. You can place any code you like in this function.

The call to **GetClientRect** retrieves the new size of the window's client rectangle. This rectangle is then deflated, and the **MoveWindow** function is called on the button. **MoveWindow** is inherited from **CWnd** and re-sizes and moves the child window for the button in one step.

When you execute the program above and re-size the application's window, you will find the button re-sizes itself correctly. In the code, the re-size event generates a call through the message map to the **OnSize** function, which calls the **MoveWindow** function to re-size the button appropriately.

Window Messages

By looking in the MFC documentation, you can see the wide variety of **CWnd** messages that the main window handles. Some are similar to the sizing message seen in the previous section. For example, ON_WM_MOVE messages are sent when a user moves a window, and ON_WM_PAINT messages are sent when any part of the window has to be repainted. In all of our programs so far, repainting has happened automatically because controls are responsible for their own appearance. If you draw the contents of the client area yourself with GDI commands (see the book "Windows NT Programming:

An Introduction Using C++ " for a complete explanation) the application is responsible for repainting any drawings it places directly in the window. In this context the ON_WM_PAINT message becomes important.

There are also some event messages sent to the window that are more esoteric. For example, you can use the ON_WM_TIMER message in conjunction with the **SetTimer** function to cause the window to receive messages at pre-set intervals. The code below demonstrates the process. When you run this code, the program will beep once each second. The beeping can be replaced by a number of useful processes.

```
// button4.cpp #include <afxwin.h> #define IDB_BUTTON 100 #define IDT_TIMER1 200
// Declare the application class class CButtonApp : public CWinApp { public: virtual BOOL
InitInstance(); }; // Create an instance of the application class CButtonApp ButtonApp; //
Declare the main window class class CButtonWindow : public CFrameWnd { CButton
*button; public: CButtonWindow(); afx_msg void HandleButton(); afx_msg void
OnSize(UINT, int, int); afx_msg void OnTimer(UINT); DECLARE_MESSAGE_MAP() }; // A
message handler function void CButtonWindow::HandleButton() { MessageBeep(-1); } //
A message handler function void CButtonWindow::OnSize(UINT nType, int cx, int cy) {
CRect r; GetClientRect(&r); r.InflateRect(-20,-20); button->MoveWindow(r); } // A
message handler function void CButtonWindow::OnTimer(UINT id) { MessageBeep(-1); }
// The message map BEGIN_MESSAGE_MAP(CButtonWindow, CFrameWnd)
ON_BN_CLICKED(IDB_BUTTON, HandleButton) ON_WM_SIZE() ON_WM_TIMER()
END_MESSAGE_MAP() // The InitInstance function is called once // when the application
first executes BOOL CButtonApp::InitInstance() { m_pMainWnd = new CButtonWindow();
m_pMainWnd->ShowWindow(m_nCmdShow); m_pMainWnd->UpdateWindow(); return
TRUE; } // The constructor for the window class CButtonWindow::CButtonWindow() {
CRect r; // Create the window itself Create(NULL, "CButton Tests",
WS_OVERLAPPEDWINDOW, CRect(0,0,200,200)); // Set up the timer
SetTimer(IDT_TIMER1, 1000, NULL); // 1000 ms. // Get the size of the client rectangle
GetClientRect(&r); r.InflateRect(-20,-20); // Create a button button = new CButton();
button->Create("Push me", WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON, r, this,
IDB_BUTTON); }
```

Inside the program above we created a button, as shown previously, and left its re-sizing code in place. In the constructor for the window we also added a call to the **SetTimer** function. This function accepts three parameters: an ID for the timer (so that multiple timers can be active simultaneously, the ID is sent to the function called each time a timer goes off), the time in milliseconds that is to be the timer's increment, and a function. Here, we passed NULL for the function so that the window's message map will route the function automatically. In the message map we have wired in the ON_WM_TIMER message, and it will automatically call the **OnTimer** function passing it the ID of the timer that went off.

When the program runs, it beeps once each 1,000 milliseconds. Each time the timer's increment elapses, the window sends a message to itself. The message map routes the message to the **OnTimer** function, which beeps. You can place a wide variety of useful code into this function.

Scroll Bar Controls

Windows has two different ways to handle scroll bars. Some controls, such as the edit control and the list control, can be created with scroll bars attached. When this is the case, the master control handles the scroll bars automatically. For example, if an edit control has its scroll bars active then, when the scroll bars are used, the edit control scrolls as expected without any additional code.

Scroll bars can also work on a stand-alone basis. When used this way they are seen as independent controls in their own right. You can learn more about scroll bars by referring to the **CScrollBar** section of the MFC reference manual. Scroll bar controls are created the same way we created static labels and buttons. They have four member functions that allow you to get and set both the range and position of a scroll bar.

The code shown below demonstrates the creation of a horizontal scroll bar and its message map.

```
// sb1.cpp #include <afxwin.h> #define IDM_SCROLLBAR 100 const int
MAX_RANGE=100; const int MIN_RANGE=0; // Declare the application class class
CScrollBarApp : public CWinApp { public: virtual BOOL InitInstance(); }; // Create an
instance of the application class CScrollBarApp ScrollBarApp; // Declare the main window
class class CScrollBarWindow : public CFrameWnd { CScrollBar *sb; public:
CScrollBarWindow(); afx_msg void OnHScroll(UINT nSBCode, UINT nPos, CScrollBar*
pScrollBar); DECLARE_MESSAGE_MAP() }; // The message handler function void
CScrollBarWindow::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar) {
MessageBeep(-1); } // The message map BEGIN_MESSAGE_MAP(CScrollBarWindow,
CFrameWnd) ON_WM_HSCROLL() END_MESSAGE_MAP() // The InitInstance function is
called once // when the application first executes BOOL CScrollBarApp::InitInstance() {
m_pMainWnd = new CScrollBarWindow(); m_pMainWnd->ShowWindow(m_nCmdShow);
m_pMainWnd->UpdateWindow(); return TRUE; } // The constructor for the window class
CScrollBarWindow::CScrollBarWindow() { CRect r; // Create the window itself
Create(NULL, "CScrollBar Tests", WS_OVERLAPPEDWINDOW, CRect(0,0,200,200)); // Get
the size of the client rectangle GetClientRect(&r); // Create a scroll bar sb = new
CScrollBar(); sb->Create(WS_CHILD|WS_VISIBLE|SBS_HORZ, CRect(10,10,r.Width()-
10,30), this, IDM_SCROLLBAR); sb->SetScrollRange(MIN_RANGE,MAX_RANGE,TRUE); }
```

Windows distinguishes between horizontal and vertical scroll bars and also supports an object called a *size box* in the **CScrollBar** class. A size box is a small square. It is formed at the intersection of a horizontal and vertical scroll bar and can be dragged by the mouse to automatically re-size a window. Looking at the code in listing 4.5, you can see that the **Create** function creates a horizontal scroll bar using the SBS_HORZ style. Immediately following creation, the range of the scroll bar is set for 0 to 100 using the two constants MIN_RANGE and MAX_RANGE (defined at the top of the listing) in the **SetScrollRange** function.

The event-handling function **OnHScroll** comes from the **CWnd** class. We have used this function because the code creates a horizontal scroll bar. For a vertical scroll bar you should use **OnVScroll**. In the code here the message map wires in the scrolling function and causes the scroll bar to beep whenever the user manipulates it. When you run the code you can click on the arrows, drag the thumb, and so on. Each event will generate a beep, but the thumb will not actually move because we have not wired in the code for movement yet.

Each time the scroll bar is used and **OnHScroll** is called, your code needs a way to determine the user's action. Inside the **OnHScroll** function you can examine the first parameter passed to the message handler, as shown below. If you use this code with the code above, the scroll bar's thumb will move appropriately with each user manipulation.

```
// The message handling function void CScrollBarWindow::OnHScroll(UINT nSBCode, UINT
nPos, CScrollBar* pScrollBar) { int pos; pos = sb->GetScrollPos(); switch ( nSBCode ) {
case SB_LINEUP: pos -= 1; break; case SB_LINEDOWN: pos += 1; break; case
SB_PAGEUP: pos -= 10; break; case SB_PAGEDOWN: pos += 10; break; case SB_TOP:
pos = MIN_RANGE; break; case SB_BOTTOM: pos = MAX_RANGE; break; case
SB_THUMBPOSITION: pos = nPos; break; default: return; } if ( pos < MIN_RANGE ) pos
= MIN_RANGE; else if ( pos > MAX_RANGE ) pos = MAX_RANGE; sb->SetScrollPos( pos,
TRUE ); }
```

The different constant values such as SB_LINEUP and SB_LINEDOWN are described in the **CWnd::OnHScroll** function documentation. The code above starts by retrieving the current scroll bar position using **GetScrollPos**. It then decides what the user did to the scroll bar using a switch statement. The constant value names imply a vertical orientation but are used in horizontal scroll bars as well: SB_LINEUP and SB_LINEDOWN apply when the user clicks the left and right arrows. SB_PAGEUP and SB_PAGEDOWN apply when the user clicks in the shaft of the scroll bar itself. SB_TOP and SB_BOTTOM apply when the user moves the thumb to the top or bottom of the bar. SB_THUMBPOSITION applies when the user drags the thumb to a specific position. The code adjusts the position accordingly, then makes sure that it's still in range before setting the scroll bar to its new position. Once the scroll bar is set, the thumb moves on the screen to inform the user visually.

A vertical scroll bar is handled the same way as a horizontal scroll bar except that you use the SBS_VERT style and the **OnVScroll** function. You can also use several alignment styles to align both the scroll bars and the grow box in a given client rectangle.

Understanding Message Maps

The message map structure is unique to MFC. It is important that you understand why it exists and how it actually works so that you can exploit this structure in your own code.

Any C++ purist who looks at a message map has an immediate question: Why didn't Microsoft use virtual functions instead? Virtual functions are the standard C++ way to handle what mesage maps are doing in MFC, so the use of rather bizarre macros like DECLARE_MESSAGE_MAP and BEGIN_MESSAGE_MAP seems like a hack.

MFC uses message maps to get around a fundamental problem with virtual functions. Look at the **CWnd** class in the MFC help file. It contains over 200 member functions, all of which would have to be virtual if message maps were not used. Now look at all of the classes that subclass the **CWnd** class. For example, go to the contents page of the MFC help file and look at the visual object hierarchy. 30 or so classes in MFC use **CWnd** as their base class. This set includes all of the visual controls such as buttons, static labels, and lists. Now imagine that MFC used virtual functions, and you created an application that contained 20 controls. Each of the 200 virtual functions in **CWnd** would require its own virtual function table, and each instance of a control would therefore have a set of 200 virtual function tables associated with it. The program would have roughly 4,000 virtual function tables floating around in memory, and this is a problem on machines that have memory limitations. Because the vast majority of those tables are never used, they are unneeded.

Message maps duplicate the action of a virtual function table, but do so on an on-demand basis. When you create an entry in a message map, you are saying to the system, "when you see the specified message, please call the specified function." Only those functions that actually get overridden appear in the message map, saving memory and CPU overhead.

When you declare a message map with DECLARE_MESSAGE_MAP and BEGIN_MESSAGE_MAP, the system routes all messages through to your message map. If your map handles a given message, then your function gets called and the message stops there. However, if your message map does not contain an entry for a message, then the system sends that message to the class specified in the second parameter of BEGIN_MESSAGE_MAP. That class may or may not handle it and the proces repeats. Eventually, if no message map handles a given message, the message arrives at a default handler that eats it.

Conclusion

All the message handling concepts described in this tutorial apply to every one of the controls and windows available in NT. In most cases you can use the ClassWizard to install the entries in the message map, and this makes the task much easier. For more information on the ClassWizard, AppWizard and the resource editors see the tutorials on these topics on the MFC Tutorials page.

PART 1 2 3

Chapter 1

Tool Bar and Dialog Bar

Tool bar and dialog bar are used extensively in all types of applications. They provide users with a better way of executing application commands. Generally a tool bar comprises a series of buttons; each button represents a specific command. A command implemented by the tool bar can be linked directly to a menu command, in which case the two items share a same command ID. Both menu and tool bar handle WM_COMMAND message for executing commands. Besides, they also handle UPDATE_COMMAND_UI message to set the state of a button or a menu item. In fact, this message is very effective in enabling, disabling, and setting checked or unchecked state for a command.

While tool bar usually contains bitmap buttons, dialog bar can include many other type of controls that can be used in a dialog box, such as edit control, spin control, etc. Both tool bar and dialog bar can be implemented either as floated or docked, this gives users more choices in customizing the user interface of an application.

In MFC, classes that can be used to implement the tool bar and dialog bar are CToolBar and CDialogBar respectively. Both of them are derived from class CControlBar, which implements bar creation, command message mapping, control bar docking and floating (both tool bar and dialog bar are called control bar). Besides the default attributes, class CToolBar further supports bitmap button creation, automatic size adjustment for different states (docked or floated). A dialog bar can be treated as a dialog box (There is one difference here: a dialog bar can be either docked or floated, a dialog box does not have this feature): its implementation is based on a dialog template; all the common controls supported by dialog box can also be used in a dialog bar; their message mapping implementations are exactly the same.

A standard SDI or MDI application created by Application Wizard will have a default dockable tool bar. From now on we will discuss how to add extra tool bars and dialog bars, how to implement message mapping for the controls contained in a control bar, and how to customize their default behavior.

1.1. Adding an Extra Docking Tool Bar

Default Tool Bar

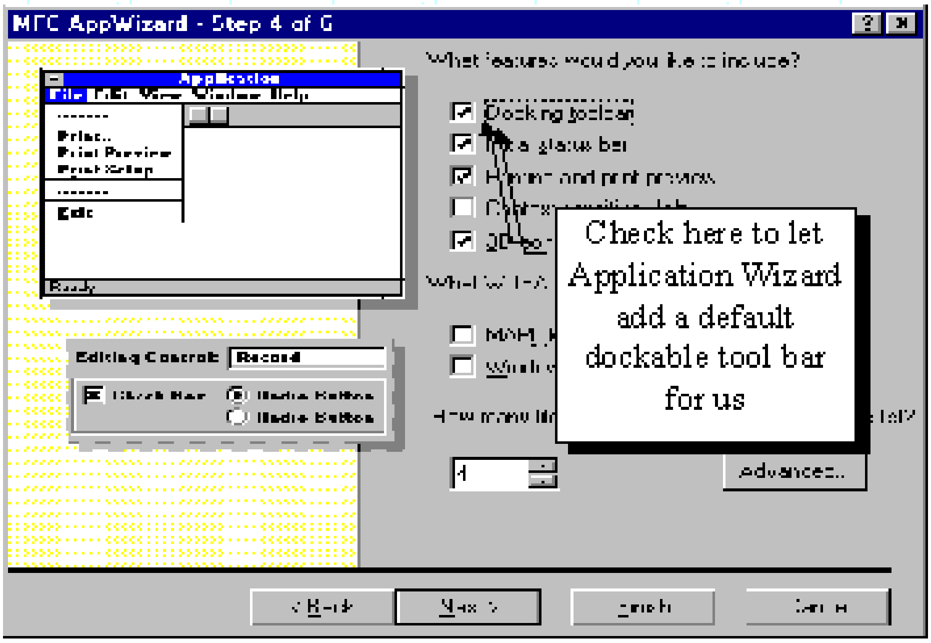
When using Application Wizard to generate SDI or MDI application skeleton, we can ask it to create a default docking tool bar for us. This can be done in the step 4 (see Figure 1-1). The default tool bar shares the same ID with the mainframe menu. It has eight bitmap buttons, which are all shortcuts to the menu commands. After executing this application, we will see a tool bar docked to the top border of the mainframe window. By using the mouse, we can easily either float it or dock it to other borders.

The tool bar resource can be opened in the Developer Studio. If we click "ResourceView" tab at the bottom of "Workspace" window, all the resources being used by the application will be listed within the window. If we asked Application Wizard to add a default tool bar for us, we will see a "Toolbar" resource node. By expanding this node (clicking on "+" node button or double clicking on the label), we will see all tool bar resources used by the application. If we double click on "IDR_MAINFRAME" ID, the tool bar bitmap will be displayed in "Tool Bar Edit" window. We can edit or delete an existing bitmap (each bitmap will be used to create a bitmap button). We can also add new bitmaps and assign them command IDs. When doing this, we can either use an existing menu command ID or a newly created one. In the latter case, we need to implement message mapping afterwards.

The Application Wizard does an excellent job in adding a very powerful tool bar. Nevertheless, as a programmer, we are kept from knowing what makes all these happen. If we need to make changes to the default tool bar (for example, we want it to be docked to the bottom border instead of top border at the beginning), which part of the source code should we modify? Obviously, we need to understand the essentials of tool bar implementation in order to customize it.

Like menu, generally tool bar is implemented in the mainframe window. When creating a mainframe menu, we need to prepare a menu resource, use class CMenu to declare a variable, then use it to load the menu resource. Creating a tool bar takes similar steps: we need to prepare a tool bar resource, use class CToolBar to declare a variable, which can be used to load the tool bar resource. After the tool bar resource is loaded successfully, we can call a series of member functions of CToolBar to create the tool bar and customize its styles.

After creating a standard SDI or MDI application using Application Wizard (with "Docking toolbar" check box checked in step 4, see Figure 1-1), we will find that a CToolBar type variable is declared in class CMainFrame:



What features would you like to include?

☑ Docking toolbar

☑ Initial status bar

☑ Printing and print preview

☐ Context-sensitive help

☑ 3D controls

Check here to let Application Wizard add a default dockable tool bar for us

Figure 1-1: Let Application Wizard add a default dockable tool bar

```
class CMainFrame : public CFrameWnd

{

......

protected: // control bar embedded members

CStatusBar m_wndStatusBar;

CToolBar m_wndToolBar;

......

}
```

The newly declared variable is m_wndToolBar. By tracing this variable, we will find out how the tool bar is implemented.

Tool Bar Implementation

Tool bar creation occurs in function CMainFrame::OnCreate(...), where the mainframe window is being created. Tool bar is created after function CFrameWnd::OnCreate(...) is called, which creates the default mainframe window. Creating a tool bar takes following steps:

1. Call CToolBar::Create(...) to create tool bar window.
2. Call CToolBar::LoadToolBar(...) to load the tool bar resource. We need to pass a tool bar resource ID to this function.
3. Call CToolBar::SetBarStyle(...) to set the attributes of the tool bar. For example, by setting different flags, we can let the tool bar have fixed or dynamic size. Besides this, we can also enable tool tips and flybys for tool bar buttons.
4. To make the tool bar dockable, we need to call function CToolBar::EnableDocking(...) and pass appropriate flags to it indicating which borders the tool bar could be docked (We can make the tool bar dockable to all four borders, or only top border, bottom border, etc.)
5. To dock the tool bar, we need to call function CMainFrame::DockControlBar(...). If we have more than one tool bar or dialog bar, this function should be called for each of them.

We need above five steps to implement a tool bar and set its attributes.

Message Mapping

Since tool bars are used to provide an alternate way of executing commands, we need to implement message mapping for the controls contained in a tool bar. This will allow the message handlers to be called automatically as the user clicks a tool bar button. The procedure of implementing message mapping for a tool bar control is exactly the same with that of a menu item. In MFC, this is done through declaring an afx_msg type member function and adding macros such as ON_COMMAND and ON_UPDATE_COMMAND_UI.

The message mapping could be implemented in any of the four default classes derived from CWinApp, CFrameWnd, CView and CDocument. Throughout this book, we will implement most of the message mappings in document. This is because for document/view structure, document is the center of the application and should be used to store data. By executing commands within the document, we don’t bother to obtain data from other classes from time to time.

The following lists necessary steps of implementing message mapping:

1. Declare afx_msg type member functions.
2. Implement these member functions for message handling.
3. Add message mapping macros between BEGIN_MESSAGE_MAP and END_MESSAGE_MAP (which are generated by Application Wizard). We need to use ON_COMMAND macro to map WM_COMMAND message, and use ON_UPDATE_COMMAND_UI macro to implement user interface updating. The message mapping should have the following format:

```
BEGIN_MESSAGE_MAP(class name, base class name)

//{{AFX_MSG_MAP(class name)

ON_COMMAND(command ID, member function name)

ON_UPDATE_COMMAND_UI(command ID, member function name)

//}}AFX_MSG_MAP

END_MESSAGE_MAP()
```

Most of the time message mapping could be implemented through using Class Wizard. In this case we only need to select a command ID and confirm the name of message handler. Although Class Wizard does an excellent job in implementing message mapping, sometimes we still need to add it manually because Class Wizard is not powerful enough to handle all cases.

Adding New Tool Bar Resource

Now that we understand how the default tool bar is implemented, it is easy for us to add extra tool bars. We can declare CToolBar type variables in class CMainFrame, create tool bars and set their styles in function CMainFrame::OnCreate(...). Then we can map tool bar command IDs to member functions so that the commands can be executed by mouse clicking.

Sample 1.1-1\Bar and 1.1-2\Bar demonstrate the above procedure. In the two applications, apart from the default tool bar, an extra tool bar that has four different buttons is added. Each button is painted with a different color: red, green, blue and yellow. If we click on one of them, a message box will pop up telling us the color of the button.

First we need to use Application Wizard to create a standard SDI application named "Bar", leaving all the settings as default. This will generate an application with a mainframe menu, a dockable tool bar and a status bar. The default four class names are CBarApp, CMainFrame, CBarDoc, CBarView.

Before modifying source code to add the second tool bar, we need to prepare the tool bar resource. In order to do this, we need the following steps to create a tool bar resource that contains four bitmap buttons:

1. Load the application project into Developer Studio.
2. Execute **Insert | Resource…** command from the menu (or press CTRL+R keys). We will be prompted to select resource type from a dialog box. If we highlight "toolbar" node and click the button labeled "New", a new blank tool

bar resource "IDR_TOOLBAR1" will be added to the project. Since default ID doesn’t provide us much implication, usually we need to modify it so that it can be easily understood. In the samples, the newly added tool bar resource ID is changed to IDR_COLOR_BUTTON. This can be implemented by right clicking on "IDR_TOOLBAR1" node in WorkSpace window, and selecting "Properties" item from the popped up menu. Now a property sheet whose caption is "Toolbar properties" will pop up, which contains an edit box that allows us to modify the resource ID of the tool bar.

3. Using the edit tools supplied by the Developer Studio, add four buttons to the tool bar, paint bitmaps with red, green, blue and yellow colors, change their IDs to ID_BUTTON_RED, ID_BUTTON_GREEN, ID_BUTTON_BLUE, ID_BUTTON_YELLOW. The tool bar bitmap window could be activated by double clicking on the tool bar IDs contained in the WorkSpace window, the graphic tools and color can be picked from "Graphics" and "Colors" windows. If they are not available, we can enable them by customizing the Developer Studio environment by executing **Tools | Customize...** command from the menu.

Declaring New Member Variable

After the resource is ready, we can add code to implement the tool bar.

The first step is to declare a new variable using CToolBar in class CMainFrame:

```
class CMainFrame : public CFrameWnd

{

......

protected:

CStatusBar m_wndStatusBar;

CToolBar m_wndToolBar;

CToolBar m_wndColorButton;

......

};
```

The new variable is m_wndColorButton, it is added right after other two variables that are used to implement the default tool bar and status bar.

Next, we can open file "MainFrm.cpp" and go to function CMainFrame::OnCreateClient(...). In Developer Studio, the easiest way to locate a member function is to right click on the function name in "WorkSpace" window, then select "Go to Definition" menu item from the popped up menu. Let’s see how the default tool bar is created:

```
......

if (!m_wndToolBar.Create(this) ||

!m_wndToolBar.LoadToolBar(IDR_MAINFRAME))

{

TRACE0("Failed to create toolbar\n");

return -1;

}

......
```

Function CToolBar::Create(...) is called first to create the tool bar window. Then CToolBar::LoadToolBar(...) is called to load the bitmaps (contained in tool bar resource IDR_MAINFRAME). When calling function CToolBar::Create(...), we need to specify the parent window of the tool bar by providing a CWnd type pointer (Generally, a tool bar must be owned by another window). Because this function is called within the member function of class CMainFrame, we can use "this" as the pointer of parent window.

The following code fragment shows how the styles of the default tool bar are set:

```
m_wndToolBar.SetBarStyle

(

m_wndToolBar.GetBarStyle() | CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC

);
```

Function CToolBar::SetBarStyle(...) sets tool bar’s styles, which can be a combination of different style flags using bit-wise OR operation. Because we do not want to lose the default styles, first function CToolBar::GetBarStyle() is called to retrieve the default tool bar styles, then new styles are combined with the old ones using bit-wise OR operation. In the above code fragment, three new styles are added to the tool bar: first, flag CBRS_TOOLTIPS will enable tool tips to be displayed when the mouse cursor passes over a tool bar button and stay there for a few seconds; second, flag CBRS_FLYBY will cause the status bar to display a flyby about this button (For details of tool tip and flyby, see section 1.11); third, flag CBRS_SIZE_DYNAMIC will allow the user to dynamically resize the tool bar, if we do not specify this style, the dimension of the tool bar will be fixed.

The following statement enables a dockable tool bar:

```
m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
```

Function CToolBar::EnableDocking(…) makes the tool bar dockable. Here, flag CBRS_ALIGN_ANY indicates that the tool bar may be docked to any of the four boarders of the frame window. We may change it to CBRS_ALIGN_TOP, CBRS_ALIGN_BOTTOM, CBRS_ALIGN_LEFT, or different combinations of these flags, whose meanings are self-explanatory.

The dockable tool bar still can’t be docked if the frame window does not support this feature. We must call function CFrameWnd::EnableDocking(…) to support docking in the frame window and call CFrameWnd::DockControlBar(…) for each control bar to really dock it. The following code fragment shows how the two functions are called for the default tool bar:

```
EnableDocking(CBRS_ALIGN_ANY);

DockControlBar(&m_wndToolBar);
```

Like function CCtrlBar::EnableDocking(…), CFrameWnd::EnableDocking(…) uses the same parameters to specify where a control bar is allowed to be docked.

Creating New Tool Bar

We need to do the same thing for the newly declared variable m_wndColorButton. We can call the above-mentioned functions to create tool bar window, set its styles, enable docking, and dock it. The following code fragment shows the updated function CMainFrame::OnCreate(…):

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)

{

if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
```

```
return -1;

if (!m_wndToolBar.Create(this) ||

!m_wndToolBar.LoadToolBar(IDR_MAINFRAME))

{

TRACE0("Failed to create toolbar\n");

return -1;

}

if (!m_wndColorButton.Create(this) ||

!m_wndColorButton.LoadToolBar(IDR_COLOR_BUTTON))

{

TRACE0("Failed to create toolbar\n");

return -1;

}

if (!m_wndStatusBar.Create(this) ||

!m_wndStatusBar.SetIndicators(indicators,

sizeof(indicators)/sizeof(UINT)))

{

TRACE0("Failed to create status bar\n");

return -1;

}

m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |

CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);

m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);

m_wndColorButton.SetBarStyle(m_wndColorButton.GetBarStyle() |
CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);

m_wndColorButton.EnableDocking(CBRS_ALIGN_ANY);

EnableDocking(CBRS_ALIGN_ANY);

DockControlBar(&m_wndToolBar);

DockControlBar(&m_wndColorButton);

return 0;

}
```

By compiling and executing the sample application at this point, we can see that the tool bar has been created. The tool bar can be docked to one of the four borders of the mainframe window or be floated. If we dock the tool bar to either left or right border, we will see that the tool bar will automatically have a vertical layout. This feature is supported by class CToolBar, we don’t need to add any line of code in order to have it.

Command Message Mapping

The new tool bar looks very disappointing. Although we made much effort to add it, none of its buttons can be used to execute command. This is because we still haven’t implemented any message handler for the new commands, therefore the buttons will be disabled all the time.

In Windowsä applications, commands are executed through sending WM_COMMAND message. As the user clicks a menu command or a tool bar button, the system will send a WM_COMMAND message to the application. All the Windowsä messages have two parameters, WPARAM and LPARAM (They are nothing but two integers, as an application receives a message, it will also receive the message parameters). For WM_COMMAND message, its WPARAM parameter is used to store the control ID (Command ID, such as ID_BUTTON_RED in our samples), which can be examined by the application to make appropriate response.

In a general Windowsä application, message is received and processed by a callback function. When an application is initialized, it stores the address of the callback function in the system. When a message is generated, the system uses this address to call the callback function and pass the message as well as the associated parameters to the application. Besides processing the message, the application can also choose to pass the message to other applications in the system.

If an application has callback function, we can process message WM_COMMAND within it. A general callback function for this purpose looks like the following:

```
LONG APIENTRY CallBackProc
(
HWND hwnd,
UINT message,
DWORD wParam,
LONG lParam)
{
switch (message)
{
case WM_CREATE:
{
......
break;
}
case WM_COMMAND:
{
switch(wParam)
{
```

```
case ID_BUTTON_RED:
{
......
}
case ID_BUTTON_GREEN:
{
......
}
......
}
break;
}
{
......
}
```

There are many types of messages, so parameter message (second parameter of the above function) could be any of the predefined values. If we want to trap mouse clicking events on the tool bar buttons, we need to handle WM_COMMAND message. We can see that within the WM_COMMAND case of switch statement in the above example, parameter wParam is checked (It holds WPARAM parameter of the message). By comparing it with the IDs of our buttons, we are able to find out which command is being executed by the user.

MFC handles Windowsä message in a different way. Because MFC applications are built upon classes, it is more convenient to handle messages within class member functions instead of one big callback function. In MFC, this is achieved through message mapping: we can implement the functions that will be used to execute commands, and use macros defined in MFC to direct the messages into these member functions.

As mentioned before, doing message mapping generally takes three steps: declaring afx_msg type member functions, using ON_COMMAND and ON_UPDATE_COMMAND_UI macros to implement mappings, and implementing the member functions.

For WM_COMMAND type message, the message handling functions do not have any parameters and should return void type value (For other type of messages, the format of the functions may be different). The message mapping can be implemented by using ON_COMMAND macro, which has the following format:

```
ON_COMMAND(control ID, member function name)
```

For example, if we have a member function OnButtonRed() in class CBarDoc, and we want to map WM_COMMAND message to this function when the user clicks red button (whose ID is ID_BUTTON_RED), we can implement message mapping as follows:

```
BEGIN_MESSAGE_MAP(CBarDoc, CDocument)

ON_COMMAND(ID_BUTTON_RED, OnButtonRed)

END_MESSAGE_MAP()
```

Message mapping macros must be done between BEGIN_MESSAGE_MAP and END_MESSAGE_MAP. Please note that if we want member functions of other classes to receive the same message, we must implement message mapping for each class separately.

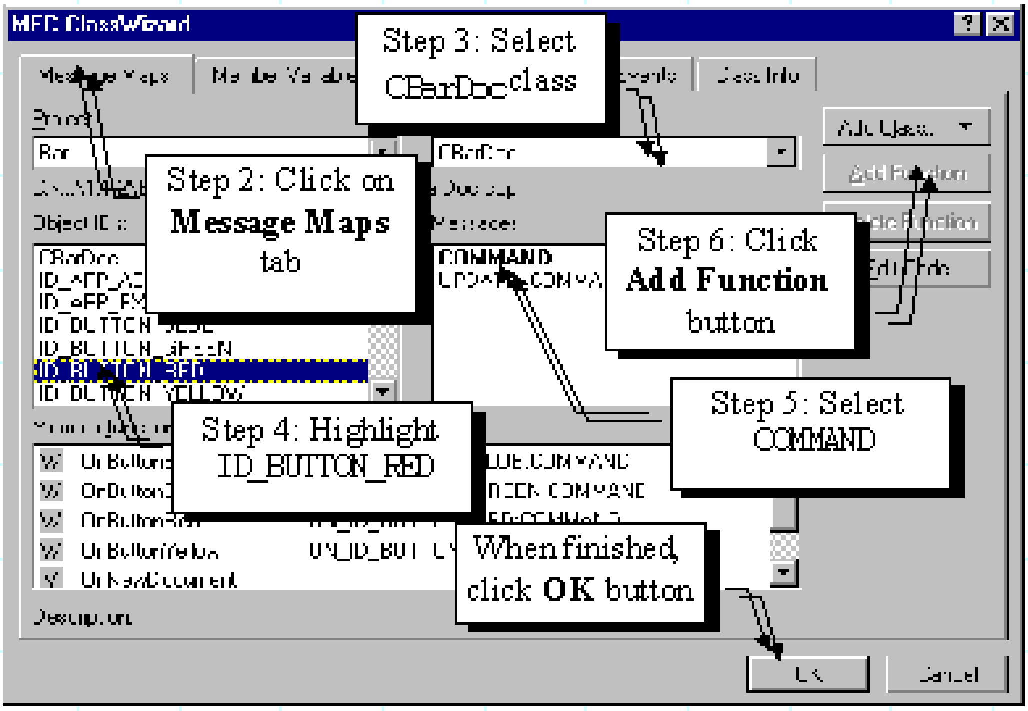
Class Wizard is designed to help us deal with message mapping. It provides us with a way of adding message handlers very easily: all we need to do is picking up messages and confirming the names of the member functions. The following descriptions list necessary steps of adding a message handler for button ID_BUTTON_RED in class CBarDoc through using Class Wizard (also see Figure 1-2):

1. In the Developer Studio, execute command **View | ClassWizard...** (or press CTRL+W keys).
2. From the popped up property sheet, click "Message Maps" tab (if the current page is not "Message Maps").

3. From "Class name" combo box, select "CBarDoc" if it is not the default class name (If the file being edited is "BarDoc.cpp", the default class name should be "CBarDoc").
4. From "Object Ids" window, highlight "ID_BUTTON_RED".
5. From "Messages" window, highlight "COMMAND".
6. Click "Add Function" button.
7. From the popped up dialog box, confirm the function name that will be used as the message handler (we may change the name according to our preference).
8. The function will be added to window "Member functions". Now we can repeat steps 4 through 7 to add message handlers for other IDs. When finished, we need to click "OK" button.

After dismissing the Class Wizard, the functions just added will be shown in the Developer Studio. By default, the message handlers are empty at the beginning, and we can add anything we want. For example, if we want a message box to pop up telling the color of the button when the user clicks it with mouse, we may implement the ID_BUTTON_RED message hander as follows:

```
void CBarDoc::OnButtonRed()

{

AfxMessageBox("Red");

}
```



Step 3: Select CBarDoc class

Step 2: Click on **Message Maps** tab

Step 6: Click **Add Function** button

Step 5: Select COMMAND

Step 4: Highlight ID_BUTTON_RED

When finished, click **OK** button

Figure 1-2. Implement message mapping through using Class Wizard

Similarly, we can write message handlers for other three buttons. In sample application 1.1-2\Bar, message handlers for all the four buttons on tool bar IDR_COLOR_BUTTON are implemented. If the user clicks any of the four buttons, a message box will pop up telling its color.

1.2. Imitating the Behavior of Radio Buttons

The buttons can be used more than just executing commands. Actually, they can serve other purposes such as status indication. We can use the buttons contained in a tool bar to imitate other two types of buttons used in dialog box: check box and radio button. When a check box is clicked, it will toggle between *Checked* and *Unchecked* states. For a radio button, only the unchecked button will toggle to checked state after being clicked. Also, at any time, only one radio button within a group can be checked.

Radio Button & Check Box

Although check box, radio button and push button look very differently from one another, they are essentially same type of controls. All of them can handle command messages, and their status can be set using the same function. The only difference among them is how they behave after being clicked by mouse. For a push button, after being clicked, it will go to checked state, as the mouse releases, it will automatically return to its normal state; for a check box, it toggles between checked and unchecked states after being clicked (it does not respond to mouse button release events); for a radio button, checking any button in a group will uncheck the previously checked button, so that at any time, only one button within a group could be checked.

On a tool bar, implementing normal check box and radio button does not make much sense, so they are not included as default features. If we want to add these controls, we can use dialog bar rather than tool bar. However, if we want the features of radio button and check box, we can use normal push buttons to imitate the behaviors of two types of controls.

The key of letting a push button behave like radio button and check box is to find a way of setting the states of buttons contained in the tool bar. In MFC, the states of tool bar buttons are managed in the same way with that of menu items. We can set a button’s state by trapping user-interface update command message (UPDATE_COMMAND_UI), then calling member functions of class CCmdUI to change a button’s state. In order to map this massage to a member function, we need to use ON_UPDATE_COMMAND_UI macro. The following is the format of this message mapping:

```
ON_UPDATE_COMMAND_UI(command ID, member function name)
```

The format of the corresponding member function is:

```
afx_msg void FunctionName(CCmdUI *pCmdUI);
```

The function has only one parameter, which is the pointer to a CCmdUI object. Class CCmdUI handles user-interface updating for tool bar buttons and menu items. Some most commonly used member functions are listed in the following table:

Function	Usage
CCmdUI::Enable(...)	Enable or disable a control
CCmdUI::SetCheck(...)	Set or remove the check state of a control
CCmdUI::SetRadio(...)	Set check state of a control, remove check state of all other controls in the group

From time to time, the operating system will send user-interface update command messages to the application, if there exists macros implementing the above-mentioned message mapping for any control contained in the tool bar, the control's state can be set within the corresponding message handler.

For a concrete example, if we want to disable ID_BUTTON_RED button under certain situations, we can declare a member function OnUpdateButtonRed(...) in class CBarDoc as follows (of course, we can also handle this message in other three classes):

```
class CBarDoc : public CDocument

{

......

protected:

afx_msg void OnUpdateButtonRed(CCmdUI *pCmdUI);

......

};
```

The message mapping can be implemented in file "BarDoc.cpp":

```
BEGIN_MESSAGE_MAP(CBarDoc, CDocument)
```

```
ON_UPDATE_COMMAND_UI(ID_BUTTON_RED, OnUpdateButtonRed)

END_MESSAGE_MAP()
```

The member function can be implemented as follows:

```
void CBarDoc::OnUpdateButtonRed(CCmdUI *pCmdUI)

{

if(under certain situations)pCmdUI->Enable(FALSE);

else pCmdUI->Enable(TRUE);

}
```

Usually we use a Boolean type variable as the flag, which represents "certain situations" in the above if statement. We can toggle this flag in other functions, this will cause the button state to be changed automatically. By doing this, the button's state is synchronized with the variable.

To set check state for a button, all we need to do is calling function CCmdUI::SetCheck(...) instead of CCmdUI::Enable(...) in the message handler.

Sample

Sample 1.2\Bar demonstrates how to make the four color buttons behave like radio buttons. At any time, one and only one button will be set to checked state (it will recess and give the user an impression that it is being held down).

To implement this feature, a member variable m_uCurrentBtn is declared in class CBarDoc. The value of this variable could be set to any ID of the four buttons in the member functions (other values are not allowed). In the user-interface update command message handler of each button, we check if the value of m_nCurrentBtn is the same with the corresponding button's ID. If so, we need to set check for this button, otherwise, we remove its check.

The following lists the steps of how to implement these message handlers:

1. Open file "BarDoc.h", declare a protected member variable m_uCurrentBtn in class CBarDoc:

```
class CBarDoc : public CDocument
```

```
{
......
protected:
UINT m_uCurrentBtn;
......
};
```

2. Go to file "BarDoc.cpp", in CBarDoc's constructor, initialize m_uCurrentBtn red button's resource ID:

```
CBarDoc::CBarDoc()
{
m_uCurrentBtn=ID_BUTTON_RED;
}
```

This step is necessary because we want one of the buttons to be checked at the beginning.

3. Implement UPDATE_COMMAND_UI message mapping for four button IDs. This is almost the same with adding ON_COMMAND macros, which could be done through using Class Wizard. The only difference between two implementations is that they select different message types from "Message" window (see step 5 previous section). Here we should select "UPDATE_COMMAND_UI" instead of "COMMAND".
1. Implement the four message handlers as follows:

```
void CBarDoc::OnUpdateButtonBlue(CCmdUI* pCmdUI)
{
pCmdUI->SetCheck(pCmdUI->m_nID == m_uCurrentBtn);
}
```

```
void CBarDoc::OnUpdateButtonGreen(CCmdUI* pCmdUI)

{

pCmdUI->SetRadio(pCmdUI->m_nID == m_uCurrentBtn);

}

void CBarDoc::OnUpdateButtonRed(CCmdUI* pCmdUI)

{

pCmdUI->SetRadio(pCmdUI->m_nID == m_uCurrentBtn);

}

void CBarDoc::OnUpdateButtonYellow(CCmdUI* pCmdUI)

{

pCmdUI->SetRadio(pCmdUI->m_nID == m_uCurrentBtn);

}
```

One thing to mention here is that CCmdUI has a public member variable m_nID, which stores the ID of the control that is about to be updated. We can compare it with variable CBarDoc::m_uCurrentBtn and set the appropriate state of the control.

With the above implementation, the red button will be checked from the beginning. We need to change the value of variable m_uCurrentBtn in order to check another button. This should happen when the user clicks on any of the four buttons, which will cause the application to receive a WM_COMMAND message. In the sample, this will cause the message handlers CBarDoc::OnButtonRed(), CBarDoc::OnButtonBlue()... to be called. Within these member functions, we can change the value of m_uCurrentBtn to the coresponding command ID in order to check that button:

```
void CBarDoc::OnButtonBlue()

{

m_uCurrentBtn=ID_BUTTON_BLUE;

}

void CBarDoc::OnButtonGreen()
```

```
{
m_uCurrentBtn=ID_BUTTON_GREEN;
}
void CBarDoc::OnButtonRed()
{
m_uCurrentBtn=ID_BUTTON_RED;
}
void CBarDoc::OnButtonYellow()
{
m_uCurrentBtn=ID_BUTTON_YELLOW;
}
```

The message box implementation is removed here. By executing the sample application and clicking on any of the four color buttons, we will see that at any time, one and only one button will be in the checked state.

1.3. Check Box Implementation

Using Boolean Type Variables

Using the method discussed in section 1.2, it is very easy to implement check-box-like buttons. We can declare Boolean type variables for each control, and toggle their values between FALSE and TRUE in the WM_COMMAND message handlers. Within UPDATE_COMMAND_UI message handlers, we can set check for any button according to the corresponding value of the Boolean type variable.

Sample 1.3-1\Bar is implemented in this way. In this sample, first variable m_uCurrentBtn is removed, then WM_COMMAND and UPDATE_COMMAND_UI message handlers are made empty. Four new Boolean type variables are declared in class CBarDoc:

```
class CBarDoc : public CDocument
```

```
{
......
protected:
BOOL m_bBtnRed;
BOOL m_bBtnGreen;
BOOL m_bBtnBlue;
BOOL m_bBtnYellow;
......
}
```

Their values are initialized in the constructor:

```
CBarDoc::CBarDoc()
{
m_bBtnRed=FALSE;
m_bBtnGreen=FALSE;
m_bBtnBlue=FALSE;
m_bBtnYellow=FALSE;
}
```

Two types of message handlers (altogether eight member functions) are rewritten. The following shows the implementation of two member functions for button ID_BUTTON_RED:

```
void CBarDoc::OnButtonRed()
{
m_bBtnRed=!m_bBtnRed;
}
void CBarDoc::OnUpdateButtonRed(CCmdUI* pCmdUI)
```

```
{
pCmdUI->SetRadio(m_bBtnRed);
}
```

If we execute the application at this point, we will see that the four color buttons behave like check boxes.

Function CButton::SetButtonInfo(…)

Although this is a simple way to implement "check box" buttons, sometimes it is not efficient. Suppose we have ten buttons that we expect to behave like check boxes, for every button we need to add a Boolean type variable and implement a UPDATE_COMMAND_UI message handler. Although this is nothing difficult, it is not the most efficient way of doing it.

Class CToolBar has a member function that can be used to set the button styles. The function allows us to set button as a push button, separator, check box, or the start of a group of check boxes. We can also use it to associate an image with a button contained in the tool bar. The following is the format of this function:

```
void CToolBar::SetButtonInfo(int nIndex, UINT nID, UINT nStyle, int iImage);
```

To use this function, we need to provide the information about the button, the style flags, and the image information. Parameter nIndex indicates which button we are gong to customize. It is a zero-based index, and button 0 is the left most button or separator on the tool bar (a separator is also considered a button). Parameter nID specifies which command ID we want to associate with this button. Parameter nStyle could be one of the following values, which indicates button’s style:

Flag	Meaning
TBBS_BUTTON	push button
TBBS_SEPARATOR	separator
TBBS_CHECKBOX	check box
TBBS_GROUP	start of a group
TBBS_CHECKGROUP	start of a check box group

The last parameter iImage indicates which image will be used to create the bitmap button. This is also a zero-based number, which indicates the image index of the tool bar resource. In our case, the tool bar resource contains four images, which are simply painted red, green, blue and yellow. The images are indexed according to their sequence, which means the red image is image 0, the green image is image 1, and so on.

When we create a tool bar resource, it seems that a button's command ID and the associated image are fixed from the beginning. Actually both of them can be modified through calling the above function. We can assign any command ID and image to any button. Also, we can change a button to a separator. In a normal application, there is no need to call this function, so the button's command ID and image are set according to the tool bar resource.

We have no intention of changing the default arrangement of the buttons. What we need to do here is modifying the button's style, which is set to TBBS_BUTTON by default. Sample 1.3-2\Bar demonstrates how to modify this style. It is based on sample 1.3-1\Bar.

To implement the new sample, first we need to delete four old UPDATE_COMMAND_UI message handlers. This can be done through using Class Wizard, which will delete the declaration of message handlers and the message mapping macros. We need to remove the implementation of the functions by ourselves.

We can set the button's style after the tool bar is created. This can be implemented in function CMainFrame::OnCreate(…). The following portion of this function shows what is added in the sample application:

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)

{

……

m_wndColorButton.SetButtonInfo(0, ID_BUTTON_RED, TBBS_CHECKBOX, 0);

m_wndColorButton.SetButtonInfo(1, ID_BUTTON_GREEN, TBBS_CHECKBOX, 1);

m_wndColorButton.SetButtonInfo(2, ID_BUTTON_BLUE, TBBS_CHECKBOX, 2);

m_wndColorButton.SetButtonInfo(3, ID_BUTTON_YELLOW, TBBS_CHECKBOX, 3);

……
}
```

With this modification, all four buttons will behave like check boxes. Similarly, if we want them to behave like push buttons, we just need to use style flag TBBS_BUTTON.

The state of a button can be retrieved by another member function of CToolBar. It lets us find out a button's command ID, and current state (checked or unchecked, the associate image):

```
void CToolBar::GetButtonInfo(int nIndex, UINT& nID, UINT& nStyle, int& iImage);
```

At any time, we can call this function to obtain the information about buttons. No additional variable is needed to remember their current states.

The method discussed here can also be used to create radio buttons. In order to do so, we need to use TBBS_CHECKGROUP instead of TBBS_CHECKBOX flag when calling function CToolBar::SetButtonInfo(…).

1.4. Message Mapping for a Contiguous Range of Command IDs

Contiguous IDs

In the previous sections, we implemented message handler for every control. If we want to handle both WM_COMMAND and UPDATE_COMMAND_UI messages, we need to add two message handlers for each control. Although Class Wizard can help us with function declaration and adding mapping macros, we still need to type in code for every member function. If we have 20 buttons on the tool bar, we may need to implement 40 message handlers. So here the question is, is there a more efficient way to implement message mapping?

The answer is yes. As long as the button IDs are contiguous, we can write a single message handler and direct all the messages to it. To implement this, we need to use two new macros: ON_COMMAND_RANGE and ON_UPDATE_COMMAND_UI_RANGE, which correspond to ON_COMMAND and ON_UPDATE_COMMAND_UI respectively. The formats of the two macros are:

```
ON_COMMAND_RANGE(start ID, end ID, member function name)
```

```
ON_UPDATE_COMMAND_UI_RANGE(start ID, end ID, member function name)
```

The formats of the corresponding member functions are:

```
afx_msg void FunctionName(UINT uID);

afx_msg void FunctionName(CCmdUI *pCmdUI);
```

When we create tool bar resource, the control IDs are generated contiguously according to the sequence of creation. For example, if we first create the blue button, then the green button, the two IDs will have the following relationship:

```
ID_BUTTON_GREEN = ID_BUTTON_BLUE+1
```

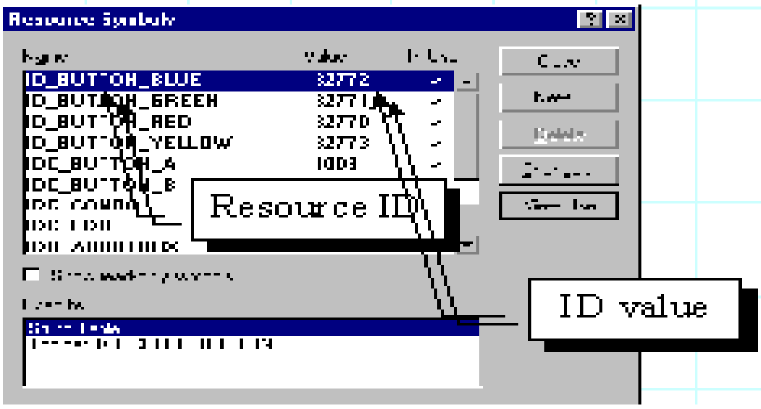
Modifying an ID

Sometimes we don’t know if the IDs of the tool bar buttons have contiguous values, because most of the time we use only symbolic IDs and seldom care about the actual values of them. If the IDs do not meet our requirement and we still want to use the above message mapping macros, we need to modify the ID values by ourselves.

By default, all the resource IDs are defined in file "resource.h". Although we could open it with a text editor and make changes, there is a better way to do so. First, an ID value could be viewed in the Developer Studio by executing **View | Resource symbols…** command. This command will bring up a dialog box that contains all the resource IDs used by the application (Figure 1-3).

If we want to make change to any ID value, first we need to highlight that ID, then click the button labeled "Change…". After that, a "Change Symbol" dialog box will pop up, if the ID is used for more than one purpose, we need to select the resource type in "Used by" window (This happens when this ID is used for both command ID and string ID, in which case the string ID may be used to implement flyby and tool tip. See Figure 1.9). In our sample, there is only one type of resource that uses the button IDs, so we do not need to make any choice. Now click "View Use" button (Figure 1-4), which will bring up "Toolbar Button Properties" property sheet. Within "General" page, we can change the ID’s value by typing in a new number in the window labeled "ID". For example, if we want to change the value of ID_BUTTON_RED to 32770, we just need to type in an equal sign and a number after the symbolic ID so that this edit window has the following contents (Figure 1-5):

```
ID_BUTTON_RED=32770
```



ID_BUTTON_BLUE	32772	✓
ID_BUTTON_GREEN	32771	✓
ID_BUTTON_RED	32770	✓
ID_BUTTON_YELLOW	32773	✓
IDC_BUTTON_A	1003	✓
IDC_BUTTON_B		

Figure 1-3. View resource IDs and their values

With this method, we can easily change the values of four resource IDs (ID_BUTTON_RED, ID_BUTTON_GREEN, ID_BUTTON_BLUE, ID_BUTTON_YELLOW) and make them contiguous. After this we can map all of them to a single member function instead of implementing message handlers for each ID.

Unfortunately, Class Wizard doesn’t do range mappings, so we have to implement it by ourselves. Sample 1.4\Bar demonstrates how to implement this kind of mapping. It is based upon sample 1.2\Bar, which already contains the default message mapping macros:

```
BEGIN_MESSAGE_MAP(CBarDoc, CDocument)

//{{AFX_MSG_MAP(CBarDoc)

ON_COMMAND(ID_BUTTON_BLUE, OnButtonBlue)

ON_COMMAND(ID_BUTTON_GREEN, OnButtonGreen)

ON_COMMAND(ID_BUTTON_RED, OnButtonRed)

ON_COMMAND(ID_BUTTON_YELLOW, OnButtonYellow)

ON_UPDATE_COMMAND_UI(ID_BUTTON_BLUE, OnUpdateButtonBlue)

ON_UPDATE_COMMAND_UI(ID_BUTTON_GREEN, OnUpdateButtonGreen)

ON_UPDATE_COMMAND_UI(ID_BUTTON_RED, OnUpdateButtonRed)

ON_UPDATE_COMMAND_UI(ID_BUTTON_YELLOW, OnUpdateButtonYellow)

//}}AFX_MSG_MAP

END_MESSAGE_MAP()
```

The following lists the necessary steps of changing message mapping from the original implementation to using contiguous IDs:

1. Delete the above eight message handlers along with the message mapping macros added by the Class Wizard.
2. Declare two new functions that will be used to process WM_COMMAND and ON_COMMAND_RANGE messages in class CBarDlg as follows:

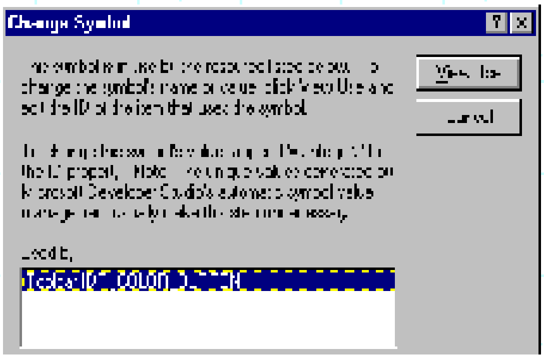
```
class CBarDoc : public CDocument

{


......

protected:

UINT m_uCurrentBtn;

//{{AFX_MSG(CBarDoc)

//}}AFX_MSG

afx_msg void OnButtons(UINT);

afx_msg void OnUpdateButtons(CCmdUI* pCmdUI);


......

}
```

3. Open file "BarDoc.cpp", find BEGIN_MESSAGE_MAP and END_MESSAGE_MAP macros of class CBarDoc, add the message mappings as follows:

```
BEGIN_MESSAGE_MAP(CBarDoc, CDocument)

//{{AFX_MSG_MAP(CBarDoc)

//}}AFX_MSG_MAP

ON_COMMAND_RANGE(ID_BUTTON_RED, ID_BUTTON_YELLOW, OnButtons)

ON_UPDATE_COMMAND_UI_RANGE(ID_BUTTON_RED, ID_BUTTON_YELLOW,
OnUpdateButtons)
```

```
END_MESSAGE_MAP()
```

In the sample application, the values of ID_BUTTON_RED, ID_BUTTON_GREEN, ID_BUTTON_BLUE, and ID_BUTTON_YELLOW are 32770, 32771, 32772, and 32773 respectively.



Change Symbol

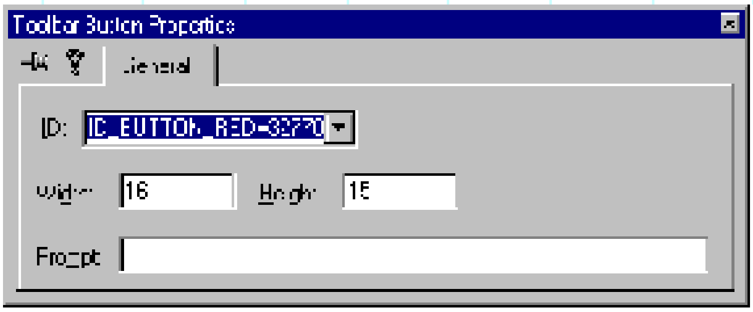
The symbol is in use by the resources listed below. To change the symbol's name or value, click View Use and edit the ID of the item that uses the symbol.

To [illegible] the [illegible] ID property. Note: [illegible] unique value correction [illegible] Microsoft Developer Studio's automatic symbol value management [illegible] unnecessary.

Used by:

Toolbar IDR_COLOR_[illegible]

Figure 1-4. Click "View Use" button to change the ID value



Toolbar Button Properties

General

ID: ID_BUTTON_RED=32770

Width: 16 Height: 15

Prompt:

Figure 1-5. Change the value of resource ID

4. Implement the two message handlers as follows:

```
void CBarDoc::OnButtons(UINT uID)

{

m_uCurrentBtn=uID;

}

void CBarDoc::OnUpdateButtons(CCmdUI* pCmdUI)

{

pCmdUI->SetRadio(pCmdUI->m_nID == m_uCurrentBtn);

}
```

Please compare the above code with the implementation in section 1.2. When we ask Class Wizard to add message mapping macros, it always adds them between //{{AFX_MSG comments. Actually, these comments are used by the Class Wizard to locate macros. To distinguish between the work done by ourselves and that done by Class Wizard, we can add the statements outside the two comments.

1.5. Fixing the Size of Tool Bar

Remember in section 1.1, when creating the color bar, we used CBRS_SIZE_DYNAMIC style:

```
m_wndToolBar.SetBarStyle

(

m_wndToolBar.GetBarStyle() | CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC

);
```

This allows the size of a tool bar to change dynamically. When the bar is floating or docked to top or bottom border of the client area, the buttons will have a horizontal layout. If the bar is docked to left or right border, they will have a vertical layout.

Sometimes we may want to fix the size of the tool bar, and disable the dynamic layout feature. This can be achieved through specifying CBRS_SIZE_FIXED flag instead of CBRS_SIZE_DYNAMIC flag when calling function CToolBar::SetBarStyle(...).

By default, the buttons on the tool bar will have a horizontal layout. If we fix the size of the tool bar, its initial layout will not change throughout application's lifetime. This will cause the tool bar to take up too much area when it is docked to either left or right border of the client area (Figure 1-6).

Instead of fixing the layout this way, we may want to wrap the tool bar from the second button, so the width and height of the tool bar will be roughly the same at any time (Figure 1-7).

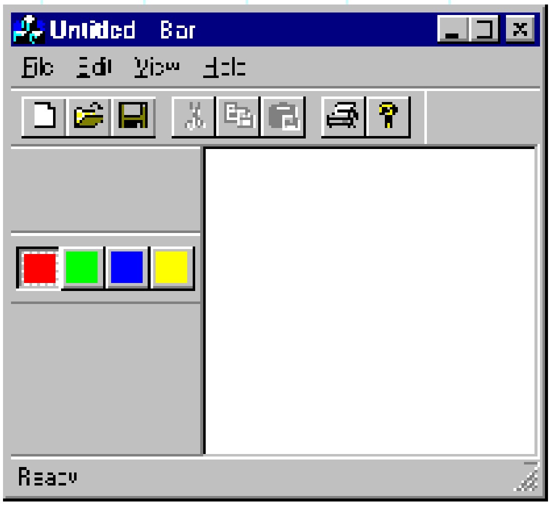


Figure 1-6: Docking a tool bar with horizontal layout to the left or right border will take up too much window area

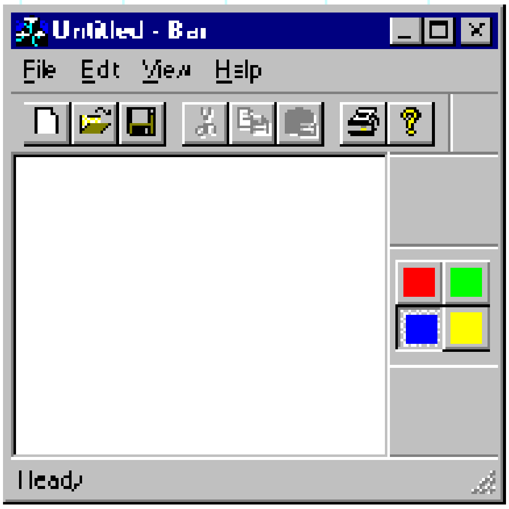


Figure 1-7. Fixing the layout this way will let a tool bar with fixed size take less area when it is docked to any border

We can call function CToolBar::SetButtonStyle(…) to implement the wrapping. This function has been discussed in section 1.3. However, there we didn't discuss the flag that can be used to wrap the tool bar from a specific button. This style is TBBS_WRAPPED, which is not documented in MFC.

Sample 1.5\Bar is based on sample 1.4\Bar that demonstrates this feature. The following shows the changes made to the original CMainFrame::OnCreate(…) function:

1. Replace CBRS_SIZE_DYNAMIC with CBRS_SIZE_FIXED when setting the tool bar style. The following statement shows this change:

```
m_wndColorButton.SetBarStyle

(
m_wndColorButton.GetBarStyle() | CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_FIXED
);
```

2. Add the following statement after this:

```
m_wndColorButton.SetButtonStyle
(
1, m_wndColorButton.GetButtonStyle(1) | TBBS_WRAPPED

);
```

To avoid losing default styles, in the second step, function CToolBar::GetButtonStyle(...) is first called to retrieve the original styles, which are bit-wise ORed with the new style before calling function CToolBar::SetButtonStyle(...).

1.6. Adding Combo Box to Tool Bar

By default, a tool bar can have only buttons and separators, and all the buttons must have the same size. This prevents us from adding other types of controls to the tool bar. However, by using some special properties of tool bar, we can still manage to add other types of common controls such as combo box to it.

Remember that all controls are actually different type of windows in essence. When we design a dialog template and add different common controls, we are given an impression that these controls are implemented "Statically". In fact, we can create any type of common controls by calling function CWnd::Create(...) at any time. This member function is supported by all the classes that are derived from CWnd. We can use it to create a control and put it anywhere on the tool bar.

Dynamically creating window is rarely used in normal programming because in this case the programmer has to calculate the size and position of the window carefully. If we implement this from a dialog template, we can see the visual effect immediately after a new control is added. If we implement this through function calling, we have to compile the project before we can see the final result.

However, because tool bar resource does not let us add controls other than buttons, dynamic method is the only way we can pursue to implement combo box on the tool bar. The question here is: because the buttons are positioned side by side, where can we put a combo box that will definitely take up a larger area?

To prevent the controls from interfering with each other, one control should not overlap another. So first, we must find an inactive area on the tool bar where we could create the combo box.

On the tool bar, separator is an inactive control. If we click mouse on it, there will be no response. We already know that we can call function CToolBar::SetButtonInfo(…) to change a button to a separator. Also, when doing this, we can specify the width of the separator by using iImage parameter (when we pass TBBS_SEPARATOR to nStyle parameter, the meaning of iImage parameter becomes the width of the separator).

On top of the separator, we can create any controls using dynamic method.

Sample 1.6\Bar demonstrates how to add combo box to the tool bar. This sample is based upon sample 1.4\Bar. In the new sample, the third button (blue button) is changed to a combo box. The following lists necessary steps of implementing this:

1. Change the blue button to a separator with a width of 150 after the tool bar is created. For this purpose, the following statement is added to function CMainFrame::OnCreate(…):

```
m_wndColorButton.SetButtonInfo(2, ID_BUTTON_BLUE, TBBS_SEPARATOR, 150);
```

Here the first parameter indicates that we want to modify the third button. The second parameter is the blue button’s resource ID. The fourth parameter specifies the width of the separator. If we compile and execute the sample at this point, we will see that the blue button does not exist anymore. Instead, a blank space with width of 150 is added between the third and fourth button. This is the place where we will create the combo box.

2. Use CComboBox to declare a variable m_wndComboBox in class CMainFrame as follows:

```
class CMainFrame : public CFrameWnd

{


……

protected:

CStatusBar m_wndStatusBar;
CToolBar m_wndToolBar;

CToolBar m_wndColorButton;

CComboBox m_wndComboBox;


......

}
```

3. Use the newly declared variable to call function CComboBox::Create(...) in CMainFrame::OnCreate(...) after the blue button has been changed to separator.

Function CComboBox::Create(...) has four parameters. We must specify combo box's style, size & position, parent window, along with the control ID in order to call this function. The following is the format of this function:

```
BOOL CComboBox::Create(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT
nID);
```

We can use the first parameter to set the styles of a combo box. A combo box can have different styles, in our sample, we just want to create a very basic drop down combo box (For other types of combo boxes, see Chapter 5). The following code fragment shows how this function is called within CMainFrame:: OnCreate(...):

```
......

m_wndColorButton.SetButtonInfo(2, ID_BUTTON_BLUE, TBBS_SEPARATOR, 150);

m_wndColorButton.GetItemRect(2, rect);

rect.bottom=rect.top+150;

if(!m_wndComboBox.Create(WS_CHILD | CBS_DROPDOWN |

CBS_AUTOHSCROLL | WS_VSCROLL | CBS_HASSTRINGS,

rect, &m_wndColorButton, ID_BUTTON_BLUE))

{ return -1;

}
```

```
m_wndComboBox.ShowWindow(SW_SHOW);
……
```

Function CToolBar::GetItemRect(…) is called in the second statement of above code to retrieve the size and position of the separator. After calling this function, the information is stored in variable rect, which is declared using class CRect.

A drop down combo box contains two controls: an edit box and a list box. Normally the list box is not shown. When the user clicks on the drop down button of the combo box, the list box will be shown. Because the size of the combo box represents its total size when the list box is dropped down (Figure 1-8), we need to extend the vertical dimension of the separator before using it to set the size of the combo box. The third statement of above code sets the rectangle’s vertical size to 150. So when our combo box is dropped down, its width and the height will be roughly the same.

The fourth statement of above code creates the combo box. Here a lot of styles are specified, whose meanings are listed below:

Style Flag	Meanings
WS_CHILD	Indicates that the window (combo box) being created is a child window. We must specify this flag in order to embed the combo box in another window
CBS_DROPDOWN	The combo box has a list control that can be dropped down by clicking its drop down button
CBS_AUTOHSCROLL	When the user types text into the edit control, the text will be automatically scrolled to the left if it is too long to be fully displayed
WS_VSCROLL	If too many items are added to the list controls and not all of them can be displayed at the same time, a vertical scroll bar will be added to the list control
CBS_HASSTRINGS	The items in the list control contains strings

The above styles are the most commonly used ones for a combo box. For details about this control, please refer to chapter 5.

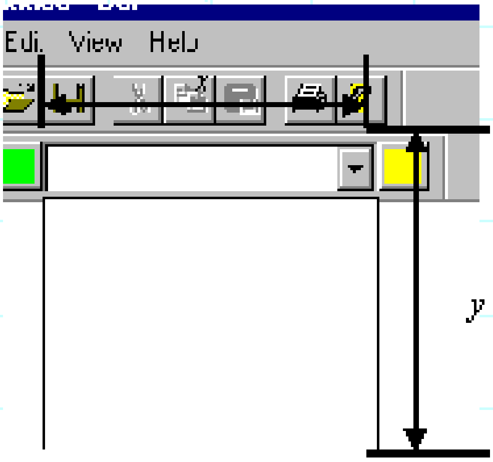


Figure 1-8. The size of a combo box is the total size when the list box is dropped down

Because the blue button will not be pressed to execute command anymore, we use ID_BUTTON_BLUE as the ID of the combo box. Actually, we can specify any other number so long as it is not used by other controls.

Finally, we must call function CWnd::ShowWindow(…) and pass SW_SHOW flag to it to show any window created dynamically.

By compiling and executing the sample at this point, we will see that the blue button has been changed to a combo box.

1.7. Modifying the Default Styles of Tool Bar

A tool bar with combo box is more useful than a normal one contains only bitmap buttons. However, this feature makes it difficult to implement dynamic layout. If we execute the sample created in the previous section and dock the tool bar to the left or right border, we will see that its layout becomes very awkward (Figure 1-9). This is because the layout feature of CTooBar is designed for a tool bar that contains only buttons with the same size. If we want to change this feature, we need to override the default layout implementation.

Because the combo box does not fit well when the tool bar has a vertical layout, we may want to change it back to the blue button when the tool bar is docked to the left or right border, and change the button back to combo box again when the bar is floated or docked to the top or bottom border.

This can be easily implemented by calling function CToolBar::SetButtonStyle(…) back and forth and setting the button's style according to the current layout. The problem here is that we must be notified when the tool bar's layout is about to change so that we can call the above function before the layout of tool bar actually changes.

When a tool bar's layout is about to change, function CToolBar::CalcDynamicLayout(...)will be called to retrieve the dimension of the tool bar. The default implementation of this function calculates the tool bar layout according to the sizes of the controls contained in the tool bar and tries to arrange them to let the tool bar have a balanced appearance. What we could do here is changing the combo box back to the button when this function is called for calculating tool bar's vertical layout size, and changing the button back to combo box when it is called for calculating the horizontal layout size.

We could override function CToolBar::CalcDynamicLayout(...)to make this change. The new function should be implemented as follows:

```
Overridden CalcDynamicLayout(...)

{

Change the combo box to button or vice versa if necessary;

CToolBar::CalcDynamicLayout(...);

}
```

The default implementation of this function is called after the button information is set correctly. By doing this way, the tool bar can always have the best layout appearance.

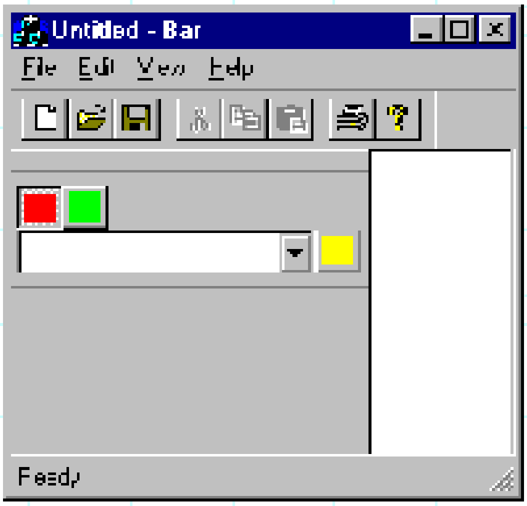


Figure 1-9. The default vertical layout of the custom tool bar looks awkward

We need to know when the tool bar will change from horizontal layout to vertical layout, or vice versa. This can be judged from the parameters passed to function CControlBar::CalcDynamicLayout(...). Let's take a look at the function prototype first:

```
CSize CControlBar::CalcDynamicLayout(int nLength, DWORD dwMode);
```

The function has two parameters, the second parameter dwMode indicates what kind of size is being retrieved. It can be the combination of many flags, in this section, we need to know only two of them:

Flag	Meanings
LM_HORZDOCK	The horizontal dock dimension is being retrieved
LM_VERTDOCK	The vertical dock dimension is being retrieved

What we need to do in the overridden function is examining the LM_HORZDOCK bit and LM_VERTDOCK bit of dwMode parameter and setting the button information correspondingly.

To override the member function of CToolBar, we must first derive a new class from it, then implement a new version of this function in the newly created class. Sample 1.7\Bar demonstrates how to change the button's style dynamically, it is based on sample 1.6\Bar.

First, we need to declare the new class, in the sample, this class is named CColorBar:

```
class CColorBar : public CToolBar

{

public:

CColorBar();

BOOL AddComboBox();

BOOL ShowComboBox();

BOOL HideComboBox();

virtual ~CColorBar();

virtual CSize CalcDynamicLayout(int , DWORD);

//{{AFX_VIRTUAL(CColorBar)
//}}AFX_VIRTUAL
protected:
CComboBox m_wndComboBox;
//{{AFX_MSG(CColorBar)
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
```

Instead of declaring a CComboBox type variable in class CMainFrame, here we implement the declaration in the derived class. This is a better implementation because the combo box should be made the child window of the tool bar. By embedding the variable here, we can make it a protected variable so that it is not accessible from outside the class.

Three functions are added to change the tool bar’s style. Function CColorBar::AddComboBox() changes the blue button to a separator and creates the combo box window:

```
BOOL CColorBar::AddComboBox()
{
CRect rect;
GetItemRect(2, rect);
rect.bottom=rect.top+150;
if
(
!m_wndComboBox.Create
(
WS_CHILD | CBS_DROPDOWN | CBS_AUTOHSCROLL | WS_VSCROLL | CBS_HASSTRINGS,
rect,
this,
ID_BUTTON_BLUE
)
)return FALSE;
else return TRUE;
}
```

This is the same with what we did in function CMainFrame::OnCreate(...) in the previous section. The only difference is that when creating the combo box within the member function of CMainFrame, the combo box's parent window needs to be set to m_wndColorButton. Here, since the combo box variable is embedded in the parent window's class, we need to use this pointer to indicate the parent window.

Function CColorBar::ShowComboBox() and CColorBar::HideComboBox() change the combo box to the button and vice versa. They should be called just before the default layout is about to be carried out:

```
BOOL CColorBar::ShowComboBox()
{
CRect rect;
SetButtonInfo(2, ID_BUTTON_BLUE, TBBS_SEPARATOR, 150);
if(m_wndComboBox.m_hWnd != NULL)
{
m_wndComboBox.ShowWindow(SW_SHOW);
}
return TRUE;
}
BOOL CColorBar::HideComboBox()
{
SetButtonInfo(2, ID_BUTTON_BLUE, TBBS_BUTTON, 2);
if(m_wndComboBox.m_hWnd != NULL)m_wndComboBox.ShowWindow(SW_HIDE);

return TRUE;

}
```

Finally, function CalcDynamicLayout(...) is overridden as follows:

```
CSize CColorBar::CalcDynamicLayout(int nLength, DWORD dwMode)

{

if(dwMode & LM_HORZDOCK)ShowComboBox();

else HideComboBox();

return CToolBar::CalcDynamicLayout(nLength, dwMode);

}
```

Before calling the base class version of this function, we examine LM_HORZDOCK bit of dwMode parameter, if it is set, we call function CColorBar::ShowComboBox() to change the button to the combo box. If not, we call function CColorBar::HideComboBox() to change the combo box back to the default button.

It is relatively easy to use this class: we just need to include the header file of CColorBar class in file "MainFrm.h", then change the prototype of m_wndColorBar from CToolBar to CColorBar. Because the combo box is embedded in CColorBar class, we need to remove variable wndComboBox declared in the previous section. In function CMainFrame::OnCreate(), instead of creating the combo box by ourselves, we can just call the member function of CColorBar. Here is how the combo box is created using this new method:

```
......

m_wndColorButton.AddComboBox();

m_wndColorButton.ShowComboBox();

......
```

We can see that the original five statements have been reduced to two statements.

Now we can compile and execute the sample again to see the behavior of the combo box.

1.8. Dialog Bar

As we have noticed, the limitation of the tool bar is that when we design a tool bar from resource, only bitmap buttons with the same size can be included. If we try to modify the size of one bitmap button, the size of all other buttons will be automatically adjusted. If we want to include controls other than buttons, we need to write code to add them dynamically.

If we want to implement a control bar that contains other types of common controls starting from resource editing, we need to use another type of control bar: dialog bar. In MFC, the class that can be used to implement this type of control bar is CDialogBar.

Like tool bar, dialog bar is also derived from control bar. Both of them share some common features. For example, both types of control bars can be either docked or floated, and they all support tool tip and flyby implementation. The difference between tool bar and dialog bar is that they are designed to contain different types of controls: while tool bar is more suitable for containing a row of bitmap buttons with the same size, dialog bar can be implemented to contain any type of controls that can be used in a dialog box.

Implementing dialog bar is similar to that of dialog box. The first step is to design a dialog-template resource. We can add buttons, edit boxes, combo boxes, even animate controls to a dialog bar.

Dialog bar shares the same type of resource with dialog box. So when starting to create resource for dialog bar, we first need to add a "dialog" type resource to the application (In order to do this, we can execute **Insert | Resource…** command, then select "Dialog" from the popped up "Insert Resource" dialog box). When specifying the dialog properties, we must set "child" and "no border" styles. This can be customized in "Dialog Properties" property sheet (Figure 1-10).

Sample 1.8\Bar demonstrates how to use dialog bar, it is based on sample 1.7\Bar. In this sample, besides the extra tool bar added in the previous sections, a new dialog bar that contains some common controls is added to the application. The resource ID of this dialog bar is ID_DIALOG_BAR. It contains two push buttons (ID_BUTTON_A, ID_BUTTON_B), one edit box (IDC_EDIT) and one combo box (IDC_COMBO). Also, there are other three static text controls (Figure 1-11).

Adding a dialog bar to the application is similar to that of a tool bar. First we need to declare a variable in class CMainFrame. Then within function CMainFrame::OnCreate(…), we can call the member functions of CDialogBar and

CFrameWnd to create the dialog bar, set its styles and dock it.

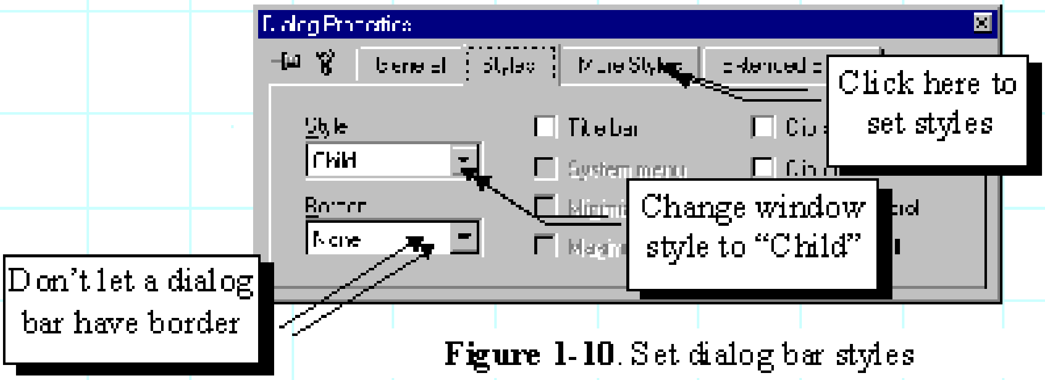
The following lists necessary steps of adding this dialog bar:

1. Use CDialogBar to declare a new variable m_wndDialogBar in CMainFrame class:

```
class CMainFrame : public CFrameWnd

{

......

protected:

......

CDialogBar m_wndDialogBar;

......

};
```

2. In function CMainFrame::Create(...), call CDialogBar::Create(...) to create the dialog bar window. When doing this, we need to provide the pointer of its parent window, the dialog template ID, styles and the control ID of the dialog bar. Here, the control ID could be a different number from its template ID, so long as it is not being occupied by other resources. The following code fragment shows how this function is called in the sample:

```
......

if

(

!m_wndDialogBar.Create

(

this,
IDD_DIALOG_COLORBAR,
CBRS_BOTTOM | CBRS_TOOLTIPS | CBRS_FLYBY,
IDD_DIALOG_COLORBAR
)
)
{
TRACE0("Failed to create toolbar\n");
return -1;
}
……
```



Style: Child

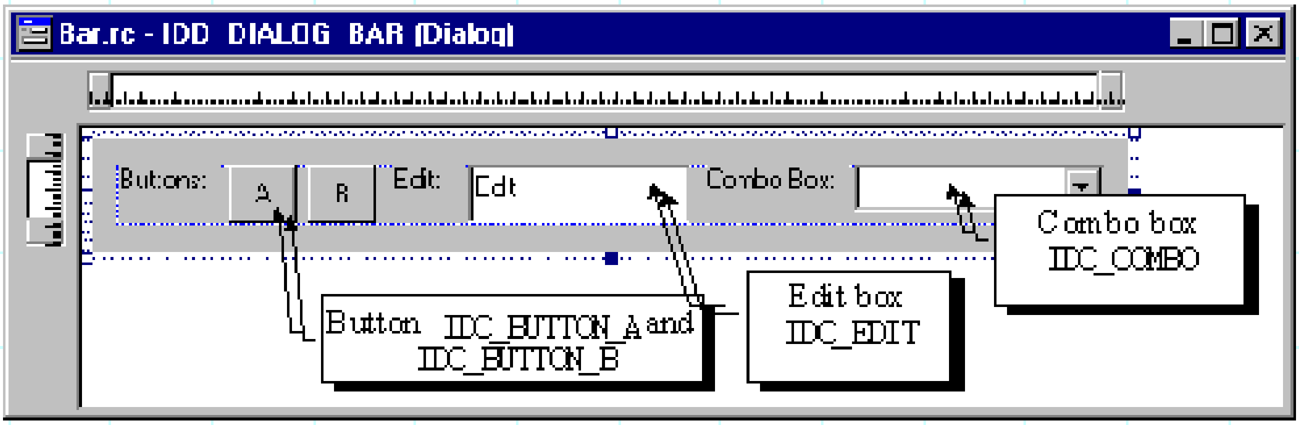
Border: None

Click here to set styles

Change window style to "Child"

Don't let a dialog bar have border

Figure 1-10. Set dialog bar styles



Button IDC_BUTTON_A and IDC_BUTTON_B

Edit box IDC_EDIT

Combo box IDC_COMBO

Figure 1-11. Dialog bar template resource

3. Enable docking by calling function `CDialogBar::EnableDocking(…)`, dock the dialog bar by calling function `CMainFrame::DockControlBar(…)`:

……

```
m_wndDialogBar.EnableDocking(CBRS_ALIGN_ANY);

……

DockControlBar(&m_wndDialogBar);

……
```

Because class CDialogBar is derived from CControlBar, in step 3, when we call function CDialogBar::EnableDocking(…) to enable docking for the dialog bar, we are actually calling function CControlBar::EnableDocking(…). This is the same with that of tool bar. Because of this, both tool bar and dialog bar have the same docking properties.

By compiling and executing the sample at this point, we can see that the dialog bar is implemented, which is docked to the bottom border at the beginning. We can drag the dialog bar and dock it to other borders. As we do this, we may notice the difference between dialog bar and tool bar: while the size of the tool bar will be automatically adjusted when it is docked differently (horizontally or vertically), the size of dialog will not change under any condition. The reason for this is that a dialog bar usually contains irregular controls, so it is relatively difficult to adjust its size automatically. By default, dynamic size adjustment is not supported by class CDialogBar. If we want our dialog bar to support this feature, we need to override function CDialogBar::CalcDynamicLayout(…).

To prevent a dialog bar from taking up too much area when it is docked to left or right border, we can put restriction on the dialog bar so that it can only be docked to top or bottom border. To implement this, we can change the style flag from CBRS_ALIGN_ANY to CBRS_ALIGN_TOP | CBRS_ALIGN_BOTTOM when calling function CDialogBar::EnableDocking(…) in step 3 discussed above.

1.9. Resizable Dialog Bar

Because dialog bar can contain more powerful controls than tool bar, we could use it to implement control bars with more flexibility. To make user-friendly interface, sometimes we may really want to dynamically change a dialog bar’s size.

Class CControlBar has two member functions dealing with control bar layout: CControlBar:: CalcFixedLayout(…) and CControlBar::CalcDynamicLayout(…). The first function returns the fixed size of a control bar, which is determined from the resource of a control bar. The second function is designed for implementing dynamic layout, however in class CControlBar, this function does noting but calling

CControlBar::CalcFixedLayout(...). So actually CControlBar does not have dynamic layout feature.

Class CToolBar overrides function CControlBar::CalcDynamicLayout(...), which adjusts the size of control bar according to its current docking state and the size of buttons. Whenever its docking state is changed, this function will be called to obtain the information of a tool bar before its layout is changed accordingly. With this implementation, a tool bar can always have a balanced appearance.

Unlike CToolBar, CDialogBar does not override this function, so when the docking state of a dialog bar is changed, the default CControlBar::CalcDynamicLayout(...) is called, which of course, will not perform any dynamic layout for the dialog bar. If we want to add any dynamic layout feature, we must override this member function.

Sample 1.9\Bar demonstrates how to build a dialog bar that can be resized dynamically. The application is a standard SDI application generated from Application Wizard, with four classes named CBarApp, CMainFrame, CBarDoc, and CBarView. In the sample, a dialog bar with an edit control is added to the application. This dialog bar will support dynamic layout feature: when we float or dock it to different borders of the mainframe window, the size of the dialog bar will change accordingly.

Deriving New Class from CDialogBar

To implement dialog bar, first we need to add a dialog-template resource. In the sample, the ID of the new resource is IDD_DIALOGBAR, which contains an edit box. The ID of this edit box is IDC_EDIT, it supports multiple-line editing. To enable this style, first we can open "Edit Properties" property sheet of the edit control (To invoke this property sheet, when editing the dialog template, we can double click on the edit control or right click on it and move to "Properties" menu item), then we need to click "Styles" tab and check "Multiline" check box.

To override the default function, first we need to derive a new class from CDialogBar. The following code fragment shows the derived class MCDialogBar, which resides in file "MDlgBar.h":

```
class MCDialogBar : public CDialogBar

{

public:

MCDialogBar();

//{{AFX_DATA(MCDialogBar)
//}}AFX_DATA

virtual CSize CalcDynamicLayout(int, DWORD);

//{{AFX_VIRTUAL(MCDialogBar)

//}}AFX_VIRTUAL

protected:

//{{AFX_MSG(MCDialogBar)

afx_msg void OnSize(UINT nType, int cx, int cy);

//}}AFX_MSG

DECLARE_MESSAGE_MAP()

};
```

Besides constructor, the only things included in this class are two functions. As we already know, MCDialogBar::CalcDynamicLayout(...) will be used to support dynamic layout. Another afx_msg type function MCDialogBar::OnSize(...) is a message handler, which will be used to resize the edit control contained in the dialog bar. By doing this, we can see that whenever the size of the dialog bar is adjusted, the size of the edit box will also change accordingly. This will let the edit box fit well within the dialog bar.

The new class can be added by opening new files (".h" and ".cpp" files) and typing in the new class and function implementations. Then we can execute **Project | Add To Project | Files...** command to add the newly created files to the project. However, if we do so, we can not use Class Wizard to add member variables and functions to the class. In this case, we need to implement message mapping manually. If this is our choice, we must make sure that DECLARE_MESSAGE_MAP() macro is included in the class, and BEGIN_MESSAGE_MAP, END_MESSAGE_MAP macros are included in the implementation file (".cpp" file) so that the class will support message mapping.

We can also use Class Wizard to add new class. In order to do this, after invoking the Class Wizard, we can click button labeled "Add Class..." then select "New..." from the popup menu. This will invoke a dialog box that lets us add a new class to the project. We can type in the new class name, select the header and implementation file names, and designate base class name. Unfortunately, CDialogBar is not in the list of base classes. A workaround is that we can select CDialog as the base class, after the class is generated, delete the unnecessary functions, and change all CDialog keywords to CDialogBar in both header and implementation files.

Resizing Edit Control

The edit control should be resized whenever the size of its parent window changes. In order to do this, we can trap WM_SIZE message, which is sent to a window when its size is about to change. We need to declare an afx_msg type member function as the message handler, and implement the message mapping using ON_WM_SIZE macro. The message handler of WM_SIZE should have the following format:

```
afx_msg void OnSize(UINT nType, int cx, int cy);
```

Here nType indicates how the window's size will be changed (is it maximized, minimized...), cx and cy indicate the new window size.

It is not very difficult to add message mapping macro, we can either add it manually, or ask Class Wizard to do it for us:

```
BEGIN_MESSAGE_MAP(MCDialogBar, CDialogBar)

//{{AFX_MSG_MAP(MCDialogBar)

ON_WM_SIZE()

//}}AFX_MSG_MAP

END_MESSAGE_MAP()
```

Please note that we do not need to specify function name when using macro ON_WM_SIZE. Instead, we must use OnSize to name the message handler of WM_SIZE.

To change a window's size, we can call function CWnd::MoveWindow(...):

```
void CWnd::MoveWindow(int x, int y, int nWidth, int nHeight, BOOL bRepaint=TRUE);
```

We need to provide new position and size in order to call this function. Because the function is a member of CWnd, we need to first obtain a pointer to the edit window then use it to call this function.

For controls contained in a dialog box, their window pointers can be obtained by calling function CWnd::GetDlgItem(...). This function requires a valid control ID:

```
CWnd *CWnd::GetDlgItem(int nID);
```

The function returns a CWnd type pointer. With this pointer, we can call any member functions of CWnd to retrieve its information or change the properties of the control.

Because we want to set the edit control's size according to the parent window's size (dialog bar), we need to find a way of retrieving a window's dimension. This can be implemented by calling another member function of CWnd:

```
void CWnd::GetClientRect(LPRECT lpRect);
```

It is very easy to use this function. We can just declare a CRect type variable, and pass its pointer to the above function when calling it. After this, the position and size of the window will be stored in the variable.

The following shows how message WM_SIZE is handled in the sample:

```
void MCDialogBar::OnSize(UINT nType, int cx, int cy)

{

CWnd *ptrWnd;

CRect rectWnd;

CDialogBar::OnSize(nType, cx, cy);

GetClientRect(rectWnd);

ptrWnd=GetDlgItem(IDC_EDIT);

if(ptrWnd != NULL)

{

ptrWnd->MoveWindow

(

rectWnd.left+15,

rectWnd.top+15,

rectWnd.Width()-30,

rectWnd.Height()-30
);

}

}
```

We can not use parameter cx and cy to resize the edit control directly because after the dialog bar gets this information, its layout may change again. The ultimate dimension of the dialog bar depends on both cx, cy and the layout algorithm. So before adjusting the size of edit control, we have to call CDialogBar::OnSize(…) first to let the dialog bar adjust its own size, then call CWnd::GetClientRect(…) to retrieve the final dimension of the dialog bar.

The rest part of this function can be easily understood: we first obtain a window pointer to the edit control and store it in pointer ptrWnd, then use it to call function CWnd::MoveWindow(…) to resize the edit control.

Dynamic Layout

Now we need to implement function MCDialogBar::CalcDynamicLayout(…). Like what we did in section 1.7, here we need to return a custom layout size when the function is called for retrieving either horizontal or vertical docking size. The following is our layout algorithm: when the bar is docked horizontally, we set its width to the horizontal size of the mainframe window’s client area, and set its height to the dialog bar’s floating vertical size; when it is docked vertically, we set its height to the vertical size of the mainframe window’s client area, and set its width to the dialog bar’s floating horizontal size.

Parameter dwMode of this function is used to tell what types of dimension is be inquired. If either LM_VERTDOCK or LM_HORZDOCK bit is set, we need to return a custom docking size. In this case, we can use another bit LM_HORZ to check if the dialog bar is docked horizontally or vertically. If this bit is set, the horizontal docking size is being inquired, otherwise the vertically docking size is being inquired.

The floating size can be obtained from a public variable: CDialogBar::m_sizeDefault. By default, this variable is initialized to the dialog template size, and is updated when the user changes the size of the dialog bar when it is floating. So this variable always represents the floating size of the dialog bar.

The following is the implementation of this function:

```
CSize MCDialogBar::CalcDynamicLayout(int nLength, DWORD dwMode)

{
```

```
CSize size;
CMainFrame *ptrWnd;
CRect rect;
ptrWnd=(CMainFrame *)(AfxGetApp()->m_pMainWnd);
ptrWnd->GetClientRect(rect);
if((dwMode & LM_VERTDOCK) || (dwMode & LM_HORZDOCK))
{
size.cx=(dwMode & LM_HORZ) ? rect.Width():m_sizeDefault.cx;
size.cy=(dwMode & LM_HORZ) ? m_sizeDefault.cy:rect.Height();
return size;
}
return CDialogBar::CalcDynamicLayout(nLength, dwMode);
}
```

First, we obtain the dimension of mainframe window's client area. For this purpose, first a window pointer to the mainframe window is obtained, then function CMainFrame::GetClientRect(...) is called to retrieve its dimension. Here the pointer to the mainframe window is obtained from a public member variable of class CWinApp. In MFC, every application has a CWinApp derived class, which contains a pointer m_pMainWnd pointing to the mainframe window. For any application, the pointer to the CWinApp object can be obtained anywhere in the program by calling function AfxGetApp(). Using this method, we can easily find the mainframe window of any MFC application.

Because the application supports status bar and tool bar, part of its client area may be covered by the control bar. So we need to deduct the covered area when calculating the dimension of the client area. For this purpose, in CMainFrame, function CWnd::GetClientRect(...) is overridden. Within the overridden function, the client area is adjusted if either the status bar or the tool bar is present:

```
void CMainFrame::GetClientRect(LPRECT lpRect)
{
CRect rect;
CFrameWnd::GetClientRect(lpRect);

if(m_wndToolBar.IsWindowVisible())

{

m_wndToolBar.GetClientRect(rect);

lpRect->bottom-=rect.Height();

}

if(m_wndStatusBar.IsWindowVisible())

{

m_wndStatusBar.GetClientRect(rect);

lpRect->bottom-=rect.Height();

}

}
```

Now back to MCDialogBar::CalcDynamicLayout(...) implementation. After obtaining the size of mainframe window's client area, we examine LM_VERTDOCK and LM_HORZDOCK bits of dwMode parameter to see if the docking size is being inquired. If so, we further examine LM_HORZ bit to see if we should return horizontally docked size or vertically docked size. We return different sizes for different cases. For all other conditions, we just return the default implementation of the base class.

Using the New Class

To use this class, first we need to declare a MCDialogBar type variable in class CMainFrame. We also need to make sure that the header file of this class is included. In the sample application, this new variable is m_wndDialogBar. Then, as we have experienced many times, we need to create the window of the dialog bar in function CMainFrame::OnCreate(...). When doing this, we need to specify CBRS_SIZE_DYNAMIC flag in order to let the dialog bar be resized dynamically. Then we can call CDialogBar::EnableDocking(...), CDialogBar::SetBarStyle(...) and CMainFrame::DockControlBar(...) to set styles and implement docking.

Now we can compile and execute the new project. Originally, the dialog bar is docked at the bottom border of the frame window. We may drag and dock it to any other border, or make it floating. As we do this, the dimension of the dialog bar will be adjusted automatically to suit different docking styles. Also, the edit control contained in the dialog bar will be resized dynamically according to the change on the dialog bar.

1.10. Adding Flyby and Tool Tip

Flyby and tool tip are two very nice features that can be added to both tool bar and dialog bar. If we enable these features, when the user moves mouse cursor over a control contained in tool bar or dialog bar and stay there for a while, a describing text about this control will appear on the status bar (It is called *Flyby*). At the same time, a small window with a short description will pop up (It is called *Tool Tip*. See Figure 1-12 for two types of controls).

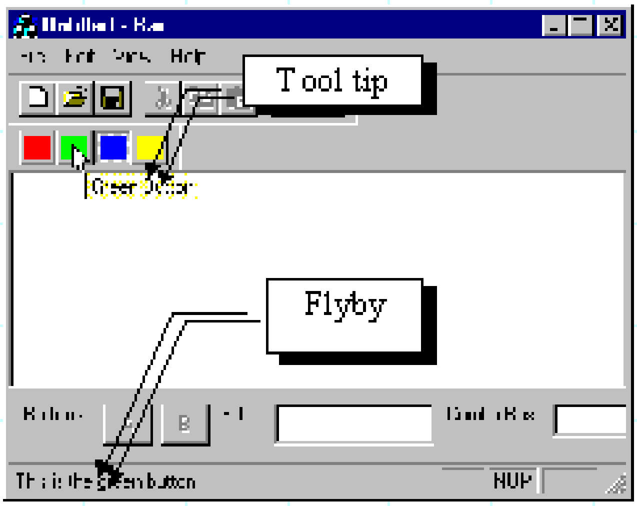
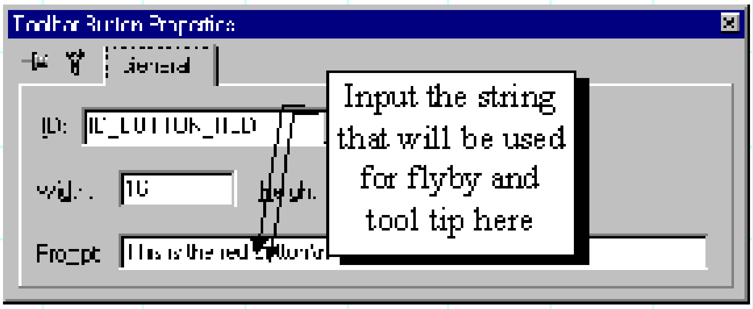


Figure 1-12. Tool tip and flyby

Both features can be enabled by calling function CControlBar::SetBarStyle(...) using the corresponding flags. To enable tool tip, we need to set CBRS_TOOLTIP flag bit, to enable flyby, we need to set CBRS_FLYBY flag bit. Actually, in the previous sections, whenever we create a tool bar or dialog bar, the two flags are always set.

Just setting the above flags can not activate the tool tip and flyby. We need to provide the text that will be used by tool tip and flyby. The text string must be implemented as application resources, and the ID of the string must be the same with the command ID of the control. For example, if we want to add flyby and tool tip for button ID_BUTTON_RED, we must create a string resource using ID_BUTTON_RED as its ID. This string will be used for both flyby and tool tip implementation. Within the string, the text is separated into two parts by an ‘\n’ character, with the sub-string before ‘\n’ used for flyby, and the sub-string after ‘\n’ used for tool tip. For example, if we want the flyby and tool tips for the red button to be "This is the red button" and "Red Button" respectively, the resource string should be "This is the red button\nRed Button". If we do not provide a string resource for this command ID, the flyby and the tool tip will not be displayed even we enable CBRS_TOOLTIP and CBRS_FLYBY flags. If string resource does not have a second sub-string (In this case, there is no '\n' character contained in the string), the whole string will be used for flyby, and no tool tip will be implemented.

Adding this string for tool bar buttons is very easy. By opening the property sheet "Toolbar Button Properties", we will find an edit box labeled "Prompt". Inputting a string into this edit box will add the string resource automatically (Figure 1-13).



Toolbar Button Properties

General

ID: ID_BUTTON_RED

Width: 16

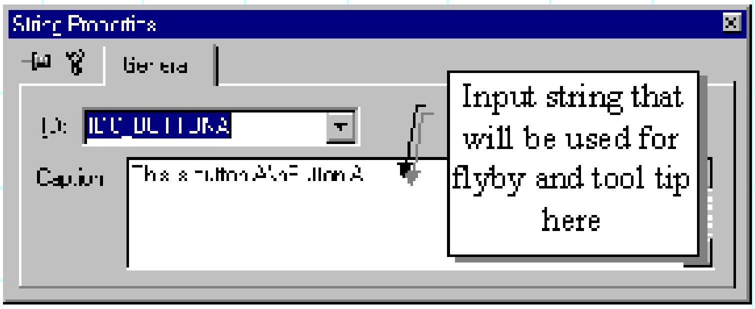
Height:

Prompt: This is the red button\n

Input the string that will be used for flyby and tool tip here

Figure 1-13. Add flyby and tool tip string for tool bar control

For dialog bar, we don't have the place to input this string in the property sheet. So we need to edit string resource directly. This can also be implemented very easily. In the Developer Studio, if we execute command **Insert | Resource…**(or press CTRL+R keys), an "Insert Resource" dialog box will pop up. To add a string resource, we need to highlight node "String Table" and press "New" button. After this, a new window with the string table will pop up. By scrolling to the bottom of the window and double clicking an empty entry, a "String Properties" property sheet will pop up, which can be used to add a new string resource (Figure 1-14).



String Properties

General

ID: IDC_BUTTON_A

Caption: This is button A\nButton A

Input string that will be used for flyby and tool tip here

Figure 1-14. Edit string resource directly to implement flyby and tool tip

Sample 1.10\Bar demonstrates how to implement flybys and tool tips. It is based on sample 1.8\Bar. Actually, the only difference between the two projects is that some new string resources are added to sample 1.10\Bar. In sample 1.10\Bar, following string resources are added:

String ID	**String Contents**

IDC_COMBO	This is combo box\nCombo Box
IDC_EDIT	This is edit box\nEdit Box
IDC_BUTTONA	This is button A\nButton A
IDC_BUTTONB	This is button B\nButton B
ID_BUTTON_RED	This is the red button\nRed Button
ID_BUTTON_GREEN	This is the green button\nGreen Button
ID_BUTTON_BLUE	This is the blue button\nBlue Button
ID_BUTTON_YELLOW	This is the yellow button\nYellow Button

After executing this sample, we can put the mouse cursor over the controls contained in the tool bar or dialog bar. By doing this, the flyby and tool tip will pop up after the cursor stays there for a short while.

1.11. Toggling Control Bars On/Off

Compared to the default tool bar (IDR_MAINFRAME) created by the Application Wizard, our custom control bar can not be turned on and off freely. Although we can float it then click the "C " button located at the top-right corner of the window to dismiss it, once we do this, there is no way to get it back. The default tool bar (also the status bar) has a much better feature: there is a corresponding command in the mainframe menu, when the tool bar is off, we could execute command **View | Toolbar** to turn it on.

When a control bar is turned off, its window actually becomes hidden instead of being destroyed. Thus when we turn it on again, the control bar can still retain its old states (Its original size, position, and docking state will not change). To turn on or off a control bar, we can call function CFrameWnd:: ShowControlBar(...), which has the following format:

```
void CFrameWnd::ShowControlBar(CControlBar* pBar, BOOL bShow, BOOL bDelay);
```

This function has three parameters: the first one is a pointer to the control bar that we want to turn on or off. The second is a Boolean type variable. If it is TRUE, the control bar will be turned on; if it is FALSE, the control bar will be turned off. The third parameter is also Boolean type, it specifies if this action should be taken immediately.

Because we need to know the current state of the control bar (on or off) to determine whether we should hide or show it, we need to call another member function of CWnd to see if the control bar is currently hidden:

```
BOOL CWnd::IsWindowVisible( );
```

This function returns a TRUE or FALSE value, from which we know the control bar's current state.

Sample 1.11\Bar supports this new feature, it is based on sample 1.10\Bar. For both tool bar and dialog bar, a new command is added to the main menu, which can be used to toggle the control bar between on and off state.

The following shows necessary steps for implementing the new commands:

1. Add two menu items **View | Color** Bar and **View | Dialog Bar** to the mainframe menu IDR_MAINFRAME, whose IDs are ID_VIEW_COLORBAR and ID_VIEW_DIALOGBAR respectively.
2. Use Class Wizard to add WM_COMMAND and UPDATE_COMMAND_UI type message handlers for the above IDs in class CMainFrame. The newly added functions are CMainFrame::OnViewColorBar(), CMainFrame:: OnViewDialogBar(), CMainFrame::OnUpdateViewColorBar(...) and CMainFrame:: OnUpdateViewDialogBar(...).
3. Implement four WM_COMMAND type message handlers. The function used to handle WM_COMMAND message for command ID_VIEW_COLORBAR is implemented as follows:

```
void CMainFrame::OnViewColorBar()

{

BOOL bShow;

bShow=m_wndColorButton.IsWindowVisible() ? FALSE:TRUE;

ShowControlBar(&m_wndColorButton, bShow, FALSE);

}
```

To indicate the status of control bars, it is desirable to check the corresponding menu item when the control bar is available, and remove the check when it becomes hidden. This is exactly the same with the behavior of the default tool bar IDR_MAINFRAME. In the sample, the menu item states are handled by trapping message UPDATE_COMMAND_UI and the check is set or removed by calling function CCmdUI::SetCheck(...). The following is the implementation of one of the above message handlers (see Chapter 2 for more about menu customization):

```
void CMainFrame::OnUpdateViewColorBar(CCmdUI* pCmdUI)

{

pCmdUI->SetCheck(m_wndColorButton.IsWindowVisible());

}
```

It is exactly the same with setting or removing check for a tool bar button.

With the above implementations, the application can be executed again. We can dismiss the control bar either by executing menu command or by clicking "X" button located at the upper-right corner of the control bar when it is floating. In both cases, the control bar can be turned on again by executing corresponding menu command. We can also dock or float the control bar and turn it off, and see if the original state will remain unchanged after it is turned on later.

Summary:

1. To add an extra tool bar, first we need to add a tool bar resource, then declare a CToolBar type variable in CMainFrame class. Within function CMainFrame::OnCreate(...), we can call the member functions of CToolBar and CMainFrame to create the tool bar window and set docking styles.
2. The dialog bar can be added in the same way, however, we need to use dialog-template resource and class CDialogBar to implement it.
3. We can trap WM_COMMAND message for executing command and trap UPDATE_COMMAND_UI for updating button state. Use ON_COMMAND and ON_UPDATE_COMMAND_UI macros to implement message mapping.
4. We can use ON_COMMAND_RANGE and ON_UPDATE_COMMAND_UI_RANGE macros to map a contiguous range of command IDs to one member function.
5. When the size of a tool bar is fixed, we can set TBBS_WRAPPED flag for a button to let the tool bar wrap after that button.
6. To customize the dynamic layout feature of tool bar and dialog bar, we need to override function CalcDynamicLayout(...).
7. To add a combo box to a tool bar, first we need to set a button to separator with specified width, then we need to create the combo box dynamically.
8. Flyby and tool tip can be activated by setting CBRS_TOOLTIP and CBRS_FLYBY flags then preparing a string resource using the exact same ID with the

control.

9. To toggle control bar on and off, we can call function CFrameWnd::ShowControlBar(). We need to use function CWnd::IsWindowVisible() to check if the control bar is currently available.

BACK TO INDEX

Chapter 2 Menu

Menu is very important for all types of applications, it provides a primary way of letting user execute application commands. If we create applications using Application Wizard, mainframe menus will be implemented automatically for all SDI and MDI applications. For dialog-based applications, system menus will also be implemented, which can be used to execute system commands (Move the application window, resize it, minimize, maximize and close the application). Some user-friendly applications also include right-click pop up menus.

This chapter discusses in-depth topics on using and customizing menus, which include: how to customize the behavior of standard menus, how to make change to standard menu interface, how to implement owner-draw menu, how to create right-click menu, how to customize the system menu and implement message mapping for system menu items.

2.1 Message WM_COMMAND and UPDATE_COMMAND_UI

When creating an SDI application using Application Wizard, we will have a default menu added to the application. This menu has four sub-menus: File, Edit, View and Help, which contain the most commonly used commands for a typical application. At the beginning some of these commands are functional (such as View | Tool bar and File | Exit) but some are disabled (such as Edit | Copy). We have to add our own code in order to make them usable.

To activate a command, we need to add message handlers for it. For a general Windows(application, we need to pay attention to two messages: WM_COMMAND and UPDATE_COMMOND_UI.

Sample 2.1\Menu demonstrates how to handle two types of messages through simulating a cut, copy and paste procedure. Here, we make use of three default menu commands added by the Application Wizard: View | Cut, View | Copy, View | Paste. The application will enable View | Paste menu item only after View | Cut or View | Copy has been executed at least once (For demonstration purpose, command View | Copy and View | Cut do not actually copy data to the clipboard). Also, item View | Paste will be changed dynamically indicating if the newly copied data has been pasted.

The sample is started by generating standard SDI application using Application Wizard. The project is named "Menu" and four standard classes are CMenuApp, CMainFrame, CMenuDoc, and CMenuView respectively. All other settings are default. After compiling and executing the skeleton application, we will see a standard SDI application with a menu implemented in the mainframe window. By examining this menu, we can find that it has the following structure:

File

New...

Open...

Save...

Save As...

Separator

Recent File

Separator

Exit

Edit

Undo

Separator

Cut

Copy

Paste

View

Toolbar

Status Bar

Help

About...

By clicking "Edit" sub-menu, we will see that all the commands contained there are disabled. If we edit the menu resource and add additional commands, they will not become functional until we add message handlers for them.

Handling WM_COMMAND Command

The first step of enabling a menu command is to implement a WM_COMMAND message handler for it. This is exactly the same with what we did for a tool bar command in Chapter 1. Just as buttons contained in a tool bar, each menu command has a unique command ID. When the user clicks the menu item, the system detects the mouse events and sends a WM_COMMAND message to the application, with the command ID passed through WPARAM parameter. In MFC, as the application receives this message, a corresponding message handler will be called to execute the command. Again, the message mapping should be implemented by using ON_COMMAND macro.

The message mapping could be implemented either manually or through using Class Wizard. This procedure is also the same with adding message handlers for tool bar buttons as we did in Chapter 1. If we use Class Wizard, after invoking it, first we need to go to "Messages Maps" page. Then we need to select a class name from "Class name" combo box (In the sample, all the commands are handled in the document class, so we need to select "CMenuDoc" if it is not selected). Next, we need to find the command ID to which we want to add handlers in "Object IDs" window, and highlight "Command" item in "Messages" window. Now click "Add Function" button and confirm the member function name. After this, a new member function and the corresponding message mapping macros will be added to the application.

There is no difference between adding message handlers manually and adding them using Class Wizard. However, doing it manually will let us understand message mapping mechanism, which will make it easier for us to further customize the menu behavior.

In the sample application we will implement commands View | Cut, View | Copy and View |Paste. So at first WM_COMMAND type message handlers need to be added for commands ID_EDIT_COPY, ID_EDIT_CUT and ID_EDIT_PASTE in CMenuDoc class. The following shows the steps of adding them through using Class Wizard:

1) In file "MenuDoc.h", three member functions are declared in the class, they

will be used to handle ID_EDIT_COPY, ID_EDIT_CUT and ID_EDIT_PASTE command execution:

```
class CMenuDoc : public CDocument

{

……

//{{AFX_MSG(CMenuDoc)

afx_msg void OnEditCopy();

afx_msg void OnEditCut();

afx_msg void OnEditPaste();

//}}AFX_MSG

……

}
```

2) In file "MenuDoc.cpp", message mapping macros are added to associate the member functions with the command IDs:

```
BEGIN_MESSAGE_MAP(CMenuDoc, CDocument)

//{{AFX_MSG_MAP(CMenuDoc)

ON_COMMAND(ID_EDIT_COPY, OnEditCopy)

ON_COMMAND(ID_EDIT_CUT, OnEditCut)

ON_COMMAND(ID_EDIT_PASTE, OnEditPaste)

//}}AFX_MSG_MAP

END_MESSAGE_MAP()
```

3) Three blank message handlers are added in file "MenuDoc.cpp":

```
void CMenuDoc::OnEditCopy()
```

```
{

}

void CMenuDoc::OnEditCut()

{

}

void CMenuDoc::OnEditPaste()

{

}
```

When first added, these functions are empty. We have to add our own code in order to support command execution.

By compiling and executing the sample application at this point, we will see that Edit | Copy, Edit | Cut and Edit | Paste menu items are all enabled. This is because three blank message handlers have just been added.

Enabling & Disabling a Command

The sample application will not actually cut, copy or paste data. The three commands will be implemented just to simulate data copy and paste procedure. Before going on to implement it, we need to make following assumptions.

Suppose the application supports only internal data copy, cut and paste (it does not accept data from other applications through using system clipboard). Before command Edit | Copy or Edit | Cut is executed, there should be no data stored in the "local clipboard". Therefore, if we execute Edit | Paste command at this time, there will be an error. To avoid this, we need to disable Edit | Paste command before data has been copied to the clipboard.

The state of menu item can be set thought handling UPDATE_COMMAND_UI message. The parameter comes along with this message is a pointer to CCmdUI type object, which can be used to enable or disable a command, set or remove check for a menu item. Handling this message for menu items is the same with that of tool bar controls.

So it is easy to find out a mechanism for updating command Edit | Paste: we need to declare a Boolean type variable in class CMenuDoc and initialize it to

FALSE. We can set this flag to TRUE when Edit | Cut or Edit | Copy command is executed, and enable Edit | Paste command only if this flag is set.

In the sample application, this Boolean variable is CMenuDoc::m_bPasteAvailable. The following code fragment shows how it is declared and initialized in the constructor of class CMenuDoc:

```
class CMenuDoc : public CDocument
{
……
protected:
BOOL m_bPasteAvailable;
……
}
CMenuDoc::CMenuDoc()
{
m_bPasteAvailable=FALSE;
}
```

The value of CmenuDoc::m_bPasteAvailable is set to TRUE when user executes either Edit | Copy or Edit | Cut command:

```
void CMenuDoc::OnEditCopy()
{
m_bPasteAvailable=TRUE;
}
void CMenuDoc::OnEditCut()
{
```

```
m_bPasteAvailable=TRUE;

}
```

Now we will use CMenuDoc::m_bPasteAvailable to enable Edit | Paste menu item when it becomes TRUE. In MFC, menu items are updated through handling UPDATE_COMMAND_UI messages. When the state of a menu command needs to be updated, UPDATE_COMMAND_UI message will be automatically sent to the application. If there exists a corresponding message handler, it will be called for updating the corresponding menu item. Otherwise, the menu item will remain unchanged.

Adding an UPDATE_COMMAND_UI message handler is the same with adding a WM_COMMAND message handler. After invoking the Class Wizard, we need to select the class name, the command ID, and highlight "UPDATE_COMMAND_UI" instead of "WM_COMMAND" in "Messages" window. Finally, we need to click button "Add Function".

After adding the message handler for ID_EDIT_PASTE command, we will have a new member function declared in class CMenuDoc, and a new message mapping macro added to the class implementation file. In addition, we will have an empty function that could be modified to implement message handling:

```
void CMenuDoc::OnUpdateEditPaste(CCmdUI *pCmdUI)

{

}
```

The only parameter to this function is a pointer to CCmdUI type object. Here, class CCmdUI has several member functions, which can be used to set the state of the menu item. To enable or disable the menu item, we can call function CCmdUI::Enable(...). The function has only one Boolean type parameter, we can pass TRUE to enable the menu item and pass FALSE to disable it. Because the state of Edit | Paste command depends upon variable CMenuDoc::m_bPasteAvailable, we can use it to set the state of menu item:

```
void CMenuDoc::OnUpdateEditPaste(CCmdUI* pCmdUI)

{

pCmdUI->Enable(m_bPasteAvailable);

}
```

By compiling and executing the sample application at this point, we will see that Edit | Paste command is disabled at the beginning, and after we execute either Edit | Cut or Edit | Copy command, it will be enabled.

Changing Menu Text

We will go on to add more features to the mainframe menu. Class CCmdUI has another useful member function CCmdUI::SetText(...), which allows us to change the text of a menu item dynamically. By using this function, we can change the text of Edit | Paste menu item so that it can convey more information to the user. For example, we could set text to "Do not paste" when data is not available, and to "Please paste" when data is available. To add this feature, all we need is to call CCmdUI::SetText(...) in the above message handler as follows:

```
void CMenuDoc::OnUpdateEditPaste(CCmdUI* pCmdUI)

{

pCmdUI->Enable(m_bPasteAvailable);

pCmdUI->SetText(m_bPasteAvailable ? "Please &paste":"Do not &paste");

}
```

Checking a Menu Item

Let's further add some more interesting features to the menu commands. With the current implementation we do not know if the data has been "Pasted" after it is "cut" or "copied" to the "clipboard". We can indicate the data status by putting a check mark on the menu item so the user knows if the current "data" in the "clipboard" has been pasted (after the user executes View | Paste command). This check will be removed when either "cut" or "copy" command is executed.

Similar to tool bar, we can call function CCmdUI::SetCheck(...) to set check for a menu item. The difference between the results of this function is the changing on the interface: while setting check for a button will make it recess, setting check for a menu item will put a check mark at the left side of the menu item.

We need a new Boolean type variable to indicate the status of "data". In the sample application, this variable is CMenuDoc::m_bDataPasted, which is initialized to FALSE in the constructor. The following functions show how its value is changed under different situations:

```
void CMenuDoc::OnEditCopy()
{
m_bPasteAvailable=TRUE;
m_bDataPasted=FALSE;
}
void CMenuDoc::OnEditCut()
{
m_bPasteAvailable=TRUE;
m_bDataPasted=FALSE;
}
void CMenuDoc::OnEditPaste()
{
m_bDataPasted=TRUE;
}
```

In function OnUpdateEditPaste(...), the menu item is checked only when flag CMenuDoc:: m_bDataPasted is TRUE:

```
void CMenuDoc::OnUpdateEditPaste(CCmdUI* pCmdUI)
{
pCmdUI->Enable(m_bPasteAvailable);
pCmdUI->SetCheck(m_bDataPasted);
pCmdUI->SetText
(
m_bPasteAvailable ?

(

m_bDataPasted ? "Data &pasted":"Please &paste"):

"Do not &paste"

);

}
```

The text of the menu item is also change to "Data pasted" when the menu item is checked.

The last thing need to be mentioned here is another member function of class CCmdUI: CCmdUI:: SetRadio(...). Like CCmdUI::SetCheck(...), this function will put a check mark on a menu item. The difference between two functions is that CCmdUI::SetRadio(...) makes menu items behave like radio buttons: when this function is called to check one item, all other items in the same group will be unchecked automatically. Calling function CCmdUI::SetCheck(...) does not affect other menu items.

2.2 Right Click Pop Up Menu

In Windows 95, right-click menu becomes a standard user interface. We can right click on the desktop, task bar, or other types of windows to bring up a menu that contains the most commonly used commands. In this section, we will discuss how to add right-click menu to our application.

Adding Menu Resource

Sample 2.2\Menu demonstrates right-click menu implementation. It is a standard SDI application generated by Application Wizard with all the default settings. This is the same with the previous sample. We can also start from sample 2.1\Menu and add the new features that will be discussed below.

Like tool bar and dialog bar, a menu can be implemented starting from building menu resource. To add a menu resource, We can execute Insert | Resource command in Developer Studio, select "menu" resource type from the popped up dialog box, and click button "New". Now a new menu resource with a default ID will be added to the application. In the sample, this default ID is changed to IDR_MENU_POPUP, and a sub-menu with four menu items is created. The newly created menu items are "Pop Up Item 1", "Pop Up Item 2", "Pop Up Item 3" and "Pop Up Item 4", whose command IDs are ID__POPUPITEM1, ID__POPUPITEM2, ID__POPUPITEM3 and ID__POPUPITEM4 respectively (Figure 2-1).

Trapping Right Button Clicking Event

The first step to implement a right-click menu is to detect mouse's right clicking event, which is a standard Windows(event, and its corresponding message is WM_RBUTTONDOWN. To trap this message, we need to implement message handler.

When we click mouse's right button on a window, message WM_RBUTTONDOWN will be sent to that window. This window could be any type: mainframe window, client window, dialog box, or even button.

We need to handle this message in the class that implements the window. For example, if we want to handle right click in a dialog box, we need to add the message handler in a CDialog derived class, if we want to handle it in the client window of an SDI application, we need to add the message handler in CView derived class.

In our sample, right-clicking menu is implemented in the client window. So we need to trap message WM_RBUTTONDOWN in class CMenuView.

Adding message handler for message WM_RBUTTONDOWN is similar to that of WM_COMMAND: first we need to declare an afx_msg type member function OnRButtonDown(...), then use ON_RBUTTONDOWN macro to map the message to this function. Finally, we need to implement the message handler. Please note that OnRButtonDow(...) is the standard function name that will be automatically associated with message WM_RBUTTONDOWN. When using ON_RBUTTONDOWN macro, we do not need to specify function name.

The above-mentioned procedure can be implemented through using Class Wizard as follows: after invoking the Class Wizard, select class CMenuView, which is the class used to implement the client window. There are a lot of virtual functions and messages listed in the "Messages" window. By scrolling the vertical scroll bar, it is easy to find message WM_RBUTTONDOWN. Now highligh this message and click "Add function" button. This will cause a new member function OnRButtonDown to be added to class CMenuView, and message mapping macros to be added to the implementation file (See Figure 2-2).

In the sample, the newly added function is CMenuView::OnRButtonDown(...), which needs to be modified to implement right-click menu. By default, this function does nothing but calling the message handler implemented by the base class:

```
void CMenuView::OnRButtonDown(UINT nFlags, CPoint point)
```

```
{
CView::OnRButtonDown(nFlags, point);
}
```

Using Class CMenu

We need to modify the above function in order to implement right-click pop up menu. In MFC, there is a class designed for menu implementation: CMenu, which contains some member functions that allow us to create menu dynamically, track and update menu items, and destroy the menu.

The first function we will use is CMenu::LoadMenu(...), it allows us to load a menu resource and use it later. This function has two different versions, one allows us to load a menu resource with a numerical ID, and the other allows us to load a resource with a string ID:

```
BOOL LoadMenu(LPCTSTR lpszResourceName);
BOOL LoadMenu(UINT nIDResource);
```

In the sample application, the menu resource is stored by a numerical ID (IDR_MENU_POPUP). We can also assign a string ID to it by inputting a quoted text in the edit box labeled with "ID".

We need to use CMenu to declare a variable that will be used to load the menu resource. Normally the right-click menu will be initiated after right-clicking event has been detected. Then the mouse's activities will be tracked by the menu until the user executes one of the menu commands or dismisses the menu. Because all these things can be handled within the message handler, the variable used to implement menu can be declared as a local variable. In the sample, the menu resource is loaded as follows:

```
void CMenuView::OnRButtonDown(UINT nFlags, CPoint point)
{
CMenu menu;
menu.LoadMenu(IDR_MENU_POPUP);
CView::OnRButtonDown(nFlags, point);
```

}

Generally, one menu contains several sub-menus, and each sub-menu contains several menu items. For right click menu, only one sub-menu (instead of whole menu) will be implemented each time the user clicks mouse's right button. Because of this, in the sample application, menu IDR_MENU_POPUP contains only one sub-menu. To obtain a pointer to the sub-menu, we can call function CMenu::GetSubMenu(...), which has the following format:

CMenu *Cmenu::GetSubMenu(int nPos) const;

Parameter nPos indicates which sub-menu we are trying to obtain. In a menu resource, the left-most sub-menu is indexed 0, next sub-menu indexed 1, and so on. In the sample application, sub-menu that contains items "Pop Up Item 1"... is located at position 0.

This function returns a CMenu type pointer that could be used to further access each item contained in the sub-menu. Before the menu is displayed, we may want to set the state of each menu item: we can enable, disable, set check or change text for a menu item. Please note that for a right-click menu, we do not need to handle message UPDATE_COMMAND_UI in order to set the states of menu items. Instead, there exist two member functions that can be used:

UINT CMenu::EnableMenuItem(UINT nIDEnableItem, UINT nEnable);

UINT CMenu::CheckMenuItem(UINT nIDCheckItem, UINT nCheck);

The above two functions can be used to enable/disable, set/remove check for a menu item. When calling the two functions, we can reference a menu item by using either its command ID or its position. Normally we can pass a command ID to nIDEnableItem or nIDCheckItem parameter. If we want to reference an item by its position (0 based, for example, in the sample application, ID__POPUPITEM1's position is 0, and ID__POPUPITEM2's position is 1...), we need to set MF_BYPOSITION bit of nEnable or nCheck parameter.

The menu can be activated and tracked by calling function CMenu::TrackPopupMenu(...):

BOOL CMenu::TrackPopupMenu

(

UINT nFlags, int x, int y, CWnd* pWnd, LPCRECT lpRect=NULL

```
);
```

This function has 5 parameters. The first parameter nFlags lets us set styles of the menu (Where should the menu be put, which mouse button will be tracked). The most commonly used combination is TPM_LEFTALIGN | TPM_RIGHTBUTTON, which aligns menu's left border according to parameter x, and tracks mouse's right button activity (because we are implementing a right-click menu). The second parameter y decides the vertical position of the menu's top border. Please note that when message WM_RBUTTONDOWN is received, position of current mouse cursor will be passed to one of the parameters of function OnRButtonDown(...) as a CPoint type object. To make right-click menu easy to use, we can pass this position to function CMenu::TrackPopupMenu(...), which will create a pop up menu at the position of current mouse cursor. The fourth parameter is a CWnd type pointer, which indicates which window owns the pop up menu. In the sample, because the menu is implemented in the member function of class CMenuView, we can use this pointer to indicate the menu owner. The final parameter discribes a rectangle within which the user can click the mouse without dismissing the pop up menu. We could set it to NULL, in which case the menu will be dismissed if the user clicks outside the pop up menu.

Implementing Right-Click Menu

Now we can implement WM_RBUTTONDOWN message handler, load the menu resource and create right-click menu in the member function:

```
void CMenuView::OnRButtonDown(UINT nFlags, CPoint point)

{

CMenu menu;

CMenu *ptrMenu;

menu.LoadMenu(IDR_MENU_POPUP);

ptrMenu=menu.GetSubMenu(0);

ptrMenu->EnableMenuItem(ID__POPUPITEM1, MF_GRAYED);

ptrMenu->EnableMenuItem(ID__POPUPITEM2, MF_ENABLED);

ptrMenu->CheckMenuItem(ID__POPUPITEM3, MF_UNCHECKED);
ptrMenu->CheckMenuItem(ID__POPUPITEM4, MF_CHECKED);
ClientToScreen(&point);
ptrMenu->TrackPopupMenu
(
TPM_LEFTALIGN|TPM_RIGHTBUTTON,
point.x,
point.y,
this,
NULL
);
CView::OnRButtonDown(nFlags, point);
}
```

After implementing the right-click menu, we still need to call function CView::OnRButtonDown(...). This is to make sure that the application does not lose any default property implemented by class CView.

In the above function, before CMenu::TrackPopupMenu(...) is called, function CWnd::ClientToScreen() is used to convert the coordinates of a point from the client window to the desktop window (the whole screen). When point parameter is passed to CMenuView::OnRButtonDown(...), it is assumed to be measured in the coordinates system of the client window, which means (0, 0) is located at the upper-left corner of the client window. When we implement a menu, function CMenu::TrackPopupMenu(...) requires coordinates to be measured in the desktop window system, which means (0, 0) is located at the upper-left corner of the screen. Function CWnd::ClientToScreen(...) can convert the coordinates of a point between the two systems. This function is frequently used when we need to convert coordinates from one window to another.

By compiling and executing the application at this point, we will see that the right-click menu is implemented successfully.

Message Mapping for Right-Click Menu

Although the right-click menu is working now, we still can not use it to execute any command. The reason is simple: we haven't implemented WM_COMMAND type message handlers for the menu commands yet. For right-click menu, we cannot add message handlers using Class Wizard, because the command IDs of the menu items are not listed in "Object IDs" window of the Class Wizard. Thus, we have to do everything manually. Actually, adding message handlers for right-click menu items is the same with adding handlers for a normal menu item: we need to declare afx_msg type functions, use ON_COMMAND macros to do the message mapping, and implement the member functions. In the sample, WM_COMMAND type message handler is added for each menu item contained in the right-click menu. Within each message handler, a message box pops up indicating which menu item it is.

The following portion of code shows the member functions declared in class CMenuDoc:

```
class CMenuDoc : public CDocument

{

……

//{{AFX_MSG(CMenuDoc)

//}}AFX_MSG

afx_msg void OnPopUpItem1();

afx_msg void OnPopUpItem2();

afx_msg void OnPopUpItem3();

afx_msg void OnPopUpItem4();

DECLARE_MESSAGE_MAP()

……

}
```

Message mapping macros are implemented as follows:

```
BEGIN_MESSAGE_MAP(CMenuDoc, CDocument)
```

```
//{{AFX_MSG_MAP(CMenuDoc)
//}}AFX_MSG_MAP
ON_COMMAND(ID__POPUPITEM1, OnPopUpItem1)
ON_COMMAND(ID__POPUPITEM2, OnPopUpItem2)
ON_COMMAND(ID__POPUPITEM3, OnPopUpItem3)
ON_COMMAND(ID__POPUPITEM4, OnPopUpItem4)
END_MESSAGE_MAP()
```

Four member functions are implemented as follows:

```
void CMenuDoc::OnPopUpItem1()
{
AfxMessageBox("Pop up menu item 1");
}
void CMenuDoc::OnPopUpItem2()
{
AfxMessageBox("Pop up menu item 2");
}
void CMenuDoc::OnPopUpItem3()
{
AfxMessageBox("Pop up menu item 3");
}
void CMenuDoc::OnPopUpItem4()
{
```

```
AfxMessageBox("Pop up menu item 4");

}
```

With the above implementation, we are able to execute the commands contained in the right-click pop up menu.

2.3 Updating Menu Dynamically

Sometimes it is desirable to change the contents of a menu dynamically. For example, if we create an application that supports many commands, we may want to organize them into different groups. Sometimes we want to enable a group of commands, sometimes we want to disable them.

Although we can handle UPDATE_COMMAND_UI message to enable or disable commands, sometimes it is more desirable if we can remove the whole sub-menu instead of just graying the menu text. Actually, sub-menu and menu item can all be modified dynamically: we can either add or delete a sub-menu or menu item at any time; we can also change the text of a menu item, move a sub-menu or menu item, or add a separator between two menu items. All these things can be implemented at run-time.

Menu Struture

The structure of menu --> sub menu --> menu item is like a tree. At the topmost level (the root), the menu comprises several sub-menus. Each sub-menu also comprises several items, which could be a normal command or another sub-menu. For example, in application Explorer (file browser in Windows95(), its first level menu comprises five sub-menus: File, Edit, View, Tool, and Help. If we examine File sub-menu, we will see that it comprises eight items: New, separator, Create Shortcut, Delete, Rename, Properties, separator and Close. Here, item New is another sub-menu, which comprises several other menu items. This kind of structure can continue. As long as our program needs, we can organize our menu into many different levels.

In MFC, class CMenu should be used this way. With a CMenu type pointer to a menu object, we have the access to only the menu items at certain level. If we want to access a menu item at a lower level, we first need to access the sub-menu that contains the desired menu item.

This can be explained by the previous "Explorer" example: suppose we have a CMenu type pointer to the main menu, we can use it to access the first level menu items: File, Edit, View, Tool, and Help. This means we can use the pointer to disable, enable or set text for any of the above items, but we can not use it to make change to the items belonging to other levels, for example, New item under File sub-menu. To access this item, we need to first obtain a CMenu type pointer to File sub-menu, then use it to modify item File | New.

Inserting and Removing Menu Item

Class CMenu has certain member functions that allow us to insert or delete a menu item dynamically. We can add either a menu item (including separator), or a whole sub-menu. When we remove a sub-menu, all the lower level items and sub-menus will be removed.

The function that can be used to insert menu items or sub-menus is CMenu::InsertMenu(…), it has the following format:

```
BOOL CMenu::InsertMenu

(

UINT nPosition, UINT nFlags, UINT nIDNewItem=0, LPCTSTR lpszNewItem=NULL

);
```

This function has five parameters. The first parameter, nPosition, indicates where we want our new menu item to be inserted. It could be an absolute position, 0, 1, 2…, or a command ID of the menu item. In the former case, MF_BYPOSITION bit of second parameter nFlags must be set. In the latter case, MF_BYCOMMAND bit must be set. Since not all menu items have a command ID (such as a separator), using position to indicate a menu item is sometimes necessary.

Generally, we can insert three types of items: a menu item with specified command ID, a separator or a sub-menu. To insert a menu item, we need to pass the command ID to nIDNewItem parameter, then use the final parameter lpszNewItem to specify the text of this menu item. If we want to insert a separator, we must set MF_SEPARATOR bit of parameter nFlag. In this case the rest two parameters nIDNewItem and lpszNewItem will be ignored, so we can pass any value to them. If we want to insert a sub-menu, we must pass a menu handle to parameter nIDNewItem, and use lpszNewItem to set the text for the menu item.

In Windows(programming, handle is a very important concept. Many types of resources are managed through using handles. A handle is just a unique number that can be used to reference a block of memory. After an object (program, resource, dynamically allocated memory block, etc.) is loaded into the memory, it will be assigned a unique handle that can be used to access this object. As a programmer, we don't need to know the exact value of a handle. When accessing an object, instead of using handle's absolute value, we can just use the variable that stores the handle.

Different handles have different prototypes, for a menu object, its prototype is HMENU.

In MFC, this is further simplified. When we call a member function to load an object into the memory, the handle will be automatically saved to a member variable. Later if we need this handle, we can just call a member function to retrieve it.

In the case of class CMenu, after calling function CMenu::LoadMenu(...), we can obtain the handle of the menu resource by calling function CMenu::GetSafeHmenu().

For example, in sample 2.2\Menu, after menu resource IDR_MENU_POUP is loaded into the memory, we could obtain the handle of its first sub-menu and store it to an HMENU type variable as follows:

```
CMenu menu;

CMenu *ptrMenu;

HMENU hMenu;

menu.LoadMenu(IDR_MENU_POPUP);

ptrMenu=menu.GetSubMenu(0);

hMenu=ptrMenu->GetSafeHmenu();
```

To remove a menu item, we need to use another member function of CMenu:

```
BOOL CMenu::RemoveMenu(UINT nPosition, UINT nFlags);
```

The meanings of nPosition and nFlags parameters are similar to those of function CMenu:: InsertMenu(...).

There is another similar function: CMenu::DeleteMenu(...), which can also remove a menu item or sub- menu. However, if we use this function to delete a sub-menu, the menu resource will be released from the memory. In this case, if we wand to use the sub-menu again, we need to reload the menu resource.

Sample Implementation

Sample 2.3\Menu demonstrates how to add and delete menu items dynamically. It is a standard SDI application generated by Application Wizard, with all the default settings. In this sample, there are two commands Edit | Insert Dynamic Menu and Edit | Delete Dynamic Menu. If we execute the first command, a new sub-menu will be added between File and Edit sub-menus. We can use the second command to remove this dynamically added sub-menu.

The first step is to add two menu items to IDR_MAINFRAME menu resource. In the sample, two commands are added to Edit sub-menu, their description text are "Insert Dynamic Menu" and "Delete Dynamic Menu" respectively, and their command IDs are ID_EDIT_INSERTDYNAMICMENU and ID_EDIT_DELETEDYNAMICMENU. Both of them have WM_COMMAND and UPDATE_COMMAND_UI message handlers in class CMenuDoc, whose function names are OnEditInsertDynamicMenu, OnUpdateEditInsertDynamicMenu, OnEditDeleteDynamicMenu and OnUpdateEditDeleteDynamicMenu.

Because we want to disable command ID_EDIT_DELETEDYNAMICMENU and enable command ID_EDIT_INSERTDYNAMICMENU before the sub-menu is inserted, and reverse this after the menu is inserted, another Boolean type variable m_bSubMenuOn is declared in class CMenuDoc, which will be used to indicate the state of the inserted menu. It is initialized to FALSE in the constructor.

Preparing the menu resource that will be used to implement dynamic sub-menu is the same with what we did in the previous sample. Here a resource IDR_MENU_POPUP is added to the application, whose content is the same with the resource created in sample 2.2\Menu.

In this case, we could not use a local variable to load the menu, because once the menu is inserted, it may exist for a while before the user removes it. If we still use a local variable, it will go out of scope after the messagae hander returns. In the sample, a CMenu type variable is declared in class CMenuDoc, which is used to load the menu resource in the constructor.

The following shows the modified class CMenuDoc:

```
class CMenuDoc : public CDocument

{

protected: // create from serialization only
CMenu m_menuSub;

BOOL m_bSubMenuOn;

......

protected:

//{{AFX_MSG(CMenuDoc)

afx_msg void OnEditInsertDynamicMenu();

afx_msg void OnUpdateEditInsertDynamicMenu(CCmdUI* pCmdUI);

afx_msg void OnEditDeleteDynamicMenu();

afx_msg void OnUpdateEditDeleteDynamicMenu(CCmdUI* pCmdUI);

//}}AFX_MSG

DECLARE_MESSAGE_MAP()

};
```

The following is the constructor within which the menu resource is loaded and m_bSubMenuOn is initialized:

```
CMenuDoc::CMenuDoc()

{

m_menuSub.LoadMenu(IDR_MENU_POPUP);

m_bSubMenuOn=FALSE;

}
```

The following shows two UPDATE_COMMAND_UI message handlers where two menu commands are enabled or disabled:

```
void CMenuDoc::OnUpdateEditInsertDynamicMenu(CCmdUI* pCmdUI)

{
```

```
pCmdUI->Enable(m_bSubMenuOn == FALSE);

}

void CMenuDoc::OnUpdateEditDeleteDynamicMenu(CCmdUI* pCmdUI)

{

pCmdUI->Enable(m_bSubMenuOn == TRUE);

}
```

At last, we must implement two WM_COMMAND message handlers. First, we need to find a way of accessing mainframe menu IDR_MAINFRAME of the application. In MFC, a menu associated with a window can be accessed by calling function CWnd::GetMenu(), which will return a CMenu type pointer. Once we get this pointer, we can use it to access any of its sub-menus.

The mainframe window pointer can be obtained by calling function AfxGetMainWnd() anywhere in the program. An alternate way is to call AfxGetApp() to obtain a CWinApp type pointer, then access its public member m_pMainWnd. We could use CMenu type pointer to insert or remove a sub-menu dynamically.

The following shows two message handlers that are used to insert or remove the sub-menu:

```
void CMenuDoc::OnEditInsertDynamicMenu()

{

CMenu *pTopMenu=AfxGetMainWnd()->GetMenu();

CMenu *ptrMenu=m_menuSub.GetSubMenu(0);

pTopMenu->InsertMenu

(

1, MF_BYPOSITION | MF_POPUP, (UINT)ptrMenu->GetSafeHmenu(), "&Dynamic Menu"

);
AfxGetMainWnd()->DrawMenuBar();

m_bSubMenuOn=TRUE;

}

void CMenuDoc::OnEditDeleteDynamicMenu()

{

CMenu *pTopMenu=AfxGetMainWnd()->GetMenu();

pTopMenu->RemoveMenu(1, MF_BYPOSITION);

AfxGetMainWnd()->DrawMenuBar();

m_bSubMenuOn=FALSE;

}
```

When inserting sub-menu, flag MF_BYPOSITION is used. This is because the first level menu items do not have command IDs.

After the menu is inserted or removed, we must call function CWnd::DrawMenuBar() to let the menu be updated. Otherwise although the content of the menu is actually changed, it will not be reflected to the user interface until the update is triggered by some other reasons.

2.4 Bitmap Check

The default menu check provided by MFC is a tick mark, and nothing is displayed when the check is removed. With a little effort, we can prepare our own bitmaps and use them to implement the checked and unchecked state (Figure 2-3).

To implement the checked and unchecked states of menu items using bitmaps, we need to call the following member function of CMenu:

```
BOOL CMenu::SetMenuItemBitmaps

(

UINT nPosition, UINT nFlags, const CBitmap* pBmpUnchecked,

const CBitmap* pBmpChecked
);
```

The first two parameters of this function indicate which menu item we are working with. Their meanings are the same with that of functions such as CMenu::EnableMenuItem(...). When calling this function, we can use either a command ID or an absolute position to identify a menu item. The third and fourth parameters are pointers to bitmaps (CBitmap type objects), one for checked state, one for unchecked state.

Like menu, bitmap can also be prepared as resource then be loaded at program's runtime. We can edit a bitmap in Developer Studio, and save it as application's resource. Adding a bitmap resource is the same with adding other types of resources: we can execute Insert | Resource... command, then select Bitmap from the popped up dialog box. The newly added resource will be assigned a default ID, it could also be changed by the programmer.

To load a bitmap resource into the memory, we need to use class CBitmap. This procedure is similar to loading a menu resource: first we need to use CBitmap to declare a variable, then call function CBitmap::LoadBitmap(...) to load the resource. For example, if we have a CBitmap type variable bmp, and our bitmap resource's ID is IDB_BITMAP, we can load the bitmap as follows:

```
bmp.LoadBitmap(IDB_BITMAP);
```

When calling function CMenu::SetMenuItemBitmaps(...), we can pass the pointers of CBitmap type variables to its parameters.

Sample 2.4\Menu demonstrates bitmap check implementation. It is based on sample 2.3\Menu, which adds check bitmaps to menu item ID__POPUPITEM1 and ID__POPUPITEM2. Two bitmap resources IDB_BITMAP_CHECK and IDB_BITMAP_UNCHECK are used to indicate menu item's checked and unchecked states respectively. Both bitmaps have a size of 15(15, which is a suitable size for normal menu items. If we use bigger bitmaps, they might be chopped to fit into the area of menu item.

In the sample, two new CBitmap type variables m_bmpCheck and m_bmpUnCheck are declared in class CMenuDoc, which are used to load the bitmap resources:

```
class CMenuDoc : public CDocument

{

protected:
```

```
CMenu m_menuSub;

CBitmap m_bmpCheck;

CBitmap m_bmpUnCheck;

BOOL m_bSubMenuOn;

……

}
```

In the constructor of CMenuDoc, bitmap resources IDB_BITMAP_CHECK and IDB_BITMAP_UNCHECK are loaded using two variables. Also, after we load the pop up menu resource, function CMenu:: SetMenuItemBitmap(...) is called for both ID__POPUPITEM1 and ID__POPITEM2. We use bitmaps to indicate both checked and unchecked states. The following code fragment shows how it is implemented:

```
CMenuDoc::CMenuDoc()

{

CMenu *ptrMenu;

m_menuSub.LoadMenu(IDR_MENU_POPUP);

m_bmpCheck.LoadBitmap(IDB_BITMAP_CHECK);

m_bmpUnCheck.LoadBitmap(IDB_BITMAP_UNCHECK);

ptrMenu=m_menuSub.GetSubMenu(0);

ptrMenu->SetMenuItemBitmaps(0, MF_BYPOSITION, &m_bmpUnCheck,
&m_bmpCheck);

ptrMenu->SetMenuItemBitmaps(1, MF_BYPOSITION, &m_bmpUnCheck,
&m_bmpCheck);

m_bSubMenuOn=FALSE;

}
```

When calling function CMenu::SetMenuItemBitmap(...), we use absolute position instead of command ID to identify a menu item. So the second parameter passed to the function is MF_BYPOSITION.

Besides these, two UPDATE_COMMAND_UI message handlers are also added to CMenuDoc to set item ID__POPUPITEM1 to checked state and set ID_POPUPITEM2 to unchecked state permenently. The following two functions show the implementation of the messages handlers:

```
void CMenuDoc::OnUpdatePopUpItem1(CCmdUI *pCmdUI)

{

if(m_bSubMenuOn)pCmdUI->SetCheck(TRUE);

}

void CMenuDoc::OnUpdatePopUpItem2(CCmdUI *pCmdUI)

{

if(m_bSubMenuOn)pCmdUI->SetCheck(FALSE);

}
```

In the sample, bitmaps are prepared for both checked and unchecked states for a menu item. When calling function CMenu::SetMenuItemBitmaps(...), if either of the bitmaps is not provided (the corresponding parameter is NULL), nothing will be displayed for that state. If both parameters are NULL, the default tick mark will be used for the checked state.

2.5 System Menu and Bitmap Menu Item

System Menu

By default, every application has a system menu, which is accessible through left clicking on the small icon located at the left side of application's caption bar, or right clicking on the application when it is in icon state. The system menu can be customized to meet special requirement. Especially, we can add and delete menu items dynamically just like a normal menu.

We already know how to access an application's standard menu. Once we obtained a CMenu type pointer to the application's standard menu, we can feel free to add new menu items, remove menu items, and change their attributes dynamically.

System menu is different from a standard menu. We need to call another function to obtain a pointer to it. In MFC, the function that can be used to access system menu is CWnd::GetSystemMenu(...). Please note that we must call this function for a window that has an attached system menu. For an SDI or MDI application, system menu is attached to the mainframe window. For a dialog box based application, the system menu is attached to the dialog window.

Unlike user implemented commands, system commands (commands on the system menu) are sent through WM_SYSCOMMAND rather than WM_COMMAND message. If we implement message handlers to receive system commands, we need to use ON_WM_SYSCOMMAND macro.

Bitmap Menu Item

From the samples in the previous sections, we already know how to customize a menu item: we can set check, remove check, change text dynamically. We can also use bitmaps to represent its checked and unchecked states. Besides above features, a menu item can be modified to display a bitmap instead of a text string. This can make the application more attractive, because sometimes images are more intuitive than text strings.

Sample 2.5\Menu demonstrates the two techniques described above, it is based on sample 2.4\Menu. In this sample, one of the system menu items (a separator) is changed to bitmap menu item. If we execute this "bitmap command", a message box will pop up. Also, another new command is added to the system menu, it allows the user to resume the original system menu.

New Functions

Function CWnd::GetSystemMenu(...) has only one Boolean type parameter:

```
CMenu *CWnd::GetSystemMenu(BOOL bRevert);
```

Although we can call this function to obtain a pointer to the system menu and manipulate it, the original default system menu can be reverted at any time by calling this function and passing a TRUE value to its bRevert parameter. In this case, function's returned value has no meaning and should not be treated as a pointer to a menu object. We need to pass FALSE to this parameter in order to obtain a valid pointer to the system menu.

Function CMenu::ModifyMenu(...) allows us to change any menu item to a separator, a sub-menu, a bitmap menu item. It can also be used to modify a menu item's text. This member function has two versions:

```
BOOL CMenu::ModifyMenu
(
UINT nPosition, UINT nFlags, UINT nIDNewItem=0, LPCTSTR lpszNewItem = NULL
);
BOOL CMenu::ModifyMenu
(
UINT nPosition, UINT nFlags, UINT nIDNewItem, const CBitmap* pBmp
);
```

The first version of this function allows us to change a menu item to a text item, a separator, or a sub-menu. The second version allows us to change a menu item to a bitmap item. For the second version, parameter nIDNewItem specifies the new command ID, and parameter pBmp is a pointer to a CBitmap object, which must contain a valid bitmap resource.

Menu Modification

In the sample application, a bitmap resource ID_BITMAP_QUESTION is prepared for implementing bitmap menu item. This bitmap contains a question mark. There is no restriction on the bitmap size, because the size of menu item will be adjusted automatically to let the image fit in.

To load the image, a new CBitmap type variable m_bmpQuestion is declared in class CMainFrame, and bitmap resource ID_BITMAP_QUESTION is loaded in the constructor of class CMainFrame:

```
class CMainFrame : public CFrameWnd
{
……
protected:
CStatusBar m_wndStatusBar;
CToolBar m_wndToolBar;

CBitmap m_bmpQuestion;

……

};

CMainFrame::CMainFrame()

{

m_bmpQuestion.LoadBitmap(IDB_BITMAP_QUESTION);

}
```

The pointer to the system menu is obtained in function CMainFrame::OnCreate(...). First, system menu's fifth item (a separator) is modified to a bitmap menu item, then another new command "Resume standard system menu" is inserted before this bitmap menu item. The IDs of the two commands are ID_QUESTION and ID_RESUME respectively.

Although we can use any numerical values as the command IDs of the newly added menu items, they should not be used by other resources of the application. The best way to prevent this from happening is to generate two new string resources in the application, and use ID_QUESTION and ID_RESUME as their symbolic IDs. Because Developer Studio will always allocate unused values for new resources, we can avoid sharing IDs with other resources by using this method.

The following shows how we access the system menu and make changes to its items:

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)

{

CMenu *ptrMenu;

……

ptrMenu=GetSystemMenu(FALSE);
ptrMenu->ModifyMenu(5, MF_BYPOSITION, ID_QUESTION, &m_bmpQuestion);

ptrMenu->InsertMenu(5, MF_BYPOSITION, ID_RESUME, "Resume standard system menu");

return 0;

}
```

Message Mapping for System Command

In order to trap events for the system menu, we need to handle message WM_SYSCOMMAND. We can map this message to our member function by using macro ON_WM_SYSCOMMAND. The format of the message handler is:

```
afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
```

And the format of mapping macro is:

```
ON_WM_SYSCOMMAND()
```

Of course, we can ask Class Wizard to do the mapping for us. Before using it to add the above message handler to CMainFrame class, we need to make following changes to the settings of Class Wizard: first click "Class info" tab of the Class Wizard, then select "Window" from the window "Message filter" (Figure 2-4). The default message filter for CMainFrame frame window is "Topmost frame", and WM_SYSCOMMAND will not be listed in the message list. After this modification, we can go back to "Message Maps" page, and choose "WM_SYSCOMMAND" from messages window. To add the message handler, we simply need to click "Add function" button (make sure the settings in other windows are correct). After this, the new function OnSysCommand(...) will be added to the application.

Here is how this function is declared in class CMainFrame:

```
class CMainFrame

{

......

afx_msg void OnSysCommand(UINT nID, LPARAM lParam);

......
};
```

The message mapping is as follows:

```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)

......

ON_WM_SYSCOMMAND()

//}}AFX_MSG_MAP

END_MESSAGE_MAP()
```

Message handler CMainFrame::OnSysCommand(...) has two parameters, first of which is the command ID, and the second is LPARAM message parameter. In our case, we only need to use the first parameter, because it tells us which command is being executed. If the command is ID_RESUME, we need to call function CMenu::GetSystemMenu(...) to resume the default system menu; if the command is ID_QUESTION, we pop up a message box:

```
void CMainFrame::OnSysCommand(UINT nID, LPARAM lParam)

{

if(nID == ID_RESUME)GetSystemMenu(TRUE);

if(nID == ID_QUESTION)AfxMessageBox("Question");

CFrameWnd::OnSysCommand(nID, lParam);

}
```

2.6 Owner-Draw Menu

When we highlight a bitmap menu item by selecting it using mouse, the bitmap will be inversed. We have no way of modifying this property because function CMenu::ModifyMenu(...) requires only one bitmap, which will be used to implement menu item's normal state. The other states of a bitmap menu item will be drawn using the default implementations. Normally this is good enough. However, sometimes we may want to use different bitmaps to represent a menu item's different states: selected, unselected, checked, unchecked, grayed or enabled.

Also, sometimes we may want to paint a menu item dynamically. Suppose we need to create a "Color" menu item: the menu item represents a color that the user can use, and this color can be modified to represent any available color in the system. To implement this type of menu item, we can paint it using the currently selected color. In this case it is not appropriate to create the menu item using bitmap resources: there may exist thousands of available colors in the system, and it is just too inconvenient to prepare a bitmap resource for each possible color.

The owner-draw menu can help us build more powerful user interface. With this type of menu, we can draw the menu items dynamically, using different bitmaps to represent different states of the menu item. We can also change the associated bitmaps at any time.

Overriding Two Functions

The implementation of owner draw menu is not very difficult, all we need is to override two member functions of class CMenu: CMenu::MeasureItem(...) and CMenu::DrawItem(...).

By default, the menu is drawn by the system. We can change this attribute by specifying MF_OWNERDRAW style for a menu. Any menu with this style will be drawn by its owner. Since a menu's owner is usually the mainframe window (in SDI and MDI applications), we can add code to class CMainFrame to implement dynamic menu drawing. Actually, the menu drawing has two associated messages: WM_MEASUREITEM and WM_DRAWITEM. When a menu item with MF_OWNERDRAW style needs to be drawn, the menu sends out the above two messages to the mainframe window. The mainframe window finds out the pointer to the corresponding menu and calls functions CMenu::MeasureItem(...) and CMenu::DrawItem(...). Here, the first function is used to retrieve the dimension of the menu item, which can be used to calculate its layout. The second function implements menu drawing. We need to override it in order to implement custom interface.

One simple and most commonly used way to provide graphic interface is to prepare a bitmap resource then load and draw it at run time. Please note that this is different from preparing a bitmap resource and calling CMenu::ModifyMenu(...) to associate the bitmap with a menu item. If we implement drawing by ourselves, we can manipulate drawing details. For example, when drawing the bitmap, we can change the size of the image, add a text over it. If we assign the bitmap to the menu item, we lose the control over the details of painting the bitmap.

Drawing a Bitmap

To draw a bitmap, we need to understand some basics on graphic device interface (GDI). This topic will be thoroughly discussed from chapter 8 through 12, here is just a simple discussion on bitmap drawing. In Windows(operating system, when we want to output objects (a pixel, a line or a bitmap) to the screen, we can not write directly to the screen. Instead, we must write to the device context (DC). A device context is a data structure containing information about the drawing attributes of a device (typically a display or a printer). We can use DC to write text, draw pixels, lines and bitmaps to the devices. In MFC the device context is supported by class CDC, which has many functions that can let us draw different types of objects.

We can implement bitmap drawing by obtaining a target DC and calling member functions of CDC. The target DC is usually obtained from a window.

A DC can select a lot of GDI objects, such pen, brush, font and bitmap. Pen can be different type of pens, and brush can be different types of brushes. A DC can select any pen or brush as its current tool, but at any time, a DC can select only one pen and one brush. If we want to draw a pixel or a line, we can select the appropriate pen into the target DC and use it to implement drawing. A DC can also select bitmap for drawing. However, to paint a bitmap, we can not select it into the target DC and draw it directly. The normal way of painting a bitmap is to prepare a compatible memory DC (which is a block of memory with the same attributes of the target DC), select the bitmap into the memory DC, and copy the bitmap from the memory DC to the target DC.

We will learn more about DC and bitmap drawing in later chapters. For the time being, we can neglect the drawing details.

Deriving a New Class from CMenu

Sample 2.6\Menu demonstrates owner-draw menu implementation. It is base on sample 2.5\Menu. In this sample menu items ID__POPUPITEM1 and ID__POPUPITEM2 of the dynamic menu are implemented as owner-draw menu, their menu items are painted using different bitmaps.

Like sample 2.5\Menu, some new images are prepared as bitmap resources. In the sample, the newly added bitmaps are IDB_BITMAP_QUESTIONSEL, IDB_BITMAP_SMILE and IDB_BITMAP_SMILESEL. We will use IDB_BITMAP_SMILE and IDB_BITMAP_SMILESEL to implement owner-draw menu item ID__POPITEM1, and use IDB_BITMAP_QUESTION to implement ID__POPITEM2.

First, we need to override class CMenu. In the sample, a new class MCMenu is derived from CMenu, in this class, we declare four CBitmap type variables that will be used to load bitmaps for menu drawing. Also, functions

CMenu::MeasureItem(...) and CMenu::DrawItem(...) are overridden:

```
class MCMenu : public CMenu
{
protected:
CBitmap m_bmpQuestion;
CBitmap m_bmpQuestionSel;
CBitmap m_bmpSmile;
CBitmap m_bmpSmileSel;
public:
MCMenu();
virtual ~MCMenu();
virtual void MeasureItem(LPMEASUREITEMSTRUCT);
virtual void DrawItem(LPDRAWITEMSTRUCT);
};
```

In the constructor of class MCMenu, four bitmaps are loaded:

```
MCMenu::MCMenu() : CMenu()
{
m_bmpQuestion.LoadBitmap(IDB_BITMAP_QUESTION);
m_bmpQuestionSel.LoadBitmap(IDB_BITMAP_QUESTIONSEL);
m_bmpSmile.LoadBitmap(IDB_BITMAP_SMILE);
m_bmpSmileSel.LoadBitmap(IDB_BITMAP_SMILESEL);
}
```

Overriding Function CMenu::MeasureItem(...)

Next we need to override function CMenu::MeasureItem(...). It has only one parameter, which is a pointer to MEASUREITEMSTRUCT type object. Structure MEASUREITEMSTRUCT is used to inform system the dimension of the owner-draw menu and other controls. There are three important members that will be used: itemWidth and itemHeight, which represent width and height of the menu item; itemData, from which we know the type of the menu item that is being inquired.

The program can assign special data to an owner-draw menu item when it is being created. When the mainframe window calls the overridden functions to inquire the item's attributes or paint the menu, the data will be passed through itemData member of structure MEASUREITEMSTRUCT or DRAWITEMSTRUCT (DRAWITEMSTRUCT will be passed to function CMenu::DrawItem(...)). By examining this member in the overriden functions, we know what type of menu item we are working with.

Class CBitmap has a member function that can be used to obtain the size of a bitmap: CBitmap::GetBitmap(...). We need to prepare a BITMAP type object and pass its pointer to this function. After calling this function, the structure will be filled with a lot of information about the bitmap (not only the image dimension). In this sample, we need to use only two members of BITMAP structure: bmWidth and bmHeight, which represent a bitmap's width and height.

The following is the overridden function of CMenu::MeasureItem(...):

```
void MCMenu::MeasureItem(LPMEASUREITEMSTRUCT lpMeasureItemStruct)

{

BITMAP bm;

switch(lpMeasureItemStruct->itemData)

{

case MENUTYPE_SMILE:

{

m_bmpSmile.GetBitmap(&bm);

break;
}

case MENUTYPE_QUESTION:

{

m_bmpQuestion.GetBitmap(&bm);

break;

}

}

lpMeasureItemStruct->itemWidth=bm.bmWidth;

lpMeasureItemStruct->itemHeight=bm.bmHeight;

}
```

In this function, MENUTYPE_SMILE and MENUTYPE_QUESTION are user-defined macros that represent the type of menu items. First we examine member itemData and decide the type of the menu item. For different types of menu items, the corresponding bitmap sizes are retrieved and set to members itemWidth and itemHeight of structure MEASUREITEMSTRUCT. This size will be sent to the system and be used to calculate the layout of the whole sub-menu.

Overriding Function CMenu::DrawItem(...)

ow we need to override another member function of CMenu: CMenu::DrawItem(...). This function uses a different structure, DRAWITEMSTRUCT. It is used to inform us the state of the menu item, pass the target DC, and tell us the position where we can draw the customized menu. The following table lists some of its important members:

(Table omitted)

The following is the overridden function:

```
void MCMenu::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)

{
```

```
CDC *ptrDC;
CDC dcMem;
CBitmap *ptrBmpOld;
CBitmap *ptrBmp;
CRect rect;
if(!(lpDrawItemStruct->CtlType & ODT_MENU))
{
CMenu::DrawItem(lpDrawItemStruct);
return;
}
ptrDC=CDC::FromHandle(lpDrawItemStruct->hDC);
dcMem.CreateCompatibleDC(ptrDC);
if(lpDrawItemStruct->itemState & ODS_SELECTED)
{
switch(lpDrawItemStruct->itemData)
{
case MENUTYPE_SMILE:
{
ptrBmp=&m_bmpSmileSel;
break;
}
case MENUTYPE_QUESTION:
```

```
{
ptrBmp=&m_bmpQuestionSel;
break;
}
}
}
else
{
switch(lpDrawItemStruct->itemData)
{
case MENUTYPE_SMILE:
{
ptrBmp=&m_bmpSmile;
break;
}
case MENUTYPE_QUESTION:
{
ptrBmp=&m_bmpQuestion;
break;
}
}
}
```

```
ptrBmpOld=dcMem.SelectObject(ptrBmp);

rect=lpDrawItemStruct->rcItem;

ptrDC->BitBlt

(

rect.left,

rect.top,

rect.Width(),

rect.Height(),

&dcMem,

0,

0,

SRCCOPY

);

dcMem.SelectObject(ptrBmpOld);

}
```

First, we check if the item is a menu. If not, we call the same function implemented by the base class and return. If so, first we obtain a CDC type pointer to the target device by calling function CDC::FromHandle(...). Then, we create a compatible memory DC with target DC, which will be used to draw the bitmap. Next, we check the menu item's state and type by looking at itemState and itemData members of structure DRAWITEMSTRUCT, and choose different bitmaps according to different situations. At last we select the appropriate bitmap into the memory DC, copy the bitmap to target device using function CDC::BitBlt(...). This function has many parameters: the first four are position and size on the target device; the fifth parameter is the pointer to the memory DC; the last parameter specifies drawing mode (SRCCOPY will copy the source bitmap to the target device). Finally, we must select the bitmap out of the memory DC, and resume its original state.

Using the New Class

Now we can use this class to implement owner-draw menu. In the sample application, the first two items of dynamic menu IDR_MENU_POPUP are implemented with this style.

First a new MCMenu type variable m_menuModified is declared in class CMenuDoc (the original m_menuSub variable remain unchanged):

```
class CMenuDoc : public CDocument
{
protected:
CMenu m_menuSub;
MCMenu m_menuModified;
......
}
```

The original variable m_menuSub is used to load the menu resource, whose first sub-menu is obtained by calling function CMenu::GetSubMenu(...) and attached to variable m_menuModified. By doing this, the system will call the member functions of class MCMenu instead of CMenu when the owner-draw menu needs to be painted. To change a menu item's default style, function CMenu::ModifyMenu(...) is called and MF_OWNERDRAW flag is specified:

```
CMenuDoc::CMenuDoc()
{
CMenu *ptrMenu;
m_menuSub.LoadMenu(IDR_MENU_POPUP);
m_bmpCheck.LoadBitmap(IDB_BITMAP_CHECK);
m_bmpUnCheck.LoadBitmap(IDB_BITMAP_UNCHECK);
ptrMenu=m_menuSub.GetSubMenu(0);
m_menuModified.Attach(ptrMenu->GetSafeHmenu());
ptrMenu->ModifyMenu
(
0,
MF_BYPOSITION | MF_ENABLED | MF_OWNERDRAW, ID__POPUPITEM1,
(LPCTSTR)MENUTYPE_SMILE
);
ptrMenu->ModifyMenu
(
1,
MF_BYPOSITION | MF_ENABLED | MF_OWNERDRAW, ID__POPUPITEM2,
(LPCTSTR)MENUTYPE_QUESTION
);
m_bSubMenuOn=FALSE;
}
```

In the above code, menu IDR_MENU_POPUP is loaded into m_menuSub, then the first sub-menu is obtained and attached to variable m_menuModified. Here, function CMenu::Attach(...) requires a HMENU type parameter, which can be obtained by calling function CMenu::GetSafeHmenu(). When calling function CMenu::ModifyMenu(...), we pass integer instead of string pointer to its last parameter. This does not matter because the integer provided here will not be treated as memory address, instead, it will be passed to itemData member of structure MEASUREITEMSTRUCT (and DRAWITEMSTRUCT) to indicate the type of the owner-drawn menu items.

Because the popup menu is attached to variable CMenuDoc::m_menuModified, we need to detach it before application exits. The best place of implementing this is in class MCMenu's destructor, when the menu object is about to be destroyed:

```
MCMenu::~MCMenu()

{

Detach();

}
```

Now we can compile the sample project and execute it. By executing command Edit | Insert Dynamic Menu and expanding Dynamic Menu then selecting the first two menu items, we will see that both selected and unselected states of them will be implemented by our own bitmaps.

Bitmap is not restricted to only indicating selected and normal menu states. With a little effort, we could also use bitmap to implement other menu states: grayed, checked, and unchecked. This will make our menu completely different from a menu implemented by plain text.

2.7 Changing the Whole Menu Dynamically

If we write MDI application, we can see that the main menu may change with different type of views. Generally, when there is no client window opened, the mainframe menu will have only three sub-menus (File, View, Help). After the user opens a client window, the mainframe menu will be replaced with a new menu that has more sub-menus. Although we could call CMenu::ModifyMenu(...) to implement this, it is not efficient because with this method, we have to the change menu items one by one. Actually, there is another easy way to change the whole menu dynamically: we can prepare a new menu resource, use it to replace the menu currently associated with the window at run time.

Class CWnd has a member function that can be used to set a window's menu dynamically: CWnd::SetMenu(...). The function has only one parameter, which is a CMenu type pointer. By using this function, we can change an application's mainframe menu at any time we want. Sample 2.7\Menu demonstrates this technique. It is a standard SDI application generated by Application Wizard with all default settings. In the sample, besides mainframe menu IDR_MAINFRAME, a new menu resource IDR_MAINFRAME_ACTIVE is prepared in the application. This menu has some new sub-menus (Options, Properties, Settings). For the purpose of demonstration, none of these sub-menus contains menu items. The default menu will be changed to this menu after we execute File | New or File | Open command. If we execute File | Close command from menu IDR_MAINFRAME_ACTIVE, the default menu will be resumed.

Three WM_COMMAND message handlers for commands ID_FILE_NEW, ID_FILE_OPEN and ID_FILE_CLOSE are generated using class wizard. The member functions are named OnFileNew, OnFileOpen and OnFileClose. In the first two functions, we change the mainframe menu to IDR_MAINFRAME_ACTIVE, and in the third function, we change the menu back to IDR_MAINFRAME:

```
void CMenuDoc::OnFileNew()

{

CMenu menu;

menu.LoadMenu(IDR_MAINFRAME_ACTIVE);

AfxGetMainWnd()->SetMenu(&menu);

AfxGetMainWnd()->DrawMenuBar();

menu.Detach();

}

void CMenuDoc::OnFileOpen()

{

CMenu menu;

menu.LoadMenu(IDR_MAINFRAME_ACTIVE);

AfxGetMainWnd()->SetMenu(&menu);

AfxGetMainWnd()->DrawMenuBar();

menu.Detach();

}

void CMenuDoc::OnFileClose()

{

CMenu menu;
menu.LoadMenu(IDR_MAINFRAME);

AfxGetMainWnd()->SetMenu(&menu);

AfxGetMainWnd()->DrawMenuBar();

menu.Detach();

}
```

In the above three member functions, we use a local variable menu to load the menu resource. It will be destroyed after the function exits. So before the variable goes out of scope, we must call function CMenu::Detach() to release the loaded menu resource so that it can continue to be used by the application. Otherwise, the menu resource will be destroyed automatically.

Summary

1) In order to execute commands, we need to handle message WM_COMMAND. In order to update user interfaces of menu, we need to handle message UPDATE_COMMAND_UI.

2) To implement right-click menu, we need to first prepare a menu resource, then trap WM_RBUTTONDOWN message. Within the message handler, we can use CMenu type variable to load the menu resource, and call CMenu::TrackPopupMenu(...) to activate the menu and track mouse activities. Before the menu is shown, we can call functions CMenu::EnableMenuItem(...) and CMenu::CheckMenuItem(...) to set the states of the menu items.

3) To add or remove a menu item dynamically, we need to call functions CMenu::InsertMenu(...) and CMenu::RemoveMenu(...).

4) With function CMenu::SetMenuItemBitmaps(...), we can use bitmap images to implement checked and unchecked states for a menu item.

5) A window's standard menu can be obtained by calling function CWnd::GetMenu(), and the application's system menu can be obtained by calling function CWnd::GetSysMenu(...).

6) We can change a normal menu item to a bitmap menu item, a separator, or a sub-menu by calling function CMenu::ModifyMenu(...).

7) Owner-draw menu can be implemented by setting MF_OWNERDRAW style

then overriding functions CMenu::MeasureItem(...) and CMenu::DrawItem(...).

8) We can change the whole menu attached to a window by calling function CWnd::SetMenu(...).

BACK TO INDEX

Chapter 3 Splitter Window

A splitter window resides within the frame window. It is divided into several panes, each pane can have a different size. Splitter window provides the user with several different views for monitoring data contained in the document at the same time. Normally, the size of each pane can be adjusted freely, this gives the user a better view of data. There are two types of splitter windows: Dynamic Splitter Window and Static Splitter Window. For a dynamic splitter window, all views within the splitter window are of the same type. The user can create new panes or remove old panes on the fly. For a static splitter window, the views could be of different types and the number of panes has to be fixed at the beginning. In this case, the user can not add or delete views after the program has started.

Both SDI and MDI applications can have splitter windows. In an SDI application, the splitter window is embedded in the mainframe window. In an MDI application, it is embedded in the child frame window.

3.1 Implementing Static Splitter Windows

In MFC, class CSplitterWnd is used to implement window splitting. To split a window into several panes, we must first declare a CSplitterWnd type variable in the frame window class. One CSplitterWnd can divide window into M(N sub-panes. The splitter window can be nested, which means we can further split a single pane into several sub panes by using another CSplitterWnd type variable. So if we want to create an unevenly divided splitter window, we need to declare more than one CSplitterWnd type variables.

For example, if we want to create a splitter window that has two rows, the first row has two columns and the second row has two columns, we need to first split the client area into a 1(2 splitter window, then split the first row into a 2(1 splitter window (Figure 3-1).

(Figure omitted)

To create static splitter window, first we need to declare CSplitterWnd type variable(s) in the frame window class, then in frame window's

OnCreateClient(...) member function, call functions CSplitterWnd::CreateStatic(...) and CSplitterWnd::CreateView(...). Here, function CSplitterWnd:: CreateStatic(...) is used to split the window into several panes and CSplitterWnd::CreateView(...) is used to attach a view to each pane.

Sample application 3.1\Sdi\Spw demonstrates how to implement splitter window. It is generated by Application Wizard, with all settings set to default ones. Four main classes used to implement the application are CSpwApp, CMainFrame, CSpwDoc and CSpwView.

Each pane of the splitter window must be attached with a view in order to make it work. The view could be implemented by deriving class from any of the standard view classes: CView, CScrollView, CRichEditView, CListView, CTreeView etc. In the sample, besides the default view CSpwView created by the Application Wizard, two other classes are derived from CFormView and CEditView, which will be used to implement different panes of the splitter window.

Class CFormView is a standard MFC class that can be used to create view window from dialog template. A CFormView class must have a corresponding dialog template resource, which will be used to create the view. To implement a form view, we must first design dialog template, then derive a new class from CFormView.

In the sample, the dialog template used to implement the form view is IDD_DIALOG_VIEW. Its styles are set to "Child" and "No border", this is exactly the same with that of dialog bar (This is because both splitter window and dialog bar must be child windows). The dialog template contains a static text control and a multiple-line edit box. By double clicking on the dialog template resource, we will be prompted to add a new class for it. In this case the template resource ID will be automatically selected to be used by the new class. In the sample application, the new class is named CSpwFView. If the dialog template is not open when adding this new class, we must select the dialog ID by ourselves (Figure 3-2).

(Figure omitted)

The other pane of the splitter window is implemented using edit view. The new class for this window is derived from CEditView, and its name is CSpwEView.

To split a window, we need to call function CSplitterWnd::CreateStatic(...), which has five parameters:

(Cde omitted)

The first parameter pParentWnd is a CWnd type pointer that points to the parent window. Because a splitter window is always the child of frame window, this parameter can not be set to NULL. The second and third parameters specify the number of rows and columns the splitter window will have. The fourth parameter dwStyle specifies the styles of splitter window, whose default value is WS_CHILD | WS_VISIBLE. The fifth parameter, nID, identifies which splitter window is being created. This is necessary because within one frame window, we can create several nested splitter windows. For the root splitter window (The splitter window whose parent window is the frame window), this ID must be AFX_IDW_PANE_FIRST. For other nested splitter windows, this ID need to be obtained from the parent splitter windows by calling function CSplitterWnd::IdFromRowCol(...), and passing appropriate column and row coordinates to it. The following is the format of this function:

```
int CSplitterWnd::IdFromRowCol(int row, int col);
```

To attach a specific view to a pane, we need to call function CSplitterWnd::CreateView(...), which also has five parameters:

(Code omitted)

The first two parameters specify which pane is being created. The third parameter specifies what kind of view will be used to create this pane. Usually macro RUNTIME_CLASS must be used to obtain a CRuntimeClass type pointer. The fifth parameter is a creation context used to create the view. Within CMainFrame::OnCreateClient(...), the creation context is passed through the second parameter of this function.

In the sample application, we first use m_wndSpMain to call function CSplitterWnd:: CreateStatic(...) to split the client window into a 2(1 splitter window. Then, we use this variable to call CSplitterWnd::CreateView(...) and pass two 0s to the first two parameters of this function (This specifies (0, 0) coordinates). This will attach a new view to the left pane of the splitter window. Next we use m_wndSpSub to call CSplitterWnd::CreateStatic(...) to further split the right pane into a 1(2 splitter window, and call CSplitterWnd::CreateView(...) twice to create views for the two panes. At last, instead of calling function CMainFrame::OnCreateClient(...), a TRUE value is returned. This can prevent the default client window from being created.

The following steps show how the static splitter window is implemented in the sample:

1) Declare two CSplitterWnd type variables in class CMainFrame:

```
class CMainFrame : public CFrameWnd
```

```
{
……
protected:
CStatusBar m_wndStatusBar;
CToolBar m_wndToolBar;
CSplitterWnd m_wndSpMain;
CSplitterWnd m_wndSpSub;
……
}
```

Variable m_wndSpMain will be used to split the mainframe client window into a 2(1 splitter window, and m_wndSpSub will be used to further split the right column into a 1(2 splitter window.

2) In function CMainFrame::OnCreateClient(...), create splitter windows and attach views to each pane:

(Code omitted)

For an MDI application, everything is almost the same except that here CChildFrame replaces class CMainFrame. We can create an MDI application, declare m_wndSpMain and m_wndSpSub variables in class CChildFrame, and add code to CChildFrame::OnCreateClient(...) to create splitter windows. The code required here is exactly the same with implementing splitter window in an SDI application. Sample 3.1\MDI\Spw demonstrates this.

3.2 Dynamic Splitter Window

Once we understand how to create static splitter window, it is easier for us to create dynamic splitter window because it takes fewer steps. For a dynamic splitter window, all panes are created to be the same type of view, so there is no need to call function CSplitterWnd::CreateView(...) for each individual pane. Also, instead of calling CSplitterWnd::CreateStatic(...), we need to call function CSplitterWnd:: Create(...) to create dynamic splitter window within function CFrameWnd::OnCreateClient(...). The following is the format of function CSplitterWnd::Create(...):

```
BOOL CSplitterWnd::Create

(

CWnd* pParentWnd,

int nMaxRows, int nMaxCols,

SIZE sizeMin,

CCreateContext* pContext,

DWORD dwStyle=WS_CHILD | WS_VISIBLE |WS_HSCROLL | WS_VSCROLL |
SPLS_DYNAMIC_SPLIT,

UINT nID=AFX_IDW_PANE_FIRST

);
```

The difference between function CSplitterWnd::CreateStatic(...) and CSplitterWnd::Create(...) is that when creating dynamic splitter window, we need to specify the maximum number of rows and columns. The maximum values of nMaxRows and nMaxCols parameters are both 2, which means that a window can be split to have at most 2x2 panes.

The Application Wizard has a built-in feature to add dynamic splitter window to the applications. In step 4 of the Application Wizard, if we press "Advanced..." button, an "Advanced Options" property sheet will pop up. By clicking "Window styles" tab then checking "Use split window" check box, code will be automatically added to the application for implementing dynamic split window (static splitter window can not be created this way).

It is also simple to implement splitter window manually. Like creating static split window, first we need to declare a CSplitterWnd type variable in class CMainFrame (In MDI applications, we need to do this in class CChildFrame). Then we can use Class Wizard to override function OnCreateClient(...). Within the overridden function, we can call CSplitterWnd::Create(...) to create splitter window.

Sample 3.2\Spw demonstrates how to create dynamic splitter window in an SDI application. The application is created from Application Wizard with all default settings. Then a new variable m_wndSp is declared in class CMainFrame, which will be used to implement the splitter window. In function

CMainFrame::OnCreateClinet(...), the splitter window is created as follows:

(Code omitted)

In the above code, we did not pass any value to dwStyle and nID parameters of function CSplitterWnd::Create(...), so the default values are used.

3.3 Customizing the Behavior of Split Bar

The behavior of a dynamic splitter window is different from that of a static splitter window. For the dynamic splitter window, panes could be dynamically created by double clicking on the split boxes (Figure 3-3). After new panes are added, one or more split bars will appear. If the user double clicks any of the split bar, one f the two panes divided by that split bar will be deleted (Figure 3-4). We can examine the sample applications we've created to see the difference between static splitter window and dynamic splitter window.

(Figure omitted)

This behavior could be customized. For example, sometimes by double clicking on the split bar, we want to resize the two panes instead of deleting one of them. This feature gives the user much convenience for changing the size of each pane bit by bit.

Splitter Window Layout

We need to override the following two member functions of class CSplitterWnd in order to implement this feature: CSplitterWnd::DeleteRow(...) and CSplitterWnd::DeleteColumn(...). When the user double clicks on the split bar, one of the two functions will be called to delete a row or column dynamically. In order to customize this behavior, after the split bar is clicked, we can first change the size of each pane, then judge if the size of one pane is smaller than its minimum size. If so, we call the default implementation of the corresponding function to delete one row or column.

To change a pane's size, we need to call function CSplitterWnd::SetColumnInfo(...) and CSplitterWnd::SetRowInfo(...). The current size of a pane could be obtained by their counterpart functions CSplitterWnd::GetColumnInfo(...) and CSplitterWnd::GetRowInfo(...). The following shows the formats of the above four functions:

```
void CSplitterWnd::SetColumnInfo(int col, int cxIdeal, int cxMin);

void CSplitterWnd::SetRowInfo(int row, int cyIdeal, int cyMin);
void CSplitterWnd::GetColumnInfo(int col, int& cxCur, int& cxMin);

void CSplitterWnd::GetRowInfo(int row, int& cyCur, int& cyMin);
```

In the above functions, parameters row and col are used to identify a pane with specified row and column indices, cyIdeal and cxIdeal are the ideal size of a pane, cyMin and cxMin indicate minimum size of it.

When the splitter window is being displayed, each pane's dimension is decided from its ideal size. According to the current size of the frame window, some panes may be set to their ideal sizes, but some may not (This depends on how much space is left for that pane). In any case, a pane's actual size should not be smaller than its minimum size. This is why we need both ideal size and minimum size to set a row or column's dimension.

The number of rows and columns a splitter window currently has can be obtained by calling other two member functions of CSplitterWnd:

```
int CSplitterWnd::GetRowCount();

int CSplitterWnd::GetColumnCount();
```

After we call function CSplitterWnd::SetColumnInfo(...) or CSplitterWnd::SetRowInfo(...), the old layout will not change until we call function CSplitterWnd::RecalcLayout() to update the splitter window. The system will re-calculate the layout for each pane according to their new sizes (both ideal size and minimum size), and the split bar will be moved to a new position according to the new layout.

Overriding CSplitterWnd::DeleteRow(...) and CSplitterWnd:: DeleteColumn(...)

Sample 3.3\Spw is based on sample 3.2\Spw. In the new sample, the behavior of the split bar is modified: if the user double clicks on it, it will move a small step downward (for horizontal split bar) or rightward (for vertical split bar). A pane will be deleted after it reaches its minimum size.

In the sample application, first a new class MCSplitterWnd is derived from class CSplitterWnd:

```
class MCSplitterWnd : public CSplitterWnd

{

public:
```

```
void DeleteRow(int);

void DeleteColumn(int);

};
```

The class does nothing but overriding two functions. The implementation of function MCSplitterWnd ::DeleteRow(...) is listed as follows:

(Code omitted)

Since the maximum number of rows that can be implemented in a dynamic split window is 2, we will call the default implementation of this function (the corresponding function of the base class) if the number of rows is not 2. Otherwise, we first obtain the size of upper pane (pane 0), enlarge its vertical size, and set its current size. Then the current size of lower pane is reduced, if its ideal size is smaller than its minimum size after change, we simply call function CSplitterWnd::DeleteRow(...) to delete this row. If the panes are resized instead of being deleted, we call function CSplitterWnd::RecalcLayout() to update the new layout.

Function MCSplitterWnd::DeleteColumn(int colDelete) is implemented in the same way, except that here we call all the functions dealing with column instead of row.

Using the New Class

Using this new class is simple, we just need to include the header file containing class MCSplitterWnd in file "MainFrm.h", then use it to declare variable m_wndSpw in class CMainFrame as follows:

```
class CMainFrame : public CFrameWnd

{

protected:

CMainFrame();

DECLARE_DYNCREATE(CMainFrame)

MCSplitterWnd m_wndSp;

……
}
```

After these changes, by compiling and executing the application again, we will see that the split bar behaves differently.

3.4 Customizing the Default Appearance

Drawing Functions

Class CSplitterWnd has two member functions that can be overridden to customize the appearance of split bar, split box, split border, and split tracker. The functions are CSplitterWnd::OnDrawSplitter(...) and CSplitter::OnInvertTracker(...) respectively, which have the following formats:

```
void CSplitterWnd::OnDrawSplitter(CDC *pDC, ESplitType nType, const CRect &rect);

void CSplitterWnd::OnInvertTracker(const CRect &rect);
```

Function CSplitterWnd::OnDrawSplitter(...) is called when either the split bar, split box or split border needs to be painted. It has three parameters, the first of which is a pointer to the target device DC, which will be used to draw the objects. The second parameter is an enumerate type, which indicates what type of object is being drawn. This parameter could be either CSplitterWnd::splitBox, CSplitterWnd::splitBar, or CSplitterWnd::splitBorder, which indicates different splitter window objects. The third parameter specifies a rectangle region within which the object will be drawn.

Function CSplitterWnd::OnInvertTracker(...) is called when the user clicks the mouse on the split bar and drags it to resize the panes contained in the splitter window. In this case, a tracker will appear on the screen and move with the mouse. By default, the tracker is a grayed line. By overriding this function, we could let the tracker have a different appearance.

Sample

Sample 3.4\Spw demonstrates how to customize these styles. It is based on sample 3.3\Spw. First, two functions are declared in class MCSplitterWnd to override the default implementation:

```
class MCSplitterWnd : public CSplitterWnd

{
```

```
public:

virtual void DeleteRow(int);

virtual void DeleteColumn(int);

protected:

virtual void OnDrawSplitter(CDC*, CSplitterWnd::ESplitType, const CRect&);

virtual void OnInvertTracker(const CRect& rect);

};
```

Function MCSplitterWnd::OnDrawSplitter(...) is overridden as follows:

(Code omitted)

In the above function, first parameter pDC is checked. If it is not an available DC, we do nothing but calling the default implementation of the base class. Otherwise, the object type is checked. We will go on to implement the customization if the object is either a split bar or a split box.

The simplest way to fill a rectangle with certain pattern is to use brush. A brush can be different types: solid, hatched, etc. It could also be initialized with any color. To use a brush, we need to first create brush, then select it into the device context. If we draw a rectangle with this DC, the interior of the rectangle will be automatically filled with the currently selected brush, and its border will be drawn using the currently selected pen. After using the brush, we must select it out of the DC.

Brush selection can be implemented by calling function CDC::SelectObject(...). This function will return a pointer to the old brush. After using the brush, we can call this function again and pass the old brush to it. This will let the old brush be selected into the DC so the new brush is selected out.

When creating a brush, we need to use RGB macro to indicate the brush color. The three parameters of RGB macro indicate the intensity of red, green and blue colors.

A rectangle can be drawn by calling function CDC::Rectangle(...). We need to pass a CRect type variable to indicate the position and size of the rectangle.

In the sample, function MCSplitterWnd::OnInvertTracker(...)is implemented as follows:

(Code omitted)

There is no CDC type pointer passed to this function. However, for any window, its DC could always be obtained by calling function CWnd::GetDC(). This function will return a pointer to window's device context. After we use the DC, we must release it by calling function CWnd::ReleaseDC(...). In function MCSplitterWnd::OnDrawSplitter(...), first a solid brush with red color is created, then we select it into the DC, call function CDC::PatBlt(...) to fill the interior of the rectangle using the selected brush.

Function CDC::PatBlt(...) allows us to create a pattern on the device. We can choose different color output mode: we can copy the brush color to the destination, or we can combine brush color with the color on the target device using bit-wise operations. The first four parameters of function CDC::PatBlt(...) indicate the position and size of the rectangle within which we can output the pattern. The fifth parameter indicates the output mode. In the sample we use PATINVERT drawing mode, this will combine the destination color and brush color using bit-wise XOR operation. With this mode, the tracker can be easily erased if it is drawn twice.

Since we use PATINVERT mode to paint the tracker, its color will become the complement color of red when the user resizes panes using the mouse.

3.5 Splitter Window That Can't be Resized by Tracking

Sometimes we want each pane of the splitter window to have a fixed size and prevent the user from resizing the panes through using mouse or keyboard. Since dynamic resizing is a built-in feature of class CSplitterWnd, whenever we directly derive a class from it, we will automatically have a resizable split bar. It is not easy to disable this feature because by default, mouse clicking and dragging events will be processed automatically.

In class CSplitterWnd, four mouse messages are handled to change the state of the split bar: mouse left button down message WM_LBUTTONDOWN, mouse left button up message WM_LBUTTONUP, mouse move message WM_MOUSEMOVE, and left button double click message WM_LBUTTONDBLCLK. We need to disable only the first message handler if we want to disable tracking resize feature (Once the application cannot enter the tracking state, the rest messages will be processed normally instead of being treated as part of tracking instructions).

By default, when left button is clicked on a split bar, class CSplitterWnd will respond to this event by letting the user drag the split bar and place it to a new place. We can bypass this feature by overriding WM_LBUTTONDOWN message handler. Instead of calling the message handler implemented by CSplitterWnd, we can call the default function implemented by class CWnd, which is the base class of CSplitterWnd. For a dynamic splitter window, the WM_LBUTTONDBLCLK message handler should not be overridden because after we disable the tracking, double clicking becomes the only way that can be used by the user to dynamically add or delete panes. For WM_MOUSEMOVE and WM_LBUTTONUP messages, we don't need to modify their handlers because after message WM_LBUTTONDOWN is bypassed, the tracking will not happen anymore.

Sample 3.5\Spw demonstrates how to implement splitter window that cannot be resized through tracking the split bar. It is based on sample 3.4\Spw.

To let class MCSplitterWnd support both resizable and non-resizable split bars, a new Boolean type variable m_bResizable is declared in the class. Along with this variable a new member function MCSplitterWnd::SetResizable(...) is also declared, which can be called to set m_bResizable flag and indicate if tracking resize feature is currently supported. At the beginning variable MCSplitterWnd::m_bResizable is initialized to TRUE in the constructor. The following code fragment shows the modified class:

(Code omitted)

A WM_LBUTTONDOWN message handler is added to the application. This includes function declaration, adding ON_WM_LBUTTONDOWN message mapping macro, and the implementation of member function. Before adding message mapping, we need to make sure that DECLARE_MESSAGE_MAP macro is included in the class. This will enable massage mapping for the class. The following lists necessary steps for implementing the above message mapping:

1) Declare an afx_msg type member function OnLButtonDown(...) in the class. This function is originally declared in class CWnd, here we must declare it again in order to override it:

```
class MCSplitterWnd : public CSplitterWnd

{

……

protected:

……

afx_msg void OnLButtonDown(UINT, CPoint);
DECLARE_MESSAGE_MAP()

};
```

2) In the implementation file, add ON_WM_LBUTTONDOWN macro between BEGIN_MESSAGE_MAP and END_MESSAGE_MAP macros:

```
BEGIN_MESSAGE_MAP(MCSplitterWnd, CSplitterWnd)

//{{AFX_MSG_MAP(MCSplitterWnd)

//}}AFX_MSG_MAP

ON_WM_LBUTTONDOWN()

END_MESSAGE_MAP()
```

Macro ON_WM_LBUTTONDOWN maps message WM_LBUTTONDOWN to function OnLButtonDown(...).

3) Implement the message handler as follows:

```
void MCSplitterWnd::OnLButtonDown(UINT uFlags, CPoint point)

{

if(m_bResizable == TRUE)CSplitterWnd::OnLButtonDown(uFlags, point);

else CWnd::OnLButtonDown(uFlags, point);

}
```

The function implementation is simple. If the splitter window is trackable, we call CSplitterWnd::OnLButtonDown(...), which will implement tracking if mouse cursor is over the split bar. Otherwise we bypass this feature by calling function CWnd::OnLButtonDown(...).

If the splitter window is created by the Application Wizard, there will be a command View | Split implemented in the application. By default, this command gives an alternate way to resize panes by tracking the split bar. If we want to disable the tracking completely, we also need to disable or remove this menu command.

Summary

1) To implement static splitter window, we need to derive a class from CView (or other type of view classes) for each pane, then declare a CSplitterWnd type variable in class CMainFrame. In function CMainFrame::OnCreateClient(...), we need to call CSplitterWnd::CreateStatic(...) to create the splitter window and call CSplitterWnd::CreateView(...) to attach a view to each pane.

2) Static splitter window can be nested. This means instead of attaching a view, we can use CSplitterWnd to further create splitter window within a pane.

3) Creating dynamic splitter window is simple. In order to do this, we need to declare a CSplitterWnd type variable in class CMainFrame; then in CMainFrame::OnCreateClient(...), we need to call function CSplitterWnd::Create(...).

4) We can override functions CSplitterWnd::DeleteRow(...) and CSplitterWnd::DeleteColumn(...) to customize the behavior of split bars.

5) We can override functions CSplitterWnd::OnDrawSplitter(...) and CSplitterWnd:: OnInvertTracker(...) to customize the appearance of split bar, split box, split border and tracker.

6) To disable split bar tracking, we need to call CWnd::OnLButtonDown(...) instead of CSplitterWnd:: OnLbuttonDown(...) when handling message WM_LBUTTONDOWN.

BACK TO INDEX

Chapter 4 Buttons

Button is one of the most commonly used controls, almost every application needs to include one or more buttons. It seems that it is very easy to implement a standard button in a dialog box: all we need to do is adding a button resource to the dialog template, then using Class Wizard to generate message handlers. However, it is not very easy to further customize button's default properties.

In this chapter, we will discuss how to implement different type of customized buttons. At the end of this chapter, we will be able to include some eye-catching buttons in our applications.

4.1 Bitmap Button: Automatic Method

Generally, buttons display plain text on its interface. Sometimes it is more desirable to let them have graphic user interface. A typical application that uses this type of buttons would be a CD player, everyone would like the play button to have a graphic interface instead of just displaying text such as "play", "stop" (so that it looks like a real "play" button).

Button States

Before customizing button's default feature, it is important for us to understand some basics of buttons. Every button has four states. When a button is not clicked, is in "up" state (the most common state). When it is pressed down, it is in "down" state. To emphasis a button's 3D effect, a default button will recess when it is pressed by the mouse. Also, a button could be disabled, in this state, the button will not respond to any mouse clicking (As the default implementation, when a button is disabled, it will be drawn with "grayed" effect). Finally, a button has a "focused" or "unfocused" state. In the "focused" state, the button is an active window, and is accessible through using keyboard (ENTER or downward ARROW key). For the default implementation, a rectangle with dashed border will be drawn over a button's face when it has the current focus.

Owner-Draw Bitmap Button

We can use owner draw bitmap button to add graphics to the button. To distinguish among different states, we need to associate different states with different images if necessary.

The simplest way to create this type of button is to implement bitmap button using class CBitmapButton. To create a bitmap button, first we must set its "Owner Draw" style. This can be implemented by checking "Owner Draw" check box in the "Push Button Properties" property sheet when creating the button resource (We need to add button resource in order to create button, this is the same with a normal button. See Figure 4-1). For an owner-draw button, message WM_DRAWITEM will be sent to the button when it needs to be painted (remember, a button is also a window that can receive message). Upon receiving this message, the button will be drawn by the overridden function. For non-owner-draw button, its interface is implemented by the default method, and will display a plain text on the button's face.

Class CBitmapButton handles message trapping and processing; also, it contains member functions that can be used to paint the button. If we use this class to implement bitmaps buttons, all we need to do is preparing some bitmap resources, declaring variables using class CBitmapButton, and associating bitmap resources with the corresponding buttons.

Although every button has four states, we do not need to provide four bitmaps all the time. If one of the bitmaps is not available, class CBitmapButton will draw the button's corresponding state using the default bitmap, which is the bitmap associated with button's "up" state. So the bitmap used to represent a button's "up" state is required all the time and can not be omitted.

Automatic Method

We have two ways of associating bitmap images with different states of a bitmap button: we can either let class CBitmapButton handle this automatically or we can do it manually. To use the automatic method, the IDs of all four bitmap resources must be text strings, and must be formed by suffixing one of the following four letters to the button's caption text: 'U', 'D', 'F', 'X'. These letters represent "up", "down", "focused" and "disabled" state respectively. By naming the resource IDs this way, the rest thing we need to do is calling function CBitmapButton::AutoLoad() in the dialog box's initialization stage (within member function CDialog::OnInitDialog()). Please note that we cannot call this function in the constructor of class CDialog. At that time, the dialog box window is still not created (Therefore, the buttons are still not available), and the bitmaps cannot be associated with the button correctly.

Sample

Sample 4.1\Btn demonstrates how to create bitmap button using automatic method. It is a dialog-based application that is generated by the Application Wizard. First, the ID of default dialog template is changed to IDD_DIALOG_BTN. Also, the "OK" and "Cancel" buttons originally included in the template are deleted. Then a new button IDC_PLAY is added, whose caption text is set to "Play" (Figure 4-2). Since the button will be drawn using the bitmaps, it doesn't matter how big the button resource is. Besides this, we need to set button's style to "Owner draw".

Two bitmap resources are added to the application whose IDs are "PLAYU" and "PLAYD" respectively (Figure 4-3). They correspond to button's "up" and "down" states. In addition, the sizes of the two bitmaps are exactly the same.

A CBitmapButton type variable is declared in class CBtnDlg to implement this bitmap button:

```
class CBtnDlg : public CDialog
{
……
protected:
……
CBitmapButton m_btnPlay;
……
};
```

Within function CBtnDlg::OnInitDialog(), function CBitmapButton::AutoLoad(...) is called to initialize the bitmap button and associate it with the corresponding bitmap resources. After this call, we don't need to do anything. The bitmap button will be created and its states will be set automatically:

```
BOOL CBtnDlg::OnInitDialog()
{
……
m_btnPlay.AutoLoad(IDC_PLAY, this);
……
}
```

Function CBitmapButton::AutoLoad(...) has two parameters, first of which is the button's resource ID, and the second is a pointer to the button's parent window.

In the sample application only two bitmap images are prepared. We may add two other bitmaps whose IDs are "PLAYF" and "PLAYX". Then we can enable or disable the button to see what will happen to the button's interface.

4.2 Bitmap Check Box and Radio Button: Method 1

The bitmap button implemented this way behaves like a push button. Unfortunately, in MFC, there is no class such as CBitmapCheckBox and CBitmapRadioButton to let us implement bitmap check box or radio button. However, check box and radio button are another two types of buttons, both of them can be implemented using class CButton.

A button implemented by class CButton can display either plain text or bitmap. Actually, there is a member function of CButton that allows us to associate button with a bitmap: CButton::SetBitmap(...).

If a button implemented by class CButton can also be associated with a bitmap, what is the difference between CBitmapButton and CButton? First, CBitmapButton is derived from CButton, so it has more features than CButton. For example, with CBitmapButton, we can use automatic method to associate button with bitmap resources by calling function CBitmapButton::AutoLoad(...). Second, CButton allows only one bitmap to be associated with a button at any time, and it always implement the focus state of the button by drawing a dash-bordered rectangle over button's face.

Although only one bitmap could be associated with a button at any time, we still can manage to represent a button's different states using different bitmaps.

The trick here is to change button's associated bitmap whenever its state changes. To achieve this, we must implement a WM_COMMAND message handler for the button. For example, in the case of check box, we can find out whether the current state of the check box is "Checked" or "Unchecked". Based on this information, we can decide which bitmap should be used.

Sample 4.2\Btn demonstrates the above method. It is based on sample 4.1\Btn. In the sample, three new buttons are added: one of them is implemented as a check box; the rest are implemented as radio buttons. The following describes how the bitmap check box and radio buttons are implemented in the sample:

1) Add a check box and two radio buttons to the dialog template. Name the IDs of new controls IDC_CHECK, IDC_RADIO_A and IDC_RADIO_B respectively. In the property sheet that lets us customize control's properties, check "Bitmap" check box (Figure 4-4).

(Figure omitted)

2) Add two bitmap resources, one for checked state and one for unchecked state. Their resource IDs are ID_BITMAP_CHECK and ID_BITMAO_UNCHECK respectively. The bitmaps must have a same size.

3) Declare two CBitmap type variables m_bmpCheck and m_bmpUnCheck in class CBtnDlg, in function CBtnDlg::OnInitDlg(), call CBitmap::LoadBitmap(...) to load the two bitmap resources. Then call function CButton::SetBitmap(...) to set bitmap for the check box and radio buttons. In the sample, all of the new controls are initialized to unchecked state (In order to do this, we need to associate buttons with m_bmpUnCheck instead of m_bmpCheck). The following code fragment shows the modified class CBtnDlg and the function CBtnDlg::OnInitDialog(...):

(Code omitted)

4) Declare a new member function CBtnDlg::SetCheckBitmap(...). We will use it to set a button's bitmap according to its current state. The function has one parameter nID that identifies the control. Within the function, first the button's current state is examined, if it is checked, we call CButton::SetBitmap(...) to associate it with IDB_BITMAP_CHECK; otherwise we use bitmap IDB_BITMAP_UNCHECK. The following is the implementation of this function:

(Code omitted)

5) Use Class Wizard to implement three WM_COMMAND message handlers for IDC_CHECK, IDC_RADIO_A and IDC_RADIO_B. Within each handler, we call CBtnDlg::SetCheckBitmap(...) to set appropriate bitmaps. Because two radio buttons should be treated as a group (if one is checked, the other one will be unchecked automatically), we need to set both button's bitmaps within each message handler:

(Code omitted)

In step 3, when calling CWnd::GetDlgItem(...), we pass the control's resource ID to the function to obtain a pointer to the control and use it to call function CButton::SetBitmap(...). Because CWnd::GetDlgItem(...) will return a CWnd type pointer, we must first cast it to CButton type pointer before calling the member function of CButton.

Function Cbutton::SetBitmap(...) has an HBITMAP type parameter, which requires a valid bitmap handle. A bitmap handle can be obtained by calling function CBitmap::GetSafeHandle(), of course, the returned handle is valid only after the bitmap is loaded.

In step 4, function CButton::GetCheck() is called to retrieve button's current state (checked or unchecked). The function returns a Boolean type value, if the returned value is TRUE, the button is being checked, otherwise it is not checked.

After these modifications, the bitmap check box and radio buttons will become functional.

4.3 Subclass

In section 4.1, we used automatic method to create bitmap buttons. This requires us to create owner-draw buttons with special caption text, which will be used to name the bitmap resource IDs. For simple cases, this is a very convenient method. However, if we implement bitmap buttons this way, it is difficult for us to customize them at runtime.

Implementing Subclass

Class CBitmapButton gives us another member function that can be used to associate bitmaps with an owner-draw button: CBitmapButton::LoadBitmaps(...). This function has two versions, the first version allows us to load bitmaps with string IDs, the second version allows us to load bitmaps with integer IDs.

To use this function, we must first implement subclass for the owner-draw button. "Subclass" is a very powerful technique in Windows? programming. It allows us to write a procedure, attach it to a window, and use it to intercept messages sent to this window then process it. By doing this, we are able to customize the window's behavior within the procedure.

Subclass is supported by class CWnd, so theoretically all windows (including client window, dialog box, dialog common controls...) can be "subclassed". There are two functions to implement subclass, one is CWnd::SubclassWindow(...), which allows us to customize the normal behavior of a window. Here we will use the other one: CWnd::SubclassDlgItem(...), which is specially designed to implement subclass for the common controls contained in a dialog box.

In MFC, implementing subclass is very simple. We don't need to write a special procedure to handle the intercepted messages. All we need to do is designing a class as usual, adding message handlers for the messages we want to process, and implementing the message handlers. Then we can declare a variable using the newly designed class, and call function CWnd::SubclassDlgItem(...) to implement subclass.

Function CWnd::SubclassDlgItem(...) has two parameters:

BOOL CWnd::SubclassDlgItem(UINT nID, CWnd *pParent);

Parameter nID indicates which control we are dealing with, and pParent is the pointer to the control's parent window.

Class CBitmapButton uses subclass to change the default behavior of a button. If we use automatic method to load the bitmaps, the subclass procedure is transparent to the programmer. However, if we want to load the bitmaps by ourselves, we must implement subclass first.

Bitmap Button

Sample 4.3\Btn demonstrates how to associate bitmaps with an owner draw button by calling function CBitmapButton::LoadBitmaps(...). It is based on sample 4.2\Btn. There is nothing new in this sample, except that button IDC_PLAY is implemented differently.

In the previous samples, variable CBtnDlg::m_btnPlay is declared as a CBitmapButton type variable. In the new sample, instead of using automatic method to load the bitmaps, we first implement the subclass then load the bitmaps manually in function CBitmapButton::LoadBitmaps(...):

(Code omitted)

Here, function CBitmapButton::AutoLoad(...) is replaced by three new functions. The first function added is CWnd::SubclassDlgItem(...). The second function is CBitmapButton::LoadBitmaps(...). This function has four parameters, which are the bitmap IDs corresponding to button's "Up", "Down", "Focused" and "Disabled" states respectively. They could be either string IDs or integer IDs. The last function is CBitmap::SizeToContent(), which allows us to set bitmap button's size to the size of the associated bitmaps. If we don't call this function, the bitmaps may not fit well into the button.

Now we can remove or modify bitmap button IDC_PLAY's caption text "Play. Actually, it doesn't matter if the button has caption text or not. By compiling the application and executing it at this point, we will see that the bitmap button implemented here is exactly the same as the one implemented in the previous sample.

4.4 Bitmap Check Box and Radio Button: Method 2

In sample 4.2\Btn, although we can represent the checked and unchecked states of a check box or a radio button using different bitmaps, we could not customize their "focused" state. When a button has the current focus, a rectangle with dashed border will always be put over button's face (Figure 4-5). This is because we use CButton instead of CBitmapButton to implement buttons, this allows only one bitmap to be associated with a button.

(Figure omitted)

To improve this, we can use class CBitmapButton to create both check box and radio button. By doing this, the button's focused state will be implemented using the bitmap image provided by the programmer instead of drawing a rectangle with dashed border over button's face. Since class CBitmapButton supports only push button implementation, we need to change the bitmap by ourselves to imitate check box and radio button.

Sample 4.4\Btn is based on sample 4.3\Btn. In this sample, three new buttons are added to the dialog template: one will be implemented as a check box; the other two will be implemented as radio buttons. All of them will be based on class CBitmapButton.

First, we need a Boolean type variable for each check box and radio button to represent its current state. This variable toggles between TRUE and FALSE, indicating if the button is currently checked or unchecked. When the state of a button changes, we re-associate the button with an alternate bitmap and paint the bitmap button again.

Since we use push button to implement check box and radio button, we can not call function CButton::GetCheck(...) to examine if the button is currently checked or not. This is because a push button will automatically resume to the "up" state after the mouse is released.

In the sample application, three new buttons are added to the dialog template IDD_BTN_DIALOG, and their corresponding IDs are IDC_BMP_CHECK, IDC_BMP_RADIO_A, IDC_BMP_RADIO_B respectively. Also, they all have a "Owner draw" style. Besides the new controls, two new bitmap resources are also added to the application, which will be used to implement button's "Checked" and "Unchecked" states. The IDs of the new bitmap resources are IDB_BITMAP_BTNCHECK and IDB_BITMAP_BTNUNCHECK. The difference between the new bitmaps and two old ones (whose IDs are IDB_BITMAP_CHECK and IDB_BITMAP_UNCHECK) is that the new bitmaps have a 3-D effect. In sample 4.2\Btn, the check box and radio buttons are implemented using class

CButton, which automatically adds 3-D effect to the controls. Since we want the controls to be implemented solely by programmer-provided bitmaps, we need to add 3-D effect by ourselves.

In the sample application, three new CBitmapButton type variables are declared in class CBtnDlg. Also, a new member function SetRadioBitmap() is added to associate bitmaps with the two radio buttons. This function will be called when one of the radio buttons is clicked by mouse. For the check box, associating bitmap with it is relatively simple, so it is implemented within the message handler. Besides this, a new Boolean type variable CBtnDlg::m_bBmpCheck is declared to indicate the current state of the check box, and an unsigned integer CBtnDlg::m_uBmpRadio is declared to indicate which radio button is being selected. For each button, WM_COMMAND message handler is added through using Class Wizard. These message handlers are CBtnDlg::OnBmpCheck(), CBtnDlg::OnBmpRadioA() and CBtnDlg::OnBmpRadioB() respectively. The following code fragment shows the new members added to the class:

(Code omitted)

In the constructor of CBtnDlg, variable CBtnDlg::m_bBmpCheck is initialized to FALSE and CBtnDlg::m_uBmpRadio is initialized to zero. This means the original state of the check box is unchecked, and no radio button is selected:

```
CBtnDlg::CBtnDlg(CWnd* pParent /*=NULL*/)

: CDialog(CBtnDlg::IDD, pParent)

{

......

m_bBmpCheck=FALSE;

m_uBmpRadio=0;

}
```

In the dialog initialization stage, subclass is implemented for buttons, then the corresponding bitmaps are loaded:

(Code omitted)

For each button, first we implement subclass for it, then we load the bitmap by calling function CBitmapButton::LoadBitmaps(...). Because we provide only one bitmap for each control, state transition of these buttons (e.g., normal state to focused state) will not be reflected to the interface unless we add extra code to handle it.

For check box IDC_BMP_CHECK, when it is clicked by the mouse's left button, we need to toggle the value of variable CBtnDlg::m_bCheck, load the corresponding bitmap and paint the button again. The following is the WM_DOMMAND message handler for check box:

(Code omitted)

Radio buttons are slightly different. When one radio button is checked, the other button should be unchecked. So within both CBtnDlg::OnBmpRadioA() and CBtnDlg::OnBmpRadioB(), function CBtnDlg:: SetRadioBitmap() is called to set bitmaps for both radio buttons:

(Code omitted)

When the user clicks one of the radio buttons, its resource ID is assigned to variable CBtnDlg::m_uBmpRadio. Then in function CBtnDlg::SetRadioBitmap(), this variable is compared to both radio button IDs. Bitmap IDB_BITMAP_BTNCHECK will be associated to the button whose ID is currently stored in variable CBtnDlg::m_uBmpRadio. For the other button, bitmap IDB_BITMAP_BTNUNCHECK will be loaded.

The appearance of our new check box and radio buttons is almost the same with that of old ones implemented in sample 4.2\Btn. However, here the rectangle with dashed border will not be put over a button's face when the button has the current focus (Figure 4-6).

(Figure omitted)

4.5 Irregular Shape Bitmap Button

Transparent Background

Up to now all the buttons created by us have a rectangular shape. It is relatively difficult to create a button with irregular shapes (e.g., a round button). Even for bitmap buttons, their associated bitmaps are always rectangular.

We may think that by setting the bitmap's background color to the color of the dialog box, the bitmap button may look like a non-rectangular button. For example, in the previous samples, button IDC_PLAY is made up of two regions: its foreground is the triangular region, and rest area can be treated as its background (Figure 4-7). We can change the background color to the color of the dialog box so that this area appears transparent to us when the bitmap button is implemented.

However, this is not the best solution. In Windows? operating system, the color of dialog box can be customized. The user can double click "Display" icon contained in the "Control Panel" window and set the colors for many objects, which include title bar, backdrop..., and so on. So actually we have no way of knowing the color of dialog box beforehand. If we implement a bitmap button and set its background color to the dialog box color in our system, it may not work properly in another system.

To draw a bitmap with transparent background, we need to use "mask" when painting the bitmap. We can imagine a mask as a matrix with the same dimension of the bitmap image, and each element in the matrix corresponds to a pixel contained in the bitmap. The elements in the matrix may be either "0" or "1". When we paint the bitmap, only those pixels with "0" masks are output to the device (we can also use "1" to indicate the pixels that need to be drawn).

When programming the application, we can prepare two bitmaps of exactly the same size: one stores the normal image, the other one stores mask. The mask bitmap is made up of only two types of pixels: black and white. This is because for black color, its RGB elements are all 0s; for white color, the elements are all 1s. By implementing the mask bitmap like this, the background area of the mask bitmap is white and the foreground area is black.

Windows allows us to output the pixels of a bitmap to the target device using different operation mode. We can copy a pixel directly to the target device, we can also combine a pixel contained in the bitmap with the pixel contained in the target device using various mode: bit-wise OR, AND, and XOR. For example, if we combine red color (RGB(255, 0, 0)) with green color (RGB(0, 255, 0)) using bit-wise AND operation, the output will be yellow color (RGB(255, 255, 0)).

When painting the bitmap, we first need to output the normal bitmap to the target device using bit-wise XOR drawing mode, then output the mask bitmap to the target device at the same position using bit-wise AND mode. Finally we can output the normal bitmap again using bit-wise XOR mode. By doing this, the output bitmap's background will become transparent.

The reason for this is simple. Before bitmap is painted, the target device may contain a uniform color or any image pattern. After normal bitmap is first painted, the foreground area of the target device will become the combination of source image pattern and target image pattern. After we output mask bitmap using bit-wise AND operation, the foreground area of the target device will become black. This means on the target device, every pixel in the foreground area is zero now. Since we use bit-wise XOR mode to output normal image in the last step, and XORing anything with zero will not change the source, the foreground area on the target device will finally contain the pattern of the source image. For mask area, the second ANDing operation doesn't make any change to it because ANDing a pixel with white color (all 1s) doesn't change that pixel. So the overall operations on the mask region is equivalent to two consecutive XOR operations, which will resume all the pixels in this region to their original colors.

However there is still a slight problem here: if we draw the source and mask bitmaps directly to the target device using the method mentioned above, we will see a quick flicker on the mask area of the target device. Although it lasts only for a very short period, it is an undesirable effect. To overcome this, we can prepare a bitmap in the memory, copy the target image pattern to this bitmap, do the transparent painting on the memory bitmap, and copy the final result back to the target. Since the background area of the memory bitmap has the same pattern with the background area of the target device, this final copy will not cause any flicker.

To customize the drawing behavior of bitmap button, we need to override function CBitmapButton::OnDrawItem(...). For an owner-draw button, this function will be called when a button needs to be updated. Actually, menu and combo box also use similar functions. We can create an owner draw menu or combo box by overriding the member functions with the same name. For these functions, their parameters are all pointers to DRAWITEMSTRUCT type object. This structure stores information such as button's current state (i.e. focused, disabled), the device context that can be used to implement drawing, and the rectangle indicating where we can output the image pattern.

New Class

Sample 4.5\Btn is based on sample 4.4\Btn, it demonstrates how to create bitmap buttons with transparent background. In this sample, a new class MCBitmapButton is derived from CBitmapButton. Besides the default properties inherited from base class, the following new features are added to this class:

1) The new class handles transparent background drawing automatically. Programmer can prepare a black-and-white mask bitmap and associate it with the bitmap button together with other required bitmaps. The drawing will be taken care in the member function of the class. Programmer doesn't need to add extra code.

2) The mask bitmap can be loaded along with other images by calling either AutoLoad(...) or LoadBitmaps(...).

3) Function AutoLoad(...) is overridden to load the mask image automatically. In this case, the mask bitmap must have a string ID that is created by suffixing an

'M' character to button's caption text. For example, if we want to create mask bitmap for a button whose caption text is "PLAY", the ID of the mask bitmap must be "PLAYM". If we load the mask bitmap using automatic method, there is no difference between using the new class and CBitmapButton.

4) Mask bitmap could also be loaded by calling function LoadBitmaps(...). The overridden function has five parameters, the last of which is the ID of the mask bitmap.

5) If the mask bitmap is not present, the bitmap will be output directly to the target device using the default implementation.

In the sample, a mask bitmap "PLAYM" is added to the application. It will be used to draw button IDC_PLAY with transparent background.

The new class derived from CBitmapButton is MCBitmapButton. In this class, a new CBitmap type variable m_bitmapMask is added to load the mask bitmap, also, functions AutoLoad(...) and LoadBitmaps(...) are overridden. The following code fragment shows this new class:

(Code omitted)

Function LoadBitmaps(...) has two versions, one is used to load bitmaps with string IDs, the other is used to load bitmaps with integer IDs. Both functions have five parameters.

Overriding Function CBitmapButton::LoadBitmaps(...)

When overriding functions, we can utilize the features implemented by the base class to load standard four bitmaps, and add our own code to load the mask bitmap. The following is the implementation of function MCBitmapButton::LoadBitmaps(...):

(Code omitted)

First we must use variable m_bitmapMask to call function CBitmap::DeleteObject(), which is inherited from class CGdiObject. Since bitmap is a GDI (graphics device interface) object, once it is initialized, it will allocate some memory. If we want to initialize it again, we must first release the previously allocated memory. Function CGdiObject::DeleteObject() can be used for this purpose.

Next we call function CBitmapButton::LoadBitmaps(...) to load the default four bitmaps, and see if the mask bitmap is available. If so, we use m_bitmapMask to load the mask bitmap.

Overriding Function CBitmapButton::AutoLoad(...)

In member function MCBitmapButton::AutoLoad(...), the bitmap resource IDs are obtained by suffixing 'U', 'D', 'F', 'X' or 'M' to button's caption text. With the resource IDs, function MCBitmapButton:: LoadBitmaps(...) is called to load the relevant bitmaps:

(Code omitted)

The caption text of the button can be retrieved by calling function CWnd::GetWindwoText(...). Actually, every window can have a caption text, which can be an empty string, or any text. We can use this function to retrieve the caption text of any window, for example, a title tar.

We need to call functions CWnd::SubclassDlgItem(...) and CBitmapButton::SizeToContent() to change the button's default properties. Actually, the above two functions are also called in function CBitmapButton::AutoLoad(...).

Overriding Function CBitmapButton::DrawItem(...)

Then we need to implement MCBitmapButton::DrawItem(...). In this member function, we need to check the current state of button, and choose appropriate bitmaps for painting button's face.

Class CBitmapButton has four CBitmap type variables: m_bitmap, m_bitmapSel, m_bitmapFocus and m_bitmapDisabled, which are used to store the standard four bitmaps. Because they are not declared as private members, we can access them from the derived classes and use them to paint the button.

The only parameter of function CBitmapButton::DrawItem(...) is a pointer to a DRAWITEMSTRUCT type object. When overriding this function, we need to use four members contained in DRAWITEMSTRUCT: itemState, which indicates the current state of the button; hDC, which is the handle to the target device context; rcItem, which is the rectangle specifies the position and size of the output area. Besides these, we also need to check the following bits of member itemState: ODS_SELECTED, ODS_FOCUS, ODS_DISABLED, which indicate if the current state of button is "selected", "focused" or "disabled".

At the beginning of the overridden function, we need to declare several CBitmap and CDC type variables or pointers, all of which are used for bitmap drawing:

(Code omitted)

Three DCs are declared here. To draw a bitmap, we must create a memory DC, select the bitmap into it and copy the bitmap from the memory DC to the target DC. The target could be either a device or a memory block (we could copy bitmap between two memory DCs). A DC can select only one bitmap at any time.

When there is no mask bitmap, variable memDC is used to perform normal bitmap drawing: it is used to select the normal bitmap, and copy it directly to the target DC. When there is a mask bitmap, memDC will be used along with memDCMask to implement transparent background drawing.

Variable memDCImage is used to act as the target memory DC and implement transparent background drawing. It will be used in conjunction with bmpImage, which will be selected into memDCImage. To draw the bitmap, first we need to copy the image pattern from the target device to the memory bitmap, then copy the source bitmap to the memory bitmap (perform AND and XOR drawings). Finally, we can output the result from the memory bitmap to the target device.

Variable bmpImage is used to create bitmap in memory.

Variable memDCMask is used to select mask bitmap image.

Pointer pDC will be used to store the pointer of the target device context that is created from hDC member of structure DRAWITEMSTRUCT.

Pointers pBitmap and pBitmapMask will be used to store the pointers to the normal bitmap (could be one of the bitmaps indicating the four states of the button) and the mask bitmap respectively.

The other three CBitmap pointers pOld, pOldMask and pOldImage are used to select the bitmaps out of the DCs (When the bitmaps are being selected into the DCs, these pointers are used to store the bitmaps selected out of the DCs. After bitmap drawing is finished, we can select old bitmaps back into the DCs, this will select our bitmaps out of the DCs automatically).

Variable state is used to store the current state of button.

The following portion of function MCBitmapButton::DrawItem(...) shows how to choose appropriate bitmaps:

(Code omitted)

First pBitmap is assigned the address of variable m_bitmap, which holds the default bitmap. Then we check if the mask bitmap exists, if so, we assign its address to pBitmapMask. The current state of the button is read into variable state, whose ODS_SELECTED, ODS_FOCUS and ODS_DISABLED bits are examined in turn. If any of them is set, the corresponding bitmap's address will be stored in pBitmap.

The following portion of this function creates the memory DCs and selects relative bitmaps into different DCs:

(Code omitted)

First, the address of the target DC is obtained from the DC handle by calling function CDC:: FromHandle(...). Then the memory DC that will select source image is created by calling function CDC::CreateCompatibleDC(...). Since the bitmap could be copied only between compatible DCs, each time we create a memory DC, we need to make sure that it is compatible with the target DC. Next, if the mask bitmap exists, we create three DCs: memDC for normal bitmap, memDCMask for mask bitmap and memDCImage for memory target bitmap (It will act as temparory target device DC). In this case, we also create a memory bitmap using variable bmpImage, which is selected into memDCImage (This bitmap must also be compatible with the DC that will select it). In the above implementation, we call function CBitmap::GetBitmap(...) to obtain the dimension information of a bitmap and call function CBitmap::CreateCompatibleBitmap(...) to create compatible memory bitmap). The mask bitmap is selected into memDCMask. The normal bitmap is always selected into memDC.

The following portion of function MCBitmapButton::DrawItem(...) draws the bitmap by copying normal and mask bitmaps among different DCs:

(Code omitted)

Bitmap copy is implemented by calling function CDC::BitBlt(...). This function will copy the selected bitmap from one DC to another. If there is no mask bitmap, we copy the normal bitmap (selected by memDC) directly to the target DC (pointed by pDC). Otherwise, first we copy the image pattern from the target device (pDC) to memory bitmap (selected by memDCImage). Then we copy normal bitmap and mask bitmap (selected by memDC and memDCMask) to this memory bitmap three times, using different operation modes, and copy the final result to the target DC (pDC). At last, we select the bitmaps out of DCs.

Using Class MCBitmapButton

In sample 4.5\Btn, automatic method is used to load the mask bitmap. There are two differences between this sample and sample 4.4\Btn. First, in the new sample, a new bitmap resource "PLAYM" is added to the application that is used as the mask bitmap. Second, variable CBtnDlg::m_btnPlay is declared using

MCBitmapButton instead of CBitmapButton (Also, we need to include the header file of MCBitmapButton). No other change is needed.

With the above implementation, the button will have a transparent background. We can test this by re-configuring the system colors.

4.6 Making Button Aware of Mouse Position

By now no button we've made could provide us with information of mouse position when it is pressed. Although most of the time it is not necessary to know the exact coordinates of the mouse cursor, it may help us create more powerful buttons if we have this information. For example, we can create a bitmap button with four arrows pointing to different directions. When the user clicks mouse on different arrows, we can let different commands be executed if we know the current cursor position.

Since class CButton is derived from CWnd, and CWnd handles different types of mouse events, we should be able to trap mouse related messages into the member functions of CButton. Actually, all classes derived from CWnd can trap mouse events such as left button down, left button up, left button double click, etc. In order to implement the button described above, we need to trap left button up message, which is defined as WM_LBUTTONUP under Windows.

Sample 4.6\Btn demonstrates how to handle mouse-related message for a button. It is based on sample 4.6\Btn, with a new button added to the application. The new button has four arrows pointing to different directions, if the user clicks on any of the arrows, a message box will pop up displaying a different message (Figure 4-8).

(Figure omitted)

Trapping Message WM_LBUTTONUP within Button

In the sample, WM_LBUTTONUP message handler is added to class MCBitmapButton. Like trapping any other type of message, in order to handle WM_LBUTTONUP, we need to declare an afx_msg type function and add message mapping macros. If the class is created by Class Wizard, this procedure could be very easy. Otherwise, we must add everything manually.

First we need to declare an afx_msg type member function in the class:

```
class MCBitmapButton : public CBitmapButton

{
```

```
……
protected:
……
afx_msg void OnLButtonUp(UINT, CPoint);
……
DECLARE_MESSAGE_MAP()
};
```

There must be a DECLARE_MESSAGE_MAP macro in the class in order to let it support message mapping.

The message mapping macros are added to the implementation file as follows:

```
BEGIN_MESSAGE_MAP(MCBitmapButton, CBitmapButton)
ON_WM_LBUTTONUP()
END_MESSAGE_MAP()
```

Message WM_LBUTTONUP will be trapped to member function MCBitmapButton::OnLButtonUp(...), which has the following format:

```
void MCBitmapButton::OnLButtonUp(UINT nFlags, CPoint point)
{
}
```

We see that the mouse position is passed to parameter point of this function.

Because the commands are generally handled in the parent window of the button, we need to resend mouse clicking information from the button to class CBtnDlg. In the sample application, this is implemented through sending a user-defined message.

User-Defined Message

User defined messages can be treated the same with other standard Windows messages: they can be sent from one window to another, and we can add message handlers for them. All user-defined messages must have a message ID equal to or greater than WM_USER.

In the sample, a new message WM_BTNPOS is defined in file "MButton.h":

```
#define WM_BTNPOS WM_USER+1000
```

By doing this, WM_BTNPOS becomes a message that can be used in the application. Please note that this message can not be sent to other applications. If we want to send user-defined message among different applications, we need to register that message to the system.

In function MCBitmapButton::OnLButtonUp(...), user defined message WM_BTNPOS is sent to the parent window, with the current mouse position stored in LPARAM parameter:

(Code omitted)

First the default implementation of function OnLButtonUp(...)is called. Then a CWnd type pointer of parent window is obtained by calling function CWnd::GetParent(). Class CWnd has several member functions that can be used to send Windows? message, the most commonly used ones are CWnd:: SendMessage(...) and CWnd::PostMessage(...). The difference between the two functions is that after sending out the message, CWnd::SendMessage(...) does not return until the message has been processed by the target window, and CWnd::PostMessage(...) returns immediately after the message has been sent out. In the sample, function CWnd::PostMessage(...) is used to send WM_BTNPOS message.

All messages in Windows have two parameters, WPARAM and LPARAM. For Win32 applications, both WPARAM and LPARAM are 32-bit integers. They can be used to send additional information.

In MFC, usually message parameters are passed as arguments to message handlers, so they are rarely noticed. For example, for message WM_LBUTTONDOWN, its WPARAM parameter is used to indicate if any of CTRL, ALT, or SHIFT key is held down when the mouse button is up. In the message handler, this information is mapped to the first parameter of CWnd::OnLButtonUp(...). Again, its LPARAM parameter contains the information of current mouse position, which is mapped to the second parameter of CWnd::OnLButtonUp(...). Both CWnd::SendMessage(...) and CWnd::PostMessage(...) have three parameters, the first of which specifies message ID, and the rest two are WPARAM and LPARAM parameters. If we don't want to send additional message, we can pass 0 to both of them.

In the sample, we need to use both parameters: the parent window needs to know the control ID of the button; also, it needs to know the current mouse position.

The button's ID can be retrieved by calling function CWnd::GetDlgCtrlID(), it will be sent through WPARAM parameter to the button's parent. The x and y coordinates of mouse cursor can be combined together to form an LPARAM parameter by using MAKELPARAM macro. Here, macro MAKELPARAM can combine two 16-bit numbers to form a 32-bit message. If we provide two 32-bit numbers, only the lower 16 bits will be used (Of course, screen coordinates won't use more than 16 bits).

The message is received and processed in class CBtnDlg. In MFC, general message can be mapped to a member function by using ON_MESSAGE macro. This type of message handler has two parameters, one for receiving WPARAM information and the other for receiving LPARAM information. Also, it must return a LONG type value.

The following code fragment shows how member function OnBtnPos(...) is declared in class CBtnDlg (It will be used to receive WM_BTNPOS message):

```
class CBtnDlg : public CDialog

{

……

protected:

……

afx_msg LONG OnBtnPos(UINT, LONG);

DECLARE_MESSAGE_MAP()

};
```

In the implementation file, ON_MESSAGE macro is added as follows:

```
BEGIN_MESSAGE_MAP(CBtnDlg, CDialog)

……
```

```
ON_MESSAGE(WM_BTNPOS, OnBtnPos)
END_MESSAGE_MAP()
```

The control ID and the mouse information can be extracted within the message handler as follows:

(Code omitted)

Sample

Sample 4.6\Btn has a four-arrow bitmap button. First a button resource is added to the dialog template, whose ID is IDC_PLAY_POS and caption text is "PLAYPOS" (bitmaps will be loaded through automatic method). Two new bitmap resources "PLAYPOSU" and "PLAYPOSD" are also added to the application, which will be used to draw button's "up" and "down" states.

We need to know the sizes and the positions of four arrows within the bitmap button so we can judge if the mouse cursor is over any of the arrows. Within class CBtnDlg, a CRect type array with size of 4 is declared for this purpose. Their values are initialized in function CBtnDlg::OnInitDialog(). Also an MCBitmapButton type variable m_btnPlayPos is declared to implement this new button:

```
class CBtnDlg : public CDialog
{
......
protected:
......
MCBitmapButton m_btnPlayPos;
......
CRect m_rectBtnPos[4];
......
};
```

The following portion of function CBtnDlg::OnInitDialog() shows how array m_rectBtnPos is initialized:

(Code omitted)

Here, we call function CWnd::GetClientRect() to retrieve the button size. We need to calculate the sizes and positions of the arrows after bitmaps have been loaded. This is because a button will be resized according to the bitmap size after it is initialized.

Function CBtnDlg::OnBtnPos(...) is implemented just for the purpose of demonstration: if any of the four arrows is pressed, a message box will pop up displaying a different message:

(Code omitted)

The application is now ready for compile. Based on this method, we can implement bitmap buttons with more complicated functionality.

4.7 Mouse Sensitive Button

Setting Capture

In this section, we are going to create a very special button. It is not a push button, nor a check box or radio button. The button has two states: normal and highlighted. Generally, the button will stay in normal state. When the mouse cursor is within the button's rectangle (with no mouse button being clicked), the button will become highlighted, and if the mouse moves out of the rectangle, the button will resume to normal state.

To implement this type of button, we need to trap mouse messages and implement handlers. One of the messages we need to handle is WM_MOUSEMOVE, which will be sent to a window if the mouse cursor is moving over the button. We need to respond to this message and set button's state to "highlighted" when the mouse cursor first comes into the button's rectangle. From now on, we need to keep an eye on the mouse's movement, if the mouse moves out of the rectangle, we need to resume button's normal state.

However, since message WM_MOUSEMOVE will only be sent to a window when the mouse cursor is within it, it is difficult to be informed of the event that mouse has just left the button's window. This is because once the mouse leaves, the button will not be able to receive WM_MOUSEMOVE message anymore.

To help us solve this type of problems, Windows? provides a technique that can be used to track mouse's activities after it leaves a window. This technique is called Capture. By using this method, we could capture all the mouse-related messages to one specific window, no matter where the mouse is.

We can call function CWnd::SetCapture() to set capture for a window. The capture can also be released by calling function ::ReleaseCapture(), which is a Win32 API function. Besides using this function, the capture can also be removed by the operating system under certain conditions. If this happens, the window that is losing capture will receive a WM_CAPTURECHANGED message.

New Class

Sample 4.7\Btn demonstrates how to implement "mouse sensitive button". In the application, a new class MCSenButton is created for this purpose, and is defined as follows:

(Code omitted)

The class contains only a constructor, two message handlers, and a Boolean type variable m_bCheck. Variable m_bCheck will be used to indicate the button's current state: it is TRUE if the button is currently highlighted, and is FALSE if the button is in the normal state. Within the constructor, this variable is initialized to FALSE:

```
MCSenButton::MCSenButton() : MCBitmapButton()

{

m_bCheck=FALSE;

}
```

Functions MCSenButton::OnMouseMove(...) and MCSenButton::OnCaptureChanged(...) are message handlers for WM_MOUSEMOVE and WM_CAPTURECHANGED respectively. Like all other types of messages, their message mapping macros should be added in the implementation file:

```
IMPLEMENT_DYNAMIC(MCSenButton, MCBitmapButton)

BEGIN_MESSAGE_MAP(MCSenButton, MCBitmapButton)

ON_WM_MOUSEMOVE()
```

```
ON_WM_CAPTURECHANGED()

END_MESSAGE_MAP()
```

Here, ON_WM_MOUSEMOVE and ON_WM_CAPTURECHANGED are message mapping macros defined by MFC.

The following two functions handle above two messages:

(Code omitted)

In function MCSenButton::OnMouseMove(...), if m_bCheck is FALSE, it means the button is in the normal state. In this case, we need to set mouse capture, change the button to "highlighted" state, and redraw the button. If the button is currently highlighted, we need to check the current position of mouse cursor, if it has moved out of the button window, we should resume button's normal state, and redraw the button. Here, CButton::SetState(...) is used to set the button to different states (it will cause the bitmap button to use the corresponding bitmap), and CWnd::Invalidate() is used to cause the button to be redrawn.

In function MCSenButton::OnCaptureChanged(...), we need to change m_bCheck back to FALSE, and resume button's normal state.

Implementation

In the sample, four new buttons are added to the application. The IDs of these new buttons are IDC_MOUSE_SEN_1, IDC_MOUSE_SEN_2, IDC_MOUSE_SEN_3 and IDC_MOUSE_SEN_4 respectively. An MCSenButton type array m_btnBmp (The array size is 4) is declared in class CBtnDlg and initialized in function CBtnDlg::OnInitDialog() as follows:

(Code omitted)

Here, we use subclass instead of automatic method to load bitmaps. Also, we use IDB_BITMAP_BTNUNCHECK and IDB_BITMAP_BTNCHECK to implement button's normal and highlighted states respectively.

Because mouse related messages are handled within the member functions of class MCSenButton, once we declare variables within it, the buttons will automatically become mouse sensitive. There is no need for us to write extra code for handling mouse messages outside the class.

Summary

1) We can use class CBitmapButton to implement bitmap buttons. To use this class, we need to prepare 1 to 4 bitmap resources indicating button's different states, then use class CBitmapButton to declare variables, and call either CBitmapButton::AutoLoad(...) or CBitmapButton::LoadBitmaps(...) to associate the bitmap resources with the buttons.

2) To use function CBitmapButton::AutoLoad(...), the bitmap resources must have string IDs, and must be created by suffixing 'U', 'D', 'F' or 'X' to the button's caption text.

3) Buttons, check boxes and radio buttons implemented by class CButton can display user-provided bitmaps by calling function CButton::LoadBitmap(...). With this method, the button could be associated with only one image at any time. Also, its focused state will be indicated by drawing a dash-bordered rectangle over button's face.

4) We can call function CBitmapButton::LoadBitmaps(...) at any time to change the associated bitmaps. This provides us a way of implementing check box and radio button using push button.

5) Irregular shape button can be implemented by drawing images with transparency. We can prepare a normal image and a black-and-white mask image. When drawing the button, only the unmasked region of the normal image should be output to the target device.

6) A button can handle mouse-related messages. If we want to know the mouse position when a button is being pressed, we can trap WM_LBUTTONUP message.

7) We can implement mouse sensitive button by handling WM_MOUSEMOVE message and setting window capture.

BACK TO INDEX

Chapter 5 Common Controls

In this chapter we will discuss some common controls that can be included in a dialog box. These controls include spin control, progress bar control, slider control, tree control, tab control, animate control and combo box. These controls can all be included in a dialog template as resources. Besides, all the controls have corresponding MFC classes that can be used to implement them.

In Developer Studio, Class Wizard has some features that can be used to add member variables and message handlers for the common controls. This simplifies the procedure of writing source code.

5.1 Spin Control

Spin control is a rectangular button with two arrows pointing to opposite directions (either vertically or horizontally), it is one of the most commonly used controls in a dialog box. Usually a spin is used together with another control, in most cases this control is an edit box (Though not common, this control can also be a button or a static control). By clicking on one of the arrows, the contents in the accompanying control will change accordingly indicating current position of the spin control.

The control used together with the spin control is called spin's Buddy Control. In MFC, it is very easy to use spin control along with edit box, they are specially designed to cooperate together.

Using Spin Control with Edit Box

By default, a spin control should be associated with an edit box. Usually this type of edit box contains a number indicating the current position of spin. If we make no modification, the range of this number will be from 0 to 100. If the spin's orientation is vertical, pressing the downward arrow will cause the number to increment. If the spin's orientation is horizontal, pressing the leftward arrow will have the same effect.

When adding a spin control resource, we must set several styles in order to make it work correctly. In the property page whose caption is "Spin properties", by clicking "Styles" tab, we will see all the customizable styles (Figure 5-1). Here, style "Auto buddy" will allow spin's buddy to be automatically selected (By enabling this style, we do not need to set spin's buddy within the program). If we check this selection, the window prior to the spin control in the Z order will be used as the spin's buddy window.

(Figure 5-1 omitted)

To let buddy window be automatically selected, we must first add resource for the buddy control then resource for the spin control. For example, if we want to use an edit box together with a spin control, we can add edit box resource first, then add spin control next. We can check controls' Z order by executing command Layout | Tab order (or pressing CTRL+D keys). The Z-order of the controls can be reordered by clicking them one by one according to the new sequence.

In the left-bottom corner of "Spin properties" property sheet, there is a combo box labeled "Alignment". This allows us to specify how the spin will be attached to its buddy window when being displayed. If we select "Unattatched" style, the spin and the buddy control will be separated. Usually we select either "Left" or "Right" style to attach the spin to the left or right side of the buddy control. In this case, the size and position of the spin control in the dialog template has no effect on its real size and position in the runtime, its layout will be decided according to the size and position of the buddy control.

Also, there is a "Set buddy integer" check box. If this style is set, the spin will automatically send out message to its buddy control (must be an edit box) and cause it to display a series of integers when the spin's position is changed. By default, the integer contained in the edit box will increment or decrement with a step of 1. If we want to customize this (For example, if we want to change the step or want to display floating point numbers), we should uncheck this style and set the buddy's text within the program.

Sample 5.1-1\CCtl demonstrates how to use spin control with edit control and set buddy automatically. The sample is a standard dialog based application generated by Application Wizard, with all default settings. The resource ID of the main dialog template is IDD_CCTL_DIALOG, which contains two spin controls and two edit boxes. Both spins have "Auto buddy" and "Set buddy integer" styles. Also, their alignment styles are set to "Right" (Figure 5-2).

(Figure 5-2 omitted)

Without adding a single line of code, we can compile the project and execute it. The spin controls and the edit controls will work together to let us select integers (Figure 5-3).

(Figure 5-3 omitted)

In MFC, spin control is implemented by class CSpinButtonCtrl. We need to call various member functions of this class in order to customize the properties of the spin control. In sample 5.1-2\CCtl, the control's buddy is set by calling function CSpinButtonCtrl::SetBuddy(...) instead of using automatic method. The best place to set a spin's buddy is in the dialog box's initialization stage. This corresponds to calling function CDialog::OnInitDialog().

Sample 5.1-2\CCtl is based on sample 5.1-1\CCtl. Here, style "Auto buddy" is removed for two spin controls. Also, some changes are nade to set the spin buddies manually.

There are two ways of accessing a specific spin: we can use a spin's ID to call function CWnd::GetDlgItem(...), which will return CWnd type pointer to the spin control; or we can add a CSpinButtonCtrl type variable for the spin control (through using Class Wizard). The following code fragment shows how the buddy of the two spin controls are set using the first method:

(Code omitted)

Since CWnd::GetDigItem(...) returns a CWnd type pointer, we need to first cast it to CSpinButtonCtrl type pointer in order to call any member function of class CSpinButtonCtrl. The only parameter that needs to be passed to function CSpinButtonCtrl::SetBuddy(...) is a CWnd type pointer to the buddy control, which can also be obtained by calling function CWnd::GetDlgItem(...).

Spin controls implemented in sample 5.1-2\CCtl behaves exactly the same with those implemented in sample 5.1-1\CCtl.

5.2 Customizing the Properties of Spin Control

We can customize a spin control's properties in function CDialog::OnInitDialog(). The following three functions are the most commonly used ones for doing customization:

(Table omitted)

Sample 5.2\CCtl is based on sample 5.1-1\CCtl. In this sample, the vertical spin is customized to display hexadecimal integers, whose range is set from 0x0 to 0xC8 (0 to 200), and its initial position is set to 0x64 (100). The horizontal spin still displays decimal integers, its range is from 50 to 0, and the initial position is 25. The following portion of function CCCtlDlg::OnInitDialog() shows the newly added code:

(Code omitted)

5.3 Displaying Text Strings in the Buddy Window

Sometimes we want the buddy to display text strings rather than numerical numbers. For example, we may prefer the text displayed in the buddy window to be "One", "Two", "Three"... rather than "1", "2", "3".... To customize this style, we could not use "Set buddy integer" style anymore. Instead, we need to write our own message handlers and set the buddy control's text by ourselves.

When the position of a spin has changed, the parent window of the spin control will receive a UDN_DELTAPOS message. From this message, we can get the current position of the spin control, along with the proposed change to the current position. Based on this information, we can decide what we should display in the buddy control window.

Sample 5.3\CCtl demonstrates how to display text strings in a buddy window. It is based on sample 5.2\CCtl, with a new spin control IDC_SPIN_STR and an edit box IDC_EDIT_STR added to the application. The edit control will display text strings "Zero", "One", "Two",..., "Nine" instead of integers. The buddy of spin IDC_SPIN_STR is set automatically.

The UDN_DELTAPOS message handler can be added through following steps: 1) Invoke Class Wizard, click "Messages Maps" tab. 2) Select "CCCtlDlg" class from "Class name" window, then highlight "IDC_SPIN_STR" in "Object IDs" window. 3) There will be two messages contained in "Messages" window, we need to highlight "UDN_DELTAPOS" and press "Add function" button. The newly added function will look like follows:

```
void CCCtlDlg::OnDeltaposSpinStr(NMHDR* pNMHDR, LRESULT* pResult)

{

NM_UPDOWN* pNMUpDown = (NM_UPDOWN*)pNMHDR;

*pResult = 0;

}
```

The first parameter here is a NMHDR type pointer. This is a structure that contains Windows(notification messages. A notification message is sent to the parent window of a common control to notify the changes on that control. It is used to handle events such as mouse left button clicking, left button double clicking, mouse right button clicking, and right button double clicking performed on a common control. Many types of common controls use this message to notify the parent window. For spin control, after receiving this message, we need to cast the pointer type from NMHDR to NM_UPDOWN. Here structure MN_UPDOWN is defined as follows:

```
typedef struct _NM_UPDOWN { nmud

NMHDR hdr; // notification message header

int iPos; // current position

int iDelta; // proposed change in position

} NM_UPDOWNW;
```

In the structure, member iPos specifies the current position of the spin control, and iDelta indicates the proposed change on spin's position. We can calculate the new position of the spin control by adding up these two members.

The following function shows how the buddy's text is set after receiving the message:

(Code omitted)

The buddy's text is set by calling function CWnd::SetWindowText(...). Here variable szNumber is a two- dimensional character array which stores strings "Zero", "One", "Two"..."Nine". First we calculate the current position of the spin control and store the result in an integer type variable nNewPos. Then we use it as an index to table szNumber, find the appropriate string, and use it to set the text of the edit control.

In dialog box's initialization stage, we need to set the range and position of the spin control. Since the edit box will display nothing by default, we also need to set its initial text:

(Code omitted)

With the above implementation, the spin's buddy control will display text instead of numbers.

5.4 Bitmap Button Buddy

String text is not the only appearance a buddy control can have. We can also implement a buddy that displays bitmaps. Because bitmap button can be easily implemented to display images, we can use it to implement spin's buddy control rather than using edit box.

Sample 5.4\CCtl demonstrates how to implement bitmap button buddy. It is based on sample 5.3\CCtl, with a new spin control IDC_SPIN_BMP and a new bitmap button IDC_BUTTON_BMP added to the application.

The procedure of creating a bitmap button buddy is almost the same with creating an edit box buddy. The only difference here is that instead of creating an edit box resource, we need to add a button resource, and set its "Owner draw" style.

In the sample, four bitmaps are prepared to implement the bitmap button. All of them have integer resource IDs, which are listed as follows: IDB_BITMAP_SMILE_1, IDB_BITMAP_SMILE_2, IDB_BITMAP_SMILE_3, IDB_BITMAP_SMILE_4.

A CBitmapButton type variable is declared in class CCCtlDlg. In the dialog box's initialization stage, functions CWnd::SubclassDlgItem(...), CBitmapButton::LoadBitmaps(...) and CBitmapButton:: SizeToContent() are called to initialize the bitmap button. Also, the range of the spin control is set from 0 to 3, and its initial position is set to 0:

(Code omitted)

The initially selected bitmap is IDC_BITMAP_SMILE_1. We should not load bitmaps for other states ("down", "focused" and "disabled") because the purpose of this button is to display images rather than executing commands. We need to change the currently loaded image upon receiving UDN_DELTAPOS notification. To change button's associated bitmap, in the sample application, a UDN_DELTAPOS message handler is added for IDC_SPIN_BMP, which is implemented as follows:

(Code omitted)

Although we say that the bitmap button is the buddy of the spin control, in the above implementation we see that they do not have special relationship. A spin control needs a buddy only in the case when we want the text in the buddy window to be updated automatically. If we implement this in UDN_DELTAPOS message handler, the buddy loses its meaning because we can actually set text for any control. Although this is true, here we still treat the bitmap button as the buddy of spin control because the bitmap button is under the control of the spin.

5.5 Slider

A slider is a control that allows the user to select a value from pre-defined range using mouse or keyboard. A slider can be customized to have many different styles: we can put tick marks on it, set its starting and ending ranges, make the tick marks distributed linearly or non-linearly. Besides these attributes, we can also set the line size and page size of a slider, which decide the minimum distance the slider moves when the user clicks mouse on slider's rail or hit arrow keys of the keyboard.

Including Slider Control in the Application

Sample 5.5\CCtl demonstrates how to use the slider control. It is a standard dialog based application generated by the Application Wizard. In the dialog box, three different sliders are implemented, they are used to show how to set tick marks, page size, line size, and implement other customizations.

The tick marks can be added to the slider automatically. When we add a slider resource to the dialog template, we can find two check boxes, "Tick mark" and "Auto ticks", in the property page that lets us customize slider's properties (Figure 5-4). If we check the former check box, the slider will be able to have tick marks, if we check the latter, tick marks will be added automatically.

(Figure 5-4 omitted)

In MFC, slider can be implemented through using class CSliderCtrl. We need to call its member functions in order to customize the slider.

To let the tick marks be set automatically, besides setting "Tick mark" and "Auto ticks" styles, we must also specify slider's range. A slider's range can be set by calling either function CSliderCtrl:: SetRange(...) alone or CSliderCtrl::SetRangeMin(...) together with CSliderCtrl::SetRangeMax(...)in the dialog box's initialization stage. The format of the above three functions are listed as follows:

```
void CSliderCtrl::SetRange(int nMin, int nMax, BOOL bRedraw = FALSE);

void CSliderCtrl::SetRangeMax (int nMax, BOOL bRedraw = FALSE);

void CSliderCtrl::SetRangeMin(int nMin, BOOL bRedraw = FALSE);
```

By default, the distance between two neighboring tick marks is 1. To change this, we may call function CSliderCtrl::SetTicFreq(...) to set the frequency of the tic marks. If the slider does not have "Auto ticks" style, we must call function CSliderCtrl::SetTic(...) to set tick marks for the slider. Because this function allows us to specify the position of a tic mark, we can use it to set non-linearly distributed tic marks.

Two other properties that can be modified are slider's page size and line size.

Here, page size represents the distance the slider will move after the user clicks mouse on its rail. The line size represents the distance the slider will move if the user hits left arrow or right arrow key when the slider has the current focus (Figure 5-5). Two member functions of class CSliderCtrl can be used to set the above two sizes: CSliderCtrl::SetPageSize(...) and CSliderCtrl::SetLineSize(...). The default page size is 1/5 of the total slider range and the default line size is 1.

In sample 5.5\CCtl, there are three sliders, whose IDs are IDC_SLIDER_AUTOTICK, IDC_SLIDER_TICK and IDC_SLIDER_SEL respectively. Here, slider IDC_SLIDER_AUTOTICK has "Tick marks" and "Auto ticks" styles, slider IDC_SLIDER_TICK has only one "Tick marks" style, and slider IDC_SLIDER_SEL has "Tick marks", "Auto ticks", and "Enable selection" styles.

Three sliders are initialized in function CCCtlDlg::OnInitDialog(). The following portion of this function sets the range, tick marks, page size and line size for each slider:

(Code omitted)

Since slider IDC_SLIDER_AUTOTICK has "Auto ticks" style, we don't need to set the tick marks. In the sample, the range of this slider is set from 0 to 10, and the tick mark frequency is set to 2. The tick marks will appear at 0, 2, 4, 6... 10. Also, since no page size and line size are specified here, they will be set to the default values (2 and 1). For IDC_SLIDER_TICK, its range is set from 0 to 50, and function CSliderCtrl::SetTic(...) is called to set non-linearly distributed tick marks. Here, a loop is used to set all the tick marks, which will appear at 0, 4, 8, 16.... Slider IDC_SLIDER_SEL also has "Auto ticks" style, its range is set from 50 to 100, page size set to 40 and line size set to 10. This slider also has "Enable selection" style, and function CSliderCtrl::SetSelection(...) is called to draw a selection on the slider's rail. The range of the selection is from 60 to 90.

Handling Slider Related Messages

Another feature we want to add to the application is trapping events generated by the sliders. When a slider moves, we may want to know its current position and make changes to other settings. For example, in a multimedia application, we can use slider to control the volume of speakers.

In Windows(system, there is no special message defined for the slider. Instead, a slider shares same messages with scroll bar. For horizontal sliders, we need to trap WM_HSCROLL message; for vertical sliders, we need to trap WM_VSCROLL message.

Similar to UDN_DELTAPOS message, in a dialog box, message WM_HSCROLL or WM_VSCROLL can be added through using Class Wizard. The default message handler for WM_HSCROLL will look like this:

```
void CCCtlDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)

{

CDialog::OnHScroll(nSBCode, nPos, pScrollBar);

}
```

There are three parameters in this function. The first parameter nSBCode indicates user's scrolling request, which includes left-scroll, right-scroll, left-page-scroll, right-page-scroll, etc. If we want to customize the behavior of a slider, we need to check this parameter. Parameter nPos is used to specify the current slider's position under some situations (it is not valid all the time). The third parameter pScrollBar is a window pointer to slider or scroll bar control. We can use it to check which slider is being changed, then make the corresponding response. The slider's current position can be obtained by calling function CSliderCtrl::GetPos(). In the sample application, since all sliders are horizontal, only WM_HSCROLL message is handled:

(Code omitted)

Function CWnd::GetDlgCtrlID() is called to retrieve the control ID of the slider. We use this ID to call function CWnd::GetDlgItem(...) and get the address of slider control. If the control happens to be one of our sliders, we call function CSliderCtrl::GetPos() to retrieve its current position, and output the slider ID along with its current position to the debug window. In order to see the activities of the sliders, the application must be executed in the debug mode within Developer Studio.

5.6 List Box

List box is a control that contains a list of objects such as file names and strings that can be selected by mouse clicking. When we create a list box, there are many styles that can be customized (Figure 5-6). For example, if we check "Horizontal scroll" style, the list box will automatically implement a horizontal scroll bar if any of its string is too long to be fully displayed in the list window. If we check "Vertical scroll" style, the list box will add a vertical scroll if the vertical size of the list box is not big enough for displaying all items. Some other important styles that a list box can have are "Multi-column", "Sort", "Selection" and "Owner draw".

(Figure 5-6 omitted)

Usually a list box has a single column. If we set "Multi-column" style, the list box can have multiple horizontal columns. Originally the list box will be empty, when we start to add new items, they will be added to the first column (column 0). If the first column is full, instead of creating a vertical scroll bar and continue to add items to this column, the list box will create a new column and begin to fill it. This step will be repeated until all items are filled into the list box. Here, the width of each column is always the same. Because a multiple-column list box will always try to extend horizontally rather than vertically, it is important to let this type of list box have a horizontal scroll bar.

The "Sort" style will be set by default. If we remain this style, all the strings contained in the list box will be alphabetically sorted. For the "Selection" styles, we have several choices. A "Single" style list box allows only one item in the list box to be selected at any time. A "Multiple" style list box allows several items to be selected at the same time. If we enable this style, the user can use SHIFT and CTRL keys together to select several items. Besides these two, there is also an "Extended" style. If we enable it, the items can be selected or deselected through mouse dragging. Finally, list box with "Owner draw" style allows us to implement it so that the list box can contain non-string items. In this case, we need to provide custom list box interface.

Sample 5.6\CCtl demonstrates basic styles of list box. It is a dialog-based application created by Application Wizard. There are three list boxes implemented in the application, whose IDs are IDC_LIST, IDC_LIST_MULCOL and IDC_LIST_DIR respectively. The styles of IDC_LIST are all set to default, it is a single selection, single column, sorted list box with a vertical scroll bar. The styles of IDC_LIST_MULCOL are multiple-column, multiple-selection, it does not support sort. The styles of IDC_LIST are also set to default, except that it supports "extended selection" style. To access these list boxes, three CListCtrl type member variables m_listBox, m_listMCBox and m_listDir are declared in class CCtlDlg through using Class Wizard (Figure 5-7).

(Figure 5-7 omitted)

Unless we initialize the content of these list boxes, they will be empty at the beginning. Like other common controls, initialization procedure of list box is usually implemented in function CDialog:: OnInitDlalog(). To fill a list box with strings, we need to call function CListBox::AddString(...). Strings will be added starting from item 0, 1, 2... and so on (If a list box has a sorted style, the string will be sorted automatically). Besides this function, we can also use function Clistbox::InsertString(...) to insert a new string before certain item instead of adding it to the end of the list. The following code fragment shows how the content of list boxes IDC_LIST and IDC_MULCOL are filled:

(Code omitted)

If we do not specify the column width for a multiple-column list box, the default column width will be used. We may set this width by calling function CListBox::SetColumnWidth(...). In the sample 5.6\CCtl, the column width of IDC_LIST_MULCOL is set 50 (pixels) as follows:

```
BOOL CCtlDlg::OnInitDialog()

{

......

m_listMCBox.AddString("Item 20");

m_listMCBox.SetColumnWidth(50);

......

}
```

All the columns will have the same width. For list box IDC_LIST_DIR, instead of filling each entry with a string, we can let it display a list of directories and file names for the current working directory. This can be implemented by calling function CListBox::Dir(...), which has the following format:

```
int CListBox::Dir(UINT attr, LPCTSTR lpszWildCard);
```

Here, the first parameter specifies file attributes, which can be used to specify what type of files can be added to the list. The following is a list of some attributes that are commonly used:

(Table omitted)

The second parameter is a string, which can be used to set file filter. In the sample, this function is called as follows:

```
BOOL CCtlDlg::OnInitDialog()

{

......

m_listDir.Dir(0x10, "*.*");

return TRUE;
}
```

Here, value 0x10 is passed to the first parameter of function CListBox::Dir(...) to let normal files along with directories be listed, also, we use "*.*" wildcards to allow all types of names to be added to the list.

When testing the sample application, we can use mouse along with SHIFT and CONTROL keys to select items. Also, we can drag mouse over items to test extended selection style.

5.7 Handling List Box Messages

Like other common controls, list box has its own messages that are related to mouse or keyboard activities.

Trapping Double Clicking Message

In the previous sample, it may be helpful to trap mouse double clicking message for list box IDC_LIST_DIR. When the user double clicks a directory, we can change the contents of the list box, fill it with the file and directory names contained under the directory being clicked. This feature is implemented in sample 5.7\CCtl, which is based on sample 5.6\CCtl.

The double clicking message of a list box is LBN_DBLCLK. We can easily add handler for this message through using Class Wizard: after invoking the Class Wizard, by clicking "Message maps" tab and selecting class "CCtlDlg", three IDs of the list boxes will be listed in "Object IDs" window. We can highlight "IDC_LIST_DIR", then select "LBN_DBCLK" message from "Messages" window, and press button "Add function". If we accept the default function name, a new function CtlDlg::OnDblclkListDir() will be added to the application.

Retrieving the Contents of an Item

After receiving the double clicking message, we need to obtain the string contained in the item that was clicked. For a single selection list box, the current selected item can be retrieved by calling function CListBox::GetCurSel() (After being double clicked, the item must become currently selected). This function will return a zero based index indicating which item is currently being selected. Then we can call function CListBox::GetText(...) to retrieve the string contained in the item. Since list IDC_LIST_DIR is a multiple-selection list box, retrieving text from the selected items is a little different. In this case, first function CListBox::GetSelCount() must be called to retrieve the number of items that are currently being selected. According to this value, we must allocate enough buffers for storing indices of the selected items. Then we can call function CListBox::GetSelItems(...) and pass the buffer's address to it for receiving indices of all the selected items. For each index, we can call CListBox::GetText(...) to retrieve its string.

After a double clicking, one and only one item in the list box will be selected. In this case, we can skip the first step because CListBox::GetSelCount() will surely return 1. If a list item represents directory, a pair of square brackets "[]" will be added to the directory name. So we can judge if the item being double clicked contains a file name or a directory name by examining if the string starts and ends with square brackets. If we allow drive names to be displayed, the items containing drive names will be displayed in the format of "[-X-]", where X represents the drive name. In our samples, this situation is not considered).

The following is the implementation of function CCtlDlg::OnDblclkListDir(). This function examines the clicked item. If the item contains a directory name, we need to update the contents of the list box (File and directory names under the directory being clicked will be retrieved and filled into the list box):

(Code omitted)

First function CListCtrl::GetSelItems(...) is called to retrieve the currently selected item. The result is stored to a local variable nIndex. Then text string of the selected item is obtained by calling function CListCtrl::GetText(...). If the string starts with "[", it is a directory, and we extract the directory name by calling function CString::Mid(...). Then contents of the list box are cleared by calling function CListCtrl::ResetContent(). Next, the current working directory is changed by calling function _chdir(...). Finally, the list box is filled with the new directory and file names by calling function CListCtrl::Dir(...). Since function _chdir(...) is not an MFC function, we need to include "direct.h" header file in order to use it.

Message WM_DESTROY

Besides directory changing, another new feature is also implemented in the sample application: we can use mouse to highlight any item contained in the other two list boxes. When the dialog box is closed, a message box will pop up displaying all the items that are currently being selected.

Before a window is destroyed, it will receive a WM_DESTROY message, so we can handle this message to do clean up work. In our case, this is the best place to retrieve the final state of the list boxes. Please note that we can not do this in the destructor of class CCtlDlg, because at that time the dialog box window and all its child window have already been destroyed. If we try to access them, it will cause the application to malfunction.

Message handler WM_DESTROY can be added by using Class Wizard through following steps: 1) Click "Message maps" tab and choose "CCtlDlg" class in window "Class name". 2) Highlight "CCtlDlg" in window "Object IDs". 3) In "Messages" window, find "WM_DESTROY" message and click "Add function" button. After the above steps, a new function CCtlDlg::OnDestroy() will be added to the application.

We will retrieve all the text strings of the selected items for three list boxes and display them in a message box. For list box IDC_LIST_BOX this is easy, because it allows only single selection. We can call function CListBox::GetCurSel() to obtain the index of the selected item and call CListBox::GetText(...) to retrieve the text:

(Code omitted)

Function CListBox::GetCurSel() will return value LB_ERR if nothing is being currently selected or the list box has a multiple-selection style. If there is a selected item, we use CString type variable szStrList to retrieve the text of that item.

For list box IDC_LIST_MULCOL and IDC_LIST_DIR, things become a little complicated become both of them allow multiple-selection. We need to first find out how many items are being selected, then allocate enough buffers for storing the indices of the selected items, and use a loop to retrieve the text of each item. Each time a new string is obtained, it is appended to the end of szStrList. The following code fragment shows how the text of all the selected items is retrieved for list box IDC_LIST_MULCOL:

(Code omitted)

In this function, first the number of selected items is retrieved by calling function CListBox::GetSelCount(), and the retrieved value is saved to variable nSelSize. If the size is not zero, we allocate an integer type array with size of nSelSize. Then by calling function CListBox::GetSelItems(...), we fill this buffer with the indices of selected items. Next, a loop is used to retrieve the text of each item. The procedure of retrieving selected text for list box IDC_LIST_DIR is the same.

5.8 Combo Box

Combo box is another type of common control that allows the user to select one object from a list. While a list box allows multiple selections, a combo box allows only single selection at any time. A combo box is made up of two other controls: an edit box and a list box. There are three types of combo boxes: 1) Simple combo box: the list box is placed below the edit box, and is displayed all the time; the edit box displays the currently selected item in the list box. 2) Drop down combo box: the list box is hidden most of the time; when the user clicks the drop-down arrow button located at the right corner of the edit box, the list box is shown and can be used to select an item. In both 1) and 2), the edit box can be used to input a string. 3) Drop list combo box: it is the same with drop down combo box, except that its edit box cannot be used to input string.

Using a combo box is more or less the same with that of a list box. We must first create combo box resources in the dialog template, set appropriate styles, then in the dialog's initialization stage (in function CDialog::OnInitDialog()), initialize the combo box. We can add message handlers to trap mouse or keyboard related events for combo box. Two most important messages for combo boxes are CBN_CLOSEUP and CBN_SELCHANGE. The first message indicates that the user has clicked the drop-down arrow button, made a selection from the list box, and the drop down list is about to be closed. The second message indicates that the user has selected a new item.

In MFC, combo box is supported by class CComboBox. Like CListBox, class CComboBox has a function CComboBox::AddString(...) which can be used to initialize the contents of its list box. Besides this, we can also initialize the contents of a list box when designing dialog template. In the property sheet whose caption is "Combo Box Properties", by clicking "Data" tab, we will have a multiple-line edit box that can be used to input initial data for combo box. We can use CTRL+RETURN keys to begin a new line (Figure 5-8).

(Figure 5-8 omitted)

Class CComboBox has two functions that allow us to change the contents contained in the list box dynamically: CComboBox::InsertString(...) and CComboBox:: DeleteString(...).

When designing drop-down combo box, we must set its vertical size, otherwise it will be set to the default value zero. In this case, there will be no space for the list box to be dropped down when the user clicks drop down button. To set this size, we can click the drop-down button in the dialog template. After doing this, a resizable tracker will appear. The initial size of a combo box can be adjusted by dragging the tracker's border (Figure 5-9).

(Figure 5-9 omitted)

Implementing Combo Boxes

Sample 5.8\CCtl demonstrates the basics of combo box. It is a standard dialog-based application generated by Application Wizard. Three different combo boxes are implemented in the sample, whose IDs are IDC_COMBO_SIMPLE, IDC_COMBO_DROPDOWN and IDC_COMBO_DROPLIST respectively. For these combo boxes, IDC_COMBO_SIMPLE is a "Simple" type, its items are initialized to

"Item 1", "Item 2"... "Item 4" when designing the dialog template; IDC_COMBO_DROPDOWN is a "Drop down" type, and no initialization is done in the resource; IDC_COMBO_DROPLIST is a "Drop list" type, its contents are also initialized as "Item 1", "Item 2"... "Item 4" like IDC_COMBO_SIMPLE. All other styles are set as default, this will let all three combo boxes have vertical scroll bars automatically, and their items be sorted alphabetically.

Three static text controls IDC_STATIC_SIMPLE, IDC_STATIC_DROPDOWN and IDC_STATIC_DROPLIST are added below each combo box. We will use them to display the current selection of the corresponding combo box dynamically.

Three CComboBox type member variables, m_cbSimple, m_cbDropDown and m_cbDropList, are declared in class CCCtlDlg. They will be used to access the combo boxes This can be implemented through using Class Wizard as follows: 1) Invoke Class Wizard, click "Member variables" tab, select "CCCtlDlg" from window "Class name". 2) Highlight the ID of the combo box (IDC_COMBO_SIMPLE, IDC_COMBO_DROPDOWN or IDC_COMBO_DROPLIST), press "Add variable" button. 3) Select "Control" category and input the variable name.

In function CCCtlDlg::OnInitDialog(), the contents of combo box IDC_COMBO_DROPDOWN are initialized through calling function CComboBox::AddString(...):

(Code omitted)

Handling Messages CBN_CLOSEUP and CBN_SELCHANGE

We need to implement message handlers for CBN_CLOSEUP or CBN_SELCHANGE in order to respond to mouse's events. For combo box IDC_COMBO_DROPDOWN and IDC_COMBO_DROPLIST, we know that the selection is changed if we receive message CBN_CLOSEUP. For combo box IDC_COMBO_SIMPLE, we need to use CBN_SELCHANGE because the list box will not close after a new selection is made.

Message handlers can be easily added through using Class Wizard as follows: 1) Invoke Class Wizard, click "Message maps" tab and select "CCCtlDlg" from window "Class name". 2) Highlight the appropriate combo box ID in window "Object IDs". 3) In window "Messages", highlight the appropriate message (CBN_CLOSEUP for IDC_COMBO_DROPDOWN and IDC_COMBO_DROPLIST, CBN_SELCHANGE for IDC_COMBO_SIMPLE). 3) Click button "Add function" and confirm the member function name.

These functions will be called when the user makes a new selection from the list box of a combo box. We can retrieve the index of the current selection of a combo box by calling function CComboBox:: GetCurSel(), and further retrieve the text of that item by calling function CComboBox::GetLBText(...). At last we can call function CWnd::SetWindowText(...) to display the updated content of the selected item in one of the static text controls. The implementations of three message handlers are almost the same. The following is one of them:

(Code omitted)

First index of the current selection is retrieved and stored in variable nSel. Then we check if the returned value is CB_ERR. This is possible if there is nothing being currently selected. If the returned value is a valid index, we call function CComboBox::GetLBText(...) to retrieve the text string and store it in CString type variable szStr. Finally function CWnd::GetDlgItem(...) is called to obtain the pointer to the static text window, and CWnd::SetWindowText(...) is called to update its contents.

5.9 Trapping RETURN key strokes for the Combo Box

Problem & Workaround

One feature we may want to add to the combo boxes is to let the user dynamically add new items through using their edit boxes. We can let the user input a string into the edit box of a drop-down or simple combo box, then hit the RETURN key to add the input to list item. However, in a dialog box, the RETURN key (also the ESC key) is used to exit the application by default. Even if we add message handler for combo box to trap RETURN keystrokes, it still can not receive this message because after the message reaches the dialog box, the application will exit. The message has no chance to be further routed to the child windows of a dialog box.

If we want to process RETURN keystroke events, we need to intercept the message before it is processed by the dialog box. In MFC, there is a function CWnd::PreTranslateMessage(...) that can be overridden for this purpose. This function will be called just before a message is about to be processed by the destination window. Since CDialog is derived from CWnd, we can trap any message sent to the dialog box if we override the above function. This function has the following format:

```
BOOL CWnd::PreTranslateMessage(MSG *pMsg);
```

Its only parameter is a pointer to MSG type object:

```
typedef struct tagMSG { // msg

HWND hwnd;
UINT message;

WPARAM wParam;

LPARAM lParam;

DWORD time;

POINT pt;

} MSG;
```

From this structure, we know which window is going to receive the message (from member hwnd), what kind of message it is (from member message). Also, we can obtain the message parameters from members wParam and lParam. If the message is not the one we want to intercept, we can just forward the message to its original destination by calling the base class version of this function.

Function CWnd::PreTranslateMessage(...)

Sample 5.9\CCtl demonstrates how to trap RETURN keystrokes for combo box. It is based on sample 5.8\CCtl. First, function PreTranslateMessage(...) is overridden. This function can be added by using Class Wizard through following steps: 1) Open Class Wizard, click "Message Maps" tab, select "CCCtlDlg" from "Class name" window. 2) Highlight "CCCtlDlg" in window "Object IDs". 3) Locate and highlight "PreTranslateMessage" in window "Messages". 4) Press "Add function" button.

The default member function looks like the following:

```
BOOL CCCtlDlg::PreTranslateMessage(MSG *pMsg)

{

return CDialog::PreTranslateMessage(pMsg);

}
```

If we do not want to process the message, we need to call function CDialog:: PreTranslateMessage(...) to let the dialog box process it as usual. Otherwise we need to return a TRUE value to give the operating system an impression that the message has been processed properly.

In the overridden function, first we need to check if the message is

WM_KEYDOWN and the key being pressed is RETURN:

(Code omitted)

Message WM_KEYDOWN is a standard Windows(message for non-system key strokes, and VK_RETURN is a standard virtue key code defined for RETUN key (For a list of virtual key codes, see appendix A). Some local variables are declared at the beginning. They will be used throughout this function.

Accessing the Edit Box of a Combo Box

We need to find out which combo box has the current focus in order to decide if we should process this message. If the item that has the current focus is either IDC_COMBO_DROPDOWN or IDC_COMBO_SIMPLE, we will update the corresponding list items.

In Windows(operating system, windows are managed through using handles. Like menu and bitmap resources, a window handle is also a number which could be used to identify a window. Each window's handle has a different value. As a programmer, we do not need to know the exact value of the handle, however, we can use handle to access or identify a window.

In MFC, there is a function CWnd::GetFocus(), which can be used to obtain a pointer to the child window that has the current focus. From this pointer, we can obtain that window's handle. Then we can compare the handle obtained from function CWnd::GetFocus() with the handles of combo boxes. If there is a hit, we could update the content of that combo box.

Unfortunately, since a combo box is made up of two controls: an edit box and a list box, if we are trying to input characters into the combo box, it is the edit box that has the current focus. Thus if we call CWnd::GetFocus() to obtain handle of the window that has the current focus, we will actually get the handle of the edit box. The edit box is the child window of the combo box window, and it has a different handle with its parent. So comparing the handle of the edit box with the handles of the combo boxes will never result in any hit. The correct step would be: for each combo box, obtaining the handle of its edit box, then comparing it with the handle of the focused window. This will eventually result in a hit.

Class CWnd has a member function that can be used to find a window's child windows:

```
CWnd *CWnd::GetWindow(UINT nCmd);
```

Here nCmd specifies what kind of window is being looked for. To enumerate all the child windows, we need to call this function using GW_CHILD flag to find the first child window, then, use GW_HWNDNEXT to call the same function repeatedly until it returns a NULL value. This will enumerate all the sibling windows of the first child window.

There are still problems here: function CWnd::GetWindow(...) returns a CWnd type pointer, we can not obtain further information about that window (i.e. is it an edit box or a list box?). Since a combo box has two child windows, although we can access both of them with the above-mentioned method, we do not know which one is the edit box.

In Windows(, before a new type of window is created, it must register a special class name to the system. Every window has its own class name, which could be used to tell the window's type. In the case of combo box, its edit box's class name is "Edit" and its list box's class name is "ComboLBox". Please note that this class name has nothing to do with MFC classes. It is used by the operating system to identify the window types rather than a programming implementation.

In MFC, the procedure of creating windows is handled automatically, so we never bother to register class names for the windows being created, therefore, we seldom need to know the class names of our windows.

A window's class name can be retrieved from its handle by calling an API function:

```
int ::GetClassName(HWND hWnd, LPTSTR lpClassName, int nMaxCount);
```

The first parameter hWnd is the handle of window whose class name is being retrieved; the second parameter lpClassName is the pointer to a buffer where the class name string can be put; the third parameter nMaxCount specifies the length of this buffer.

We can access the first child window of the combo box, see if its class name is "Edit". If not, the other child window must be the edit box. This is because a combo box has only two child windows.

A window's handle can be obtained by calling function CWnd::GetSafeHwnd(). If the window that has the current focus is the edit box of a combo box when RETURN is pressed, we need to notify the parent window about this event. In the sample, a user defined message is used to implement this notification:

```
#define WM_COMBO_RETURN WM_USER+1000
```

The following portion of function CCCtlDlg::PreTranslateMessage(...) shows how to retrieve the handles of the edit boxes and compare them with the handle of the focused window:

(Code omitted)

First the handle of currently focused window is stored in variable hwndFocus. If it is a valid window handle, we use m_cbDropDown to get the first child window of IDC_COMBO_DROPDOWN. Then this child window's class name is retrieved by calling function ::GetClassName(...). If the class name is "Edit", we compare its handle with the focused window handle. Otherwise we need to get the handle of the other child window before doing the comparison. This will assure that the handle being compared is the handle of the edit box. If the edit box has the current focus, we post the user defined message WM_COMBO_RETURN, whose WPARAM parameter is assigned the ID of combo box. Finally a TRUE value is returned to prevent the dialog box from further processing this message.

Message WM_COMBO_RETURN is processed in class CCCtlDlg. The member function used to trap this message is CCCtlDlg::OnComboReturn(...). The following code fragment shows how this function is declared and message mapping is implemented:

Function declaration:

```
class CCCtlDlg : public CDialog

{

……

protected:

……

afx_msg LONG OnComboReturn(UINT, LONG);

DECLARE_MESSAGE_MAP()

};
```

Message mapping macros:

```
BEGIN_MESSAGE_MAP(CCCtlDlg, CDialog)

……

ON_MESSAGE(WM_COMBO_RETURN, OnComboReturn)
```

```
END_MESSAGE_MAP()
```

Function implementation:

(Code omitted)

In this message handler, we first obtain a pointer to the combo box using the ID passed through WPARAP message parameter. Then we use above-mentioned method to get the pointer to the edit box (a child window of combo box), and assign it to variable ptrEdit. Then we use this pointer to call function CWnd:: GetWindowText(...) to retrieve the text contained in the edit box window. If the edit box is not empty (this is checked by calling function CString::IsEmpty()), we select all the text in the edit box by calling function CEdit::SetSel(...), which has the following format:

```
void CEdit::SetSel(int nStartChar, int nEndChar, BOOL bNoScroll = FALSE);
```

The first two parameters of this function allow us to specify a range indicating which characters are to be selected. If we pass 0 to nStartChar and -1 to nEndChar, all the characters in the edit box will be selected. Then we use a loop to check if the text contained in the edit box is identical to any item string in the list box. In case there is no hit, we will add this string to the list box by calling function CComboBox::AddString(...). Finally, a TRUE value is returned before this function exits.

Using this method, we can also trap other keystrokes such as DELETE, ESC to the combo box. This will make the application easier to use.

5.10 Implementing Subclass for the Edit Box of a Combo Box

Under certain conditions we may want to put restrictions on the contents of the list items. For example, sometimes we may want the combo box to hold only numerical characters ('0'-'9'), and sometimes we may expect it to hold only alphabetical characters ('a'-'z', 'A'-'Z'). In these cases, we may want to customize the properties of the edit box so that only a special set of characters can be accepted. If we are creating an edit box resource in dialog template, this can be easily achieved by setting its customizable styles. But for the edit box of a combo box, we can not customize its styles before it is created, so the edit box contained in a combo box will have only the default styles.

To customize the behavior of the edit box in a combo box, we need to use "subclass" technique. We can design our own class to intercept and process the messages sent to the edit box. Sample 5.10\CCtl demonstrates how to customize the edit box that belongs to a combo box. It is based on sample 5.9\CCtl, with two combo boxes customized as follows: combo box IDC_COMBO_SIMPLE allows only numerical characters to be input into the edit box; combo box IDC_COMBO_DROPDOWN accepts only alphabetic characters.

Designing New Classes

Before implementing subclass, we need to design two classes that have the above-mentioned new properties. In the sample, MCNumEdit and MCCharEdit are added for this purpose. Both of them are derived from class CEdit. In Developer Studio, a new class can be easily added by using Class Wizard through following steps: 1) Execute command Insert | New Class to invoke the Class Wizard. 2) Input the class name, select the header file and implemantation file name. 3) Select base class name.

To customize the input attributes of an edit box, we need to handle WM_CHAR message, which is used to indicate that a character is being input into the control. This message handler can also be added through using Class Wizard after it is invoked as follows: 1) Click "Message Map" tab, select "MCNumEdit" or "MCCharEdit" class name in window "Class name". 3) Highlight "MCNumEdit" or "MCCharEdit" in window "Object IDs". 4) Locate and highlight "WM_CHAR" in window "Messages". 5) Click "Add function" button.

The following is one of the two functions generated by Class Wizard:

```
void MCNumEdit::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)

{

CEdit::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags);

}
```

This function has three parameters. The first parameter nChar indicates the value of the key, which provides us with the information of which key being pressed. The Second parameter indicates the repeat count, and the third parameter holds extra information about the keystrokes.

If we want the keystroke to be processed normally, we need to call the base class version of this function. If we do not call this function, the input will have no effect on the edit box. The following code fragment shows two message handlers implemented in the sample:

(Code omitted)

Class MCNumEdit accepts characters '0'-'9' and backspace key, class MCCharEdit accepts characters 'A'-'Z', 'a'-'z' and backspace key.

Implementing Subclass

To use the two classes, we need to include their header files and use them to declare two new variables in class CCCtlDlg:

```
......
#include "CharEdit.h"
#include "NumEdit.h"
......
class CCCtlDlg : public CDialog
{
......
protected:
......
MCCharEdit m_editChar;
MCNumEdit m_editNum;
......
};
```

In the dialog box's initialization stage, we need to implement subclass and change the default behavior of the edit boxes. Remember in the previous chapter, function CWnd::SubclassDlgItem(...) is used to implement subclass for an item contained in a dialog box. Although the edit box within a combo box is a indirect child window of the dialog box, it is not created from dialog template. So here we must call function CWnd::SubclassWindow(...) to implement subclass. The following is the format of this function:

```
BOOL CWnd::SubclassWindow(HWND hWnd);
```

Here, parameter hWnd is the handle of the window whose behavior is to be customized. From sample 5.9\CCtl, we know how to obtain the handle of the edit box that belongs to a combo box. The following is the procedure of implementing subclass for IDC_COMBO_DROPDOWN combo box:

(Code omitted)

With the above implementation, the combo box is able to filter out the characters we do not want.

5.11 Owner Draw List Box and Combo Box

Like menu, list box and combo box do not have to bear plain text interface all the time. Sometimes we can customize them to display images. In the previous samples, when implementing a list box or a combo box, we always select "No" for the "Owner draw" style. Actually, the "Owner draw" style can be set to other two selections: "Fixed" and "Variable". For a "fixed" type owner-draw list box or combo box, each item contained in the list box must have a same height. For a "variable" type of owner draw list box or combo box, this height can be variable. Like the menu, the owner-draw list box or combo box are drawn by their owner. The owner will receive message WM_MEASUREITEM and WM_DRAWITEM when the list box or the combo box needs to be updated. For "fixed" type owner draw list box or combo box, WM_MEASUREITEM is sent when it is first created and the returned size will be used for all items. For "variable" type owner-draw list box or combo box, this message is sent for each item separately. Message WM_DRAWITEM will be sent when the interface of list box or combo box needs to be updated.

Owner-Draw Styles

Sample 5.11\CCtl demonstrates owner-draw list box and combo box. It is a dialog based application generated by Application Wizard. There are only two common controls contained in the dialog box: a list box IDC_LIST and a combo box IDC_COMBO. The list box supports "Fixed" owner-draw style, and the combo box supports "Variable" owner-draw style. The "Sort" style is not applicable to an owner-draw list box or combo-box, because their items will not contain characters.

Preparing Bitmaps

Six bitmap resources are added to the application for list box and combo box drawing. Among them, IDB_BITMAP_SMILE_1, IDB_BITMAP_SMILE_2, IDB_BITMAP_SMILE_3 and IDB_BITMAP_SMILE_4 have the same dimension, they will be used for implementing owner-draw list box. Bitmaps IDB_BITMAP_BUTTON_SEL and IDB_BITMAP_BUTTON_UNSEL have a different size with the above four bitmaps, they will be used together with IDB_BITMAP_BIG_SMILE_1 and IDB_BITMAP_BIG_SMILE_2 to implement owner-draw combo box.

Identifying Item Types

The following macros are defined for different item types:

```
#define COMBO_BUTTON 0

#define COMBO_BIGSMILE 1

#define LIST_SMILE_1 0

#define LIST_SMILE_2 1

#define LIST_SMILE_3 2

#define LIST_SMILE_4 3
```

Each macro represents a different bitmap. We will use these macros to set item data for list box and combo box. Since the item data will be sent along with message WM_DRAWITEM, we can use it to identify item types. This is the same with owner-draw menu.

Two CComboBox type variables m_cbBmp and m_lbBmp are declared in class CCCtlDlg through using Class Wizard, they will be used to access the list box and the combo box. In function CCCtlDlg::OnInitDialog(), the list box and the combo box are initialized as follows:

(Code omitted)

Instead of adding a real string, we pass predefined integers to function CComboBox::AddString(...) and CListBox::AddString(...) For owner-draw list box and combo box, these integers will not be used as buffer addresses for obtaining strings. Instead, they will be sent along with message WM_MEASUREITEM to inform us the item type.

Handling Message WM_MEASUREITEM

The standard message handlers for WM_MEASUREITEM and WM_DRAWITEM are CWnd::OnMeasureItem(...) and CWnd::OnDrawItem(...) respectively, they can be added through using Class Wizard.

The following is the format of function CWnd::OnMeasureItem(...):

void CWnd::OnMeasureItem(int nIDCtl, LPDRAWITEMSTRUCT lpDrawItemStruct);

This function is called to retrieve the size of item. It has two parameters, the first parameter nIDCtl indicates the ID of control whose item's size is being retrieved. The second parameter is a pointer to a DRAWITEMSTRUCT object, and we will use its itemData member to identify the type of the item. Since the value of this member is set in the dialog's initialization stage by calling function CComboBox::AddString(...), it must be one of our predefined macros (LIST_SMILE_1, LIST_SMILE_2...). In the overridden function, we need to check the value of nIDCtl and lpDrawItemStruct->itemData, load the corresponding bitmap resource into a CBitmap type variable, call function CBitmap::GetBitmap(...) to retrieve the dimension of the bitmap, and use it to set both lpDrawItemStrut->itemWidth and lpDrawItemStrut->itemHeight:

(Code omitted)

Handling Message WM_DRAWITEM

The following is the format of function CWnd::OnDrawItem(...):

void CWnd::OnDrawItem(int nIDCtl, LPDRAWITEMSTRUCT lpDrawItemStruct);

It also has two parameters. Like CWnd::OnMeasureItem(...), the first parameter of this function is the control ID, and the second parameter is a pointer to a DRAWITEMSTRUCT type object. This structure contains all the information we need to draw an item of list box or combo box: the DC handle, the item's state, the item data, the position and size where the drawing should be applied. The following portion of the overridden function shows how to load correct bitmap by examining nIDCtl and lpDrawItemStruct->itemData:

(Code omitted)

Five local variables are declared: bmp is used to load the bitmap; dcMemory is used to create memory DC and implement image copying; ptrBmpOld is used to restore the original state of dcMemory; ptrDC is used to store the target DC pointer, which is obtained from hDC member of structure DRAWITEMSTRUCT; bm is used to store the information (including dimension) of the bitmap; rect is used to store the position and size where the bitmap should be copied.

From the above source code we can see, if the control is IDC_LIST, we load one of the four bitmaps (IDB_BITMAP_SMILE_1, IDB_BITMAP_SMILE_2, IDB_BITMAP_SMILE_3 or IDB_BITMAP_SMILE_4) according to the value of lpDrawItemStruct->itemData. If the control is IDC_COMBO, we load IDB_BITMAP_BUTTON_SEL or IDB_BITMAP_BIG_SMILE_1 if the item is selected; and load IDB_BITMAP_BUTTON_UNSEL or IDB_BITMAP_BIG_SMILE_2 if the item is not selected. Here, ODS_SELECTED bit of member lpDrawItemStruct->itemState is checked to retrieve item's state.

The following portion of function CCCtlDlg::OnDrawItem(...) draws the bitmap:

(Code omitted)

Only after the bitmap is loaded successfully will we draw the list box or combo box item. First function CDC::FromHandle(...) is called to obtain a CDC type pointer from HDC handler. Then we create a memory DC (compatible with target DC) and select bmp into this DC. Next, function CDC::BitBlt(...) is called to copy the bitmap from memory DC to target DC. For list box items, there is no special bitmaps for their selected states. In case if an item is selected, the corresponding normal bitmap will be drawn using DSTINVERT mode. This will cause every pixel of the bitmap to change to its complement color. When we pass DSTINVERT to function CDC::BitBlt(...), its fifth argument can be set to NULL.

5.12 Tree Control

Tree control allows us to organize objects into a tree structure. One good example of this type of applications would be a file manager. A tree control can be implemented in both a view window and a dialog box. To implement tree control in a view, we can implement the view using class CTreeView. To implement tree control in a dialog box, we need to use CTreeCtrl class. In this section we will focus on dialog box implementation of tree control.

Like other common controls, we can add tree control resources to the dialog template when designing application's resource. The tree control will have an ID, which could be used to access the control (by either calling function CWnd::GetDlgItem(...) or adding CTreeCtrl type member variable).

Image List

We can associate a bitmap image with each node contained in the tree control. This will make the tree control more intuitive. For example, in a file manager application, we may want to use different images to represent different file types: folder, executable file, DLL file, etc. Before using the images to implement the tree control, we must first prepare them. For tree control (also list control and tab control), these images must be managed by Image List, which is supported by class CImageList in MFC.

Class CImageList can keep and manage a collection of images with the same size. Each image in the list is assigned a zero-based index. After an image list is created successfully, it can be selected into the tree control. We can associate a node with any image contained in the image list. Here image drawing is handled automatically.

If we provide mask bitmaps, only the unmasked portion of the images will be drawn for representing nodes. A mask bitmap must contain only black and white colors. Besides preparing mask bitmaps by ourselves, we can also generate mask bitmaps from the normal images.

To use class CImageList, first we need to declare a CImageList type variable. If we create an image list dynamically by using "new" operator, we need to release the memory when it is no longer in use. Before adding images to the list, we need to call function CImageList::Create(...) to initialize it. This function has several versions, the following is one of them:

```
BOOL CImageList::Create(int cx, int cy, UINT nFlags, int nInitial, int nGrow);
```

Here cx and cy indicate the dimension of all images, nInitial represents the number of initial bitmaps that will be included in the image list, nGrow specifies the number of bitmaps that can be added later. Parameter nFlags indicates bitmap types, it could be ILC_COLOR, ILC_COLOR4, ILC_COLOR8, etc., which specify the bitmap format of the images. For example, ILC_COLOR indicates default bitmap format, ILC_COLOR4 indicates 4-bit DIB format (16-color), ILC_COLOR8 indicates 8-bit DIB format (256-color). We can combine ILC_MASK with any of these bitmap format flags to let the image be drawn with transparency.

The images can be added by calling function CImageList::Add(...). Again, this function has three versions:

```
int CImageList::Add(CBitmap *pbmImage, CBitmap *pbmMask);

int CImageList::Add(CBitmap *pbmImage, COLORREF crMask);

int CImageList::Add(HICON hIcon);
```

The image list can be created from either bitmaps or icons. For the first version of this function, the second parameter is a pointer to the mask bitmap that will be used to implement transparent background drawing. The second version allows us to specify a background color that can be used to generate a mask bitmap from the normal image. Here parameter crMask will be used to create the mask bitmap: all pixels in the source bitmap that have the same color with crMask will be masked when the bitmap is being drawn, and their colors will be set to the current background color. We can choose a background color by calling function CImageList::SetBkColor(...).

To use image list with a tree control, we need to call function

CTreeCtr::SetImageList(...) to assign it to tree control. Then, when creating a node for the tree control, we can use the bitmap index to associate any node with this image.

Adding Nodes

At the beginning, the tree control does not contain any node. Like other common controls, we can initialize it in function CDialog::OnInitDialog(). To add a node to the tree, we need to call function CTreeCtrl::InsertItem(...).

This function also has several versions. The following is the one that has the simplest format:

```
int CTreeCtrl::InsertItem(LPTV_INSERTSTRUCT lpInsertStruct);
```

The only parameter to this function is a TV_INSERTSTRUCT type pointer:

```
typedef struct _TV_INSERTSTRUCT{

HTREEITEM hParent;

HTREEITEM hInsertAfter;

TV_ITEM item;

} TV_INSERTSTRUCT;
```

In a tree control, nodes are managed through handles. After a node is created, it will be assigned an HTREEITEM type handle. Each node has a different handle, so we can use the handle to access a specific node. In the above structure, member hParent indicates which node is the parent of the new node. If we assign NULL to this member, the new node will become the root node. Likewise, member hInsertAfter is used to indicate where the new node should be inserted. We can specify a node handle, or we can use predefined parameters TVI_FIRST, TVI_LAST or TVI_SORT to insert the new node after the first node, last node or let the nodes be sorted automatically.

Member item is a TV_ITEM type object, and the structure contains the information of the new node:

```
typedef struct _TV_ITEM {

UINT mask;

HTREEITEM hItem;
UINT state;

UINT stateMask;

LPSTR pszText;

int cchTextMax;

int iImage;

int iSelectedImage;

int cChildren;

LPARAM lParam;

} TV_ITEM;
```

In order to add new nodes, we need to understand how to use the following four members of this structure: mask, pszText, iImage and iSelectedImage.

Member mask indicates which of the other members in the structure contain valid data. Besides mask, every member of this structure has a corresponding mask flag listed as follows:

(Table omitted)

In order to use members pszText, iImage and iSelectedImage, we need to set the following bits of member mask:

```
TVIF_IMAGE | TVIF_SELECTEDIMAGE | TVIF_TEXT
```

Member pszText is a pointer to a null-terminated string text that will be used to label this node. Member iImage and iSelectedImage are indices to two images contained in the image list that will be used to represent the node's normal and selected state respectively.

By calling function CTreeCtrl::InsertItem(...) repeatedly, we could create a tree structure with desired number of nodes.

Sample

Sample 5.12\CCtl demonstrates how to use tree control in a dialog box. It is a dialog based application generated by Application Wizard. There is only one tree control IDC_TREE in the dialog template. To access it, a member variable CCCtlDlg::m_treeCtrl is added for IDC_TREE through using Class Wizard.

To create the image list, five bitmap resources are prepared, whose IDs are IDB_BITMAP_CLOSEDFOLDER, IDB_BITMAP_DOC, IDB_BITMAP_LEAF, IDB_BITMAP_OPENFOLDER and IDB_BITMAP_ROOT respectively. These bitmaps have the same dimension.

In function CCCtlDlg::OnInitDlalog(), the image list is created as follows:

(Code omitted)

A CBitmap type local variable bmp is declared to load the bitmap resources. First, function CImageList::Create(...) is called to create the image list. Here macro BMP_SIZE_X and BMP_SIZE_Y are defined at the beginning of the implementation file, they represent the dimension of the bitmaps:

```
#define BMP_SIZE_X 16
```

```
#define BMP_SIZE_Y 15
```

We use ILC_MASK flag to let the bitmaps be drawn with transparent background. Originally the image list has five bitmaps, it will not grow later (The fourth and fifth parameter of function CImageList::Create(...) are 5 and 0 respectively).

Next we use variable bmp to load each bitmap resource and add it to the list. When calling function CImageList::Add(...), we pass a COLORREF type value to its second parameter (RGB macro specifies the intensity of red, green and blue colors, and returns a COLORREF type value). This means all the white color in the image will be treated as the background. In the sample application, the background color is set to white:

```
m_pilCtrl->SetBkColor(RGB(255, 255, 255));
```

We can also change the values contained in the RGB macro to set the background to other colors.

Besides this method, we can also prepare all the images in one bitmap resource (just like the tool bar resource), and call the following versions of function CImageList::Create(...) to create the image list:

```
BOOL CImageList::Create(UINT nBitmapID, int cx, int nGrow, COLORREF crMask);
BOOL CImageList::Create(LPCTSTR lpszBitmapID, int cx, int nGrow, COLORREF crMask);
```

Here nBitmapID or lpszBitmapID specifies the bitmap resource ID, and cx specifies the horizontal dimension of an individual image. With this parameter, the system knows how to divide one big image into several small images.

After creating the image list, function CTreeCtrl::SetImageList(...) is called to assign the image list to the tree control:

```
......

m_treeCtrl.SetImageList(m_pilCtrl, TVSIL_NORMAL);

......
```

Since the image list is created dynamically, we need to release it when it is no longer in use. The best place to destroy the image list is in CDialog::OnDestroy(), when the dialog box is about to be destroyed. This function is the handler of WM_DESTROY message, which could be easily added through using Class Wizard. The following is the implementation of this function in the sample:

(Code omitted)

We call function CImageList::GetImageList(...) to obtain the pointer to the image list, then call CImageList::DeleteImageList() to delete the image list. Please note that this function releases only the images stored in the list, it does not delete CImageList type object. After the image list is deleted, we still need to use keyword "delete" to delete this object.

In the sample, a tree with the structure showed in Figure 5-10 is created.

This tree has 7 nodes. Node "Root" is the root node, it has one child node "Doc". Node "Doc" has a child node "Folder", and node "Folder" has four child nodes "Leaf1", "Leaf2", "Leaf3" and "Leaf4". The following portion of function CCCtlDlg::OnInitDialog() shows how the node "Root" is created in the sample:

(Code omitted)

Variable tvInsertStruct is declared at the beginning of function CCCtlDlg::OnInitDialog(), it is a TV_INSERTSTRUCT type object. To create a specific node, we must stuff this object with node information and call function CTreeCtrl::InsertItem(...). This function returns a handle to the newly created node, which is stored in variable hTreeItem and will be used to create its child node. The following portion of function CCCtlDlg::OnInitDialog() shows how the child node is created:

(Code omitted)

This procedure is exactly the same for other nodes. For different nodes, the only difference of this procedure is that each node has different parent node, uses different image index and text string. For all nodes, their normal states and selected states are represented by the same image (member iImage and iSelectedImage are assigned the same image index), so the image will not change if we select a node.

With the above implementations, the tree control can work. By compiling and executing the application at this point, we will see a tree with seven nodes, which are represented by different labels and images. A node can be expanded or collapsed with mouse clicking if it has child node(s).

5.13 Handling Tree Control Messages

There are many messages associated with the tree control. We need to write message handlers for the tree control in order to customize its default behavior. In sample 5.13\CCtl we will demonstrates two methods of customizing a tree control: 1) How to change a node's associated image dynamically. 2) How to enable label editing.

Sample 5.13\CCtl is based on sample 5.12\CCtl. In this sample the image associated with node "Folder" will be changed automatically according to its current state (expanded or collapsed). If it is expanded, image IDB_BITMAP_OPENFOLDER will be associated with this node; if it is collapsed, image IDB_BITMAP_CLOSED_FOLDER will be used. Also, the application supports dynamic label editing: if the user clicks mouse's left button on the label of a node, that node will enter editing mode, and we can edit the text string as if we were using an edit box.

The messages associated with node expanding and collapsing are TVN_ITEMEXPANDING and TVN_ITEMEXPANDED. The former message is sent when a node is about to be expanded or collapsed, and the latter message is sent after such action is completed. In our case, we need to handle the former message to change a node's image before its state changes.

Handling TVN_ITEMEXPANDING to Change a Node's Associated Image

In MFC, message TVN_ITEMEXPANDING can be mapped to a member function as follows:

```
void CTreeCtrl::OnItemexpanding(NMHDR *pNMHDR, LRESULT *pResult)
```

```
{

NM_TREEVIEW *pNMTreeView = (NM_TREEVIEW*)pNMHDR;

*pResult = 0;

}
```

Variable pNMTreeView is a pointer to NM_TREEVIEW type object obtained from the message parameters, it contains the information about the node being clicked:

```
typedef struct _NM_TREEVIEW{

NMHDR hdr;

UINT action;

TV_ITEM itemOld;

TV_ITEM itemNew;

POINT ptDrag;

} NM_TREEVIEW;
```

The most important member of this structure is action, it could be either TVE_EXPAND (indicating the node is about to expand) or TVE_COLLAPSE (indicating the node is about to collapse). Two other useful members are itemOld and itemNew, both of them are TV_ITEM type objects and contain old and new states of the node respectively. We can check iImage member of itemNew to see if the associated image is 2 or 3 (Indices 2 and 3 correspond to image IDB_BITMAP_CLOSED_FOLDER and IDB_BITMAP_OPENFOLDER respectively, which indicate that the node represents a folder. In the sample, we will not change other node's image when they are being expanded or collapsed), if so, we need to call function CTreeCtrl::SetItemImage(...) to change the image of the node if necessary.

We can handle this message either within class CTreeCtrl or CDialog. Handling the message in CTreeCtrl has the advantage that once the feature is implemented, we can reuse this class in other applications without adding additional code.

In the sample, a new class MCTreeCtrl is designed for this purpose. It is added to the application through using Class Wizard. Also, message handlers MCTreeCtrl::OnItemexpanding(...) and MCTreeCtrl::OnEndlabeledit(...) are added to dynamically change node's associated images and enable label editing (Label editing will be discussed later).

The following is the implementation of function MCTreeCtrl::OnItemexpanding(...):

(Code omitted)

If the node is about to expand and its associated image is 2, we associate image 3 with this node. This is implemented through calling function CTreeCtrl::SetItemImage(...), which has the following format:

```
BOOL CTreeCtrl::SetItemImage(HTREEITEM hItem, int nImage, int
nSelectedImage);
```

The first parameter of this function is the handle of tree control, which can be obtained from pNMTreeView->itemNew.hItem. Similarly, if the node is about to collapse and its associated image is 3, we call this function to associate image 2 with this node.

Handling TVN_ENDLABELEDIT to Enable Label Editing

The next feature we want to add is label editing. If we are familiar with "Explorer" application in Windows95(, we know that the file or directory names (which are node labels) can be edited dynamically by single clicking on it.

The first step of enabling label editing is to set "Edit labels" style when adding tree control resource to the dialog template. The following lsts necessary steps of doing this: 1) Invoke "Tree Control Properties" property sheet, click "Styles" tab. 2) Check "Edit labels" check box (Figure 5-11).

Label editing will be enabled if this style is selected. However, if we do not add code to change the label at the end of editing, the label will remain unchanged after it is edited. To make this happen, we must handle message TVN_ENDLABELEDIT.

Standard TVN_ENDLABELEDIT message handler added by Class Wizard will have the following format:

```
void MCTreeCtrl::OnEndlabeledit(NMHDR *pNMHDR, LRESULT *pResult)

{
```

```
TV_DISPINFO *pTVDispInfo = (TV_DISPINFO*)pNMHDR;

*pResult=0;

}
```

Here pTVDispInfo is a pointer to TV_DISPINFO type object, which can be obtained from the message parameter. The most useful member of TV_DISPINFO is item, which is a TV_ITEM type object. Three members of item contain valid information: hItem, lParam, and pszText. We could use hItem to identify the node and use pszText to obtain the updated text string. If pszText is a NULL pointer, this means the editing is canceled (Label editing can be canceled through pressing ESC key). Otherwise it will contain a NULL-terminated string. The following is the implementation of this message handler:

(Code omitted)

If the editing is not canceled, we need to call function CTreeCtrl::SetItemText(...) to set the node's new text, which has the following format:

```
BOOL CTreeCtrl::SetItemText(HTREEITEM hItem, LPCTSTR lpszItem);
```

This function is similar to CTreeCtrl::SetItemImage(...). Its first parameter is the handle of tree control, and the second parameter is a string pointer to the new label text.

There are other messages associated with label editing, one useful message is TVN_BEGINLABELEDIT, which will be sent when the editing is about to begin. We can handle this message to disable label editing for certain nodes. In the message handler, if we assign a non-zero value to the content of pResult, the edit will stop. Otherwise the label editing will go on as usual.

Using the New Class

In the new sample, variable CCCtlDlg::m_treeCtrl is declared by class MCTreeCtrl instead of CTreeCtrl. First the header file that contains class MCTreeCtrl is included in file "CCtlDlg.h", then the declaration of member variable CCCtlDlg::m_treeCtrl is changed:

```
class CCCtlDlg : public CDialog

{

......
```

```
//{{AFX_DATA(CCCtlDlg)

enum { IDD = IDD_CCTL_DIALOG };

MCTreeCtrl m_treeCtrl;
```

That's all we need to do in order to add new features to the sample.

When editing a label, we can not press RETURN key to end the editing. This is because in a dialog box, RETURN is used to close the dialog box by default. If we want to change this feature, we need to override function CDialog::PreTranslateMessage(...) and intercept RETURN key stroke messages as we did for combo box in sample 5.9\CCtl.

5.14 Drag-n-Drop

Another nice feature we can add to tree control is to change the tree structure by dragging and dropping. By implementing this, we can copy or move one node (and all its child nodes) to another place with few mouse clicks.

Sample 5.14\CCtl demonstrates drag-n-drop implementation. It is base on sample 5.13\CCtl with new messages handled in class MCTreeCtrl.

Handling New Messages

To implement node dragging and dropping, we need to handle the following three messages: TVN_BEGINDRAG, WM_MOUSEMOVE and WM_LBUTTONUP. The first message is sent when the user starts node dragging. After receiving this message, we need to prepare node dragging. Message WM_MOUSEMOVE should be handled when an item is being dragged around: when the mouse cursor hits a possible target node, we need to highlight it to remind the user that the source node could be dropped here. When we receive message WM_LBUTTONUP, we need to check if the node can be copied to the new place, if so, we need to implement node copy (or move).

In the sample, message handlers of TVN_BEGINDRAG, WM_MOUSEMOVE and WM_LBUTTONUP are added through using Class Wizard. The default TVN_BEGINDRAG message handler has the following format:

```
void MCTreeCtrl::OnBegindrag(NMHDR *pNMHDR, LRESULT *pResult)

{

NM_TREEVIEW *pNMTreeView=(NM_TREEVIEW *)pNMHDR;
*pResult=0;

}
```

Here, several issues must be considered when a node is being dragged around:

1) To determine which node is being clicked for dragging after receiving message TVB_BEGINDRAG, we can call API function ::GetCursorPos(...) to retrieve the current position of mouse cursor, call function CWnd::ScreenToClient(...) to convert its coordinates, and call CTreeCtrl::HitTest(...) to obtain the handle of the node that is being clicked.

2) We must provide a dragging image that will be drawn under the mouse cursor to give the user an impression that the node is being "dragged". An easiest way of preparing this image is to call function CTreeCtrl::CreateDragImage(...), which will create dragging image using the bitmap associated with this node. This function will return a CImageList type pointer, which could be further used to implement dragging. We can also create our own customized image list for dragging operation, the procedure of creating this type of image list is the same with creating a normal image list.

3) We can call function CImageList::SetDragCursorImage(...) to combine an image contained in the image list with the cursor to begin dragging.

4) We must lock the tree control window when a node is being dragged around to avoid any change happening to the tree structure (When a node is being dragged, the tree should not change). When we want to do a temporary update (For example, when the dragging image enters a node and we want to highlight that node to indicate that the source can be dropped there), we must first unlock the window, then implement the update. If we want the dragging to be continued, we must lock the window again.

5) Function CImageList::EnterDrag(...) can be called to enter dragging mode and lock the tree control window. Before we make any change to the tree control window (For example, before we highlight a node), we need to call function CImageList::LeaveDrag(...) to unlock the tree control window. After the updates, we need to call CImageList::EnterDrag(...) again to lock the window. This will prevent the tree control from being updated when a node is being dragged around.

6) We can show or hide the dragging image by calling function CImageList::DragShowNolock(...) without locking the tree control window. This function is usually called before CImageList::SetDragCursorImage(...) is called.

7) To begin dragging, we need to call CImageList::BeginDrag(...); to move the

dragging image to a specified position, we can call CImageList::DragMove(...); to end dragging, we need to call CImageList::EndDrag().

8) We can highlight a node by calling function CTreeCtrl::SelectDropTarget(...).

The following is a list of prototypes of the above-mentioned functions:

BOOL CImageList::DragShowNolock(BOOL bShow);

(Table omitted)

BOOL CImageList::BeginDrag(int nImage, CPoint ptHotSpot);

(Table omitted)

BOOL CImageList::DragMove(CPoint pt);

(Table omitted)

BOOL CImageList::DragEnter(CWnd *pWndLock, CPoint point);

(Table omitted)

BOOL CImageList::DragLeave(CWnd *pWndLock);

(Table omitted)

BOOL CTreeCtrl::SelectDropTarget(HTREEITEM hItem);

(Table omitted)

When the mouse button is released, we need to check if the source node can be copied to the target node. In the sample, we disable copying under the following three conditions: 1) The source node is the same with the target node. 2) The target node is a descendent node of the source node. 3) The target node does not have any child node. By setting these restrictions, a node can only be copied to become the child of its parent node (direct or indirect).

We can use function CTreeCtrl::GetParentItem(...) to decide if one node is the descendent of another node:

HTREEITEM CTreeCtrl::GetParentItem(HTREEITEM hItem);

This function will return an HTREEITEM handle, which specifies the parent of node hItem. By repeatedly calling this function we will finally get a NULL return value (This indicates that the root node was encountered). Using this method, we can easily find out a list of all nodes that are parents of a specific node.

New Member Variables and Functions

To implement drag-n-drop, several new variables and functions are declared in class MCTreeCtrl:

(Code omitted)

Here, Boolean type variable m_bIsDragging is used to indicate if the drag-n-drop activity is undergoing. Pointer m_pilDrag will be used to store the dragging image. Variables m_hTreeDragSrc and m_hTreeDragTgt are used to store the handles of source and target nodes respectively. We can use them to implement copying right after the source node is dropped. Function MCTreeCtrl::IsDescendent(...) is used to judge if one node is the descendent node of another, and MCTreeCtrl::CopyItemTo(...) will copy one node (and all its descendent nodes) to another place.

Node Copy

When copying a node, we want to copy not only the node itself, but also all its descendent nodes. Since we do not know how many descendents a node have beforehand, we need to call function MCTreeCtrl::CopyItemTo(...) recursively until all the descendent nodes are copied. The following is the implementation of this function:

(Code omitted)

This function copies node hTreeDragSrc along with all its descendent nodes, and make them the child nodes of hTreeDragTgt. First we call function CTreeCtrl::GetItem(...) to retrieve source node's information. We must pass a TV_ITEM type pointer to this function, and the corresponding object will be filled with the information of the specified node. Here, we use member item of structure TV_INSERTSTRUCT to receive a node's information (Variable tvInsertStruct is declared by TV_INSERTSTRUCT, it will be used to create new nodes). When calling this function, member mask of TV_ITEM structure specifies which member should be filled with the node's information. In our case, we want to know the handle of this node, the associated images, the text of the label, the current state (expanded, highlighted, etc.), and if the node has any child node. So we need to set the following bits of member mask: TVIF_CHILDREN, TVIF_HANDLE, TVIF_IMAGE, TVIF_SELECTEDIMAGE, TVIF_TEXT and TVIFF_STATE. Note we must provide our own buffer to receive the label text. In the function, szBuf is declared as a char type array and its address is stored in pszText member of TV_ITEM. Then we use tvInsertStruct to create a new node. Since we have already stuffed item member with valid information, here we only need to assign the target handle (stored in hTreeDragTgt) to hParent, and assign TVI_LAST to hInsertAfter. This will make the new node to become the child of the target node, and be added to the end of all child nodes under the target node. Next we check if this node has any child node. If so, we find out all the child nodes and call this function recursively to copy all the child nodes. For this step, we use the newly created node as the target node, this will ensure that the original tree structure will not change after copying.

In the final step, we call function CTreeCtrl::GetChileItem(...) to find out a node's first child node, then call function CTreeCtrl::GetNextitem(...) repeatedly to get the rest child nodes. The two functions will return NULL if no child node is found.

TVN_BEGINDRAG

Now we need to implement TVN_BEGINDRAG message handler. First, we need to obtain the node that was clicked by the mouse cursor. To obtain the current position of mouse cursor, we can call API function ::GetCursorPos(...). Since this position is measured in the screen coordinate system, we need to further call function CWnd::ScreenToClient(...) to convert the coordinates to the coordinate system of the tree control window. Then we can set variable m_bIsDragging to TRUE, and call function CTreeCtrl::HitTest(...) to find out if the mouse cursor is over any node:

(Code omitted)

Next, we need to obtain dragging image list for this node. The dragging image list is created by calling function CTreeCtrl::CreateDragImage(...). After this, the address of the image list object is stored in variable m_pilDrag. If the image list is created successfully, we call several member functions of CImageList to display the dragging image and enter dragging mode. If not, we should not start dragging, and need to set the content of pResult to a non-zero value, this will stop dragging:

(Code omitted)

WM_MOUSEMOVE

Then we need to implement WM_MOUSEMOVE message handler. Whenever the mouse cursor moves to a new place, we need to call function CImageList::DragMove(...) to move the dragging image so that the image will always follow the mouse's movement. We need to check if the mouse hits a new node by calling function CTreeCtrl::HitTest(...). If so, we must leave dragging mode by calling function CImageList:: DragLeave(...), highlight the new node by calling function CTreeCtrl::SelectDropTarget(...), and enter dragging mode again by calling function CTreeCtrl::DragEnter(...). The reason for doing this is that when dragging is undergoing, the tree control window is locked and no update could be implemented successfully. The following is the implementation of this message handler:

(Code omitted)

WM_LBUTTONUP

Finally we need to implement WM_LBUTTONUP message handler. In this handler, we must first leave dragging mode and end dragging. This can be implemented by calling functions CImageList:: DragLeave(...) and CImageList::EndDrag() respectively. Then, we need to delete dragging image list object:

(Code omitted)

The following code fragment shows how to judge if the source node can be copied to become the child of the target node:

(Code omitted)

If the source and target are the same node, or target node does not have any child node, or source node is the parent node (including indirect parent) of the target node, the copy should not be implemented. Otherwise, we need to call function MCTreeCtrl::CopyItem(...) to implement node copy:

(Code omitted)

If we want the node to be moved instead of being copied, we can delete the source node after copying it. The source node and all its child nodes will be deleted by calling function CTreeCtrl::DeleteItem(...).

Functions CWnd::SetCapture() and ::ReleaseCapture() are also called in MCTreeCtrl:: OnBegindrag(...) and MCTreeCtrl::OnLButtonUp(...) respectively to set and release the window capture. By doing this, we can still trap mouse messages even if it moves outside the client window when dragging is undergoing.

That's all we need to do for implementing drag-n-drop copying. By compiling and executing the sample application at this point, we will be able to copy nodes through mouse dragging. With minor modifications to the above message handlers, we can easily implement both node copy and move as follows: when CTRL key is held down, the node can be copied through drag-n-drop, when there is no key held down, node will be moved.

5.15 List Control

A list control is another type of control that can be used to manage a list of objects. Rather than storing items in a tree structure, a list control simply organize them into an array. There is no parent or child node in a list control.

A list control can be viewed in different styles: 1) Normal icon style ¾ each item is represented by a big icon. 2) Small icon style ¾ each item is represented by a small icon. 3) List style ¾ all items are represented by small icons contained in a vertical list. 4) Report style ¾ the details of all items are listed in several vertical lists.

In MFC, list control is supported by class CListCtrl. Implementing list control is similar to implementing tree control: the list control resource can be created in dialog template, then the list control can be initialized in the dialog's initialization stage. Each item in the list control can be associated with one or more images, they will be used to represent the item in different styles. Usually we need to associate two images for an item: one big image for normal style, and a small image for other three styles. In general case, we need to prepare two image lists to create a list control.

LV_COLUMN and LV_ITEM

The procedure of initializing list control is similar to that of tree control. First we need to create two image lists: one for normal icon style; one for small icon style. Then we need to call function CListCtrl::SelectImageList(...) to associate the image lists with the list control. The following is the format of this function:

```
CImageList *CListCtrl::SetImageList(CImageList *pImageList, int nImageList);
```

Here pImageList is a pointer to the image list, and nImageList specifies the type of image list: it could be LVSIL_NORMAL or LVSIL_SMALL, representing which style the image list will be used for.

After the image list is set, we need to add columns for the list control (Figure 5-12). This can be implemented by calling function CListCtrl::InsertColumn(...), which has the following format:

```
int CListCtrl::InsertColumn(int nCol, const LV_COLUMN* pColumn);
```

The function has two parameters. The first one indicates which column is to be added (0 based index), and the second one is a pointer to LV_COLUMN type object:

```
typedef struct _LV_COLUMN {

UINT mask;

int fmt;

int cx;

LPSTR pszText;

int cchTextMax;

int iSubItem;

} LV_COLUMN;
```

Here, member mask indicates which of the other members contain valid values, this is the same with structure LV_ITEM. Member fmt indicates the text alignment for the column, it can be LVCFMT_LEFT, LVCFMT_RIGHT, or LVCFMT_CENTER. Member cx indicates the width of the column, and iSubItem indicates its index. Member pszText is a pointer to the text string that will be displayed for each column. Finally, cchTextMax specifies the size of buffer pointed by pszText.

After columns are created, we need to add list items. For each list item, we need to insert a sub-item in each column. For example, if there are three columns and 4 list items, we need to add totally 12 sub-items.

To add a sub-item, we need to stuff an LV_ITEM type object then call function CListCtrl:: InsertItem(...), which has the following format:

```
int CListCtrl::InsertItem(const LV_ITEM* pItem);
```

The following is the format of structure LV_ITEM:

```
typedef struct _LV_ITEM {

UINT mask;

int iItem;

int iSubItem;

UINT state;
UINT stateMask;

LPSTR pszText;

int cchTextMax;

int iImage;

LPARAM lParam;

} LV_ITEM;
```

The usage of this structure is similar to that of structure TV_ITEM. For each item, we need to use this structure to add every sub-item for it. Usually only the sub-items contained in the first column will have an associated image (when being displayed in report style), so we need to set image for each item only once. Member iItem and iSubItem specify item index and column index respectively.

Sample

Sample 5.15\CCtl demonstrates how to use list control. It is a dialog-based application generated by Application Wizard. In this sample, a four-item list is implemented, which can be displayed in one of the four styles. When it is displayed in report style, the control has four columns. The first column lists four shapes: square, rectangle, circle, triangle. The second column lists the formula for calculating their perimeter, and the third column lists the formula for calculating their area.

Creating Image Lists

In the dialog template, the list control has an ID of IDC_LIST. In order to access this control, a CListCtrl type variable m_listCtrl is added to class CCCtldlg through using Class Wizard.

Four icon resources are added to the application for creating image lists. Their IDs are IDI_ICON_SQUARE, IDI_ICON_RECTANGLE, IDI_ICON_CIRCLE and IDI_ICON_TRIANGLE respectively. In the previous samples, we created image list from bitmap resource all the time. Actually, it can also be created from icon resources as well.

In function CCCtlDlg::OnInitDialog(), first two image lists are created and selected into the list control:

(Code omitted)

We could use the same icon to create both 32(32 and 16(16 image lists. When creating the 16(16 image list, the images will be automatically scaled to the size specified by the image list. Since we allocate memory for creating image list in dialog's initialization stage, we need to release it when the dialog box is being destroyed. For this purpose, a WM_DESTROY message handler is added through using Class Wizard, within which the image lists are deleted as follows:

(Code omitted)

If we release the memory used by image lists this way, we must set "Share image list" style for the list control. This allows image list to be shared among different controls. If we do not set this style, the image list will be destroyed automatically when the list control is destroyed. In this case, we don't have to release the memory by ourselves. To set this style, we need to invoke "List control properties" property sheet, go to "More styles" page, and check "Share image list" check box (Figure 5-13).

(Figure 5-13 omitted)

Creating Columns

First we need to create three columns, whose titles are "Shape", "Perimeter", and "Area" respectively. The following portion of function CCCtlDlg::OnInitDialog() creates each column:

(Code omitted)

The client window's dimension is retrieved by calling function CWnd::GetClientRect(...) and then stored in variable rect. The horizontal size of each column is set to 1/3 of the width of the client window.

Creating Sub-items

Since there are totally three columns, for each item, we need to create three sub-items. The following is the portion of function CCCtlDlg::OnInitDialog() that demonstrates creating one sub-item:

(Code omitted)

Function CListCtrl::InsertItem(...) is called to add a list item and set its first sub-item. The rest sub-items should be set by calling function CListCtrl::SetItem(...). For these sub-items, we don't need to set image again, so LVIF_IMAGE flag is not applied when function CListCtrl::SetItem(...) is called.

Changing List Style Dynamically

The style of the list control can be set in property sheet "List control properties" before the application is compiled(see Figure 5-14). But, sometimes we may want to provide the user with the power of changing this style dynamically. When the program is running, we can call API function ::SetWindwoLong(...) to change the application's style. For list control, we can choose from one of the following styles: LVS_ICON, LVS_SMALLICON, LVS_LIST and LVS_REPORT.

In the sample, four radio buttons are added to the dialog template for selecting different styles. Their IDs are IDC_RADIO_ICON, IDC_RADIO_SMALLICON, IDC_RADIO_LIST and IDC_RADIO_REPORT respectively. We need to handle BN_CLICKED message for the four radio buttons in order to respond to mouse events. These message handlers are added through using Class Wizard. Within the member functions, the style of the list control is changed according to which radio button is being clicked. The following is one of the message handlers that sets the style of the list control to "Normal Icon":

(Code omitted)

First, the list control's old style is retrieved by calling function ::GetWindowLong(...), and is bit-wisely ANDed with LVS_TYPEMASK, which will turn off all the style bits. Then style LVS_ICON is added to the window style (through bit-wise ORing), and function ::SetWindowLong(...) is called to update the new style. Both function ::GetWindowLong(...) and ::SetWindowLong(...) require a window handle, it could be obtained by calling function CWnd:: GetSafeHwnd().

The list control and tree control can also be implemented in SDI and MDI applications. In this case, we need to use classes derived from CListView or CTreeView. Although the creating procedure is a little different from that of a dialog box, the properties of the controls are exactly the same for two different types of applications. We will further explore list control and tree control in chapter 15.

5.16 Tab Control

In the previous sample, we used radio buttons to let the user set the style of list control dynamically. An alternate way of doing this is to use tab control, which is widely used in various types of applications. Usually a tab control is used together with dialog box to implement property sheets, which can let the user easily switch among different property pages. This topic will be discussed in a chapter 7. Here, we will discuss some basics on how to implement tab control and handle its messages.

Using Tab Control

In MFC, tab control can be implemented by using class CTabCtrl. A tab control can be associated with an image list, so we can display both image and text on each tab. The steps of using a tab control is very similar to that of list control and tree control: first we need to add tab control resource to the dialog template; then in the dialog's initialization stage, we need to create the image list, select it into the tab control, and initialize the tab control. The function that can be used to assign image list to a tab control is CTabCtrl::SetImageList(...), which has the following format:

```
CImageList *CTabCtrl::SetImageList(CImageList *pImageList);
```

The function that can be used to add an item to the tab control is CTabCtrl::InsertItem(...):

```
BOOL CTabCtrl::InsertItem(int nItem, TC_ITEM *pTabCtrlItem);
```

The first parameter of this function is the index of the tab (zero based), and the second parameter is a pointer to TC_ITEM type object. Before calling this function, we need to stuff structure TC_ITEM with tab's information:

```
typedef struct _TC_ITEM {

UINT mask;

UINT lpReserved1;

UINT lpReserved2;

LPSTR pszText;

int cchTextMax;

int iImage;

LPARAM lParam;

} TC_ITEM;
```

We need to use three members of this structure in order to create a tab with both text and image: mask, pszText and iImage. Member mask indicates which of the other members of this structure contains valid data; member pszText is a pointer to string text; and iImage is an image index to the associated image list. We see that using this structure is very similar to that of TV_ITEM and LV_ITEM.

To respond to tab control's activities, we need to add message handlers for it. The most commonly used messages of tab control are TCN_SELCHANGE and TCN_SELCHANGING, which indicate that the current selection has changed or the current selection is about to change respectively.

Sample 5.16\CCtl demonstrates how to use tab control, it is based on sample 5.15\CCtl. In this sample, four radio buttons are replaced by a tab control IDC_TAB (see Figure 5-14). Also, message handlers of radio buttons are removed. In order to access the tab control, a CTabCtrl type control variable m_tabCtrl is added to class CCCtlDlg through using Class Wizard. Beside this, four bitmap resources IDB_BITMAP_ICON, IDB_BITMAP_SMALLICON, IDB_BITMAP_LIST and IDB_BITMAP_REPORT are added to the application to create image list for tab control.

In function CCCtlDlg::OnInitDialog(), first the image list is created and assigned to the tab control. Next, four items are added to the tab control:

(Code omitted)

Macro TAB_BMP_WIDTH and TAB_BMP_HEIGHT are defined as the width and height of the bitmaps. In function CCCtlDlg::Destroy(), the following statements are added to delete the tab items and the image list used by the tab control:

(Code omitted)

Handling Tab Control Message

We trap message TCN_SELCHANGE to respond to the changes on the tab control. After receiving this message, we call function CTabCtrl::GetCurSel() to obtain the newly selected item, then call function ::SetWindowLong(...) to set the style of the list control accordingly. In the sample, function CCCtlDlg::OnSelchangeTab(...) is added to class CCCtlDlg through using Class Wizard for handling this message. It is implemented as follows:

(Code omitted)

With the above implementation, we can change the list control's style dynamically through using the tab control.

5.17 Animate Control and Progress Control

Using Animate Control and Progress Control

Animate and progress controls are very useful, both of them can be implemented in a dialog box very easily. In MFC, the classes used to implement animate and progress controls are CAnimateCtrl and CProgressCtrl respectively.

Sample 5.17\CCtl demonstrates how to use two types of controls. It is a dialog-based application generated by Application Wizard.

Like any other common controls, the first step of using animate and progress controls is to add their resources to the dialog template. There are very few styles that can be customized, and the meanings of them are all self-explanatory. To access the controls, we can use Class Wizard to add member variables for them.

For animate control, the functions we need to call for implementing animation are CAnimateCtrl::Open(...) and CAnimateCtrl::Play(...). The first function lets us open an animation resource either from a file or from an AVI resource. The second function lets us play the loaded AVI data.

For progress control, we need to call function CProgressCtrl::SetRange(...) to set the upper and lower limits, call CProgressCtrl::SetStep(...) to specify the incremental step, and call CProgressCtrl:: StepIt() to advance the current position of progress bar. Each time we call this function, the progress bar will advance one step. In order to let the progress bar advance continuously, we need to link it to some events that happen all the time. In the sample, a timer is used to generate this type of events.

Timer

Timer is a very useful resource in Windows(operating system. Once we set the timer and specify the time out period, it will start to count down and send us a WM_TIMER message when time out happens. The timer can be set within any CWnd derived class by calling function CWnd::SetTimer(...). Timers with different IDs are independent upon one another, so we can set more than one timer to handle complex situation.

The following is the prototype of function CWnd::SetTimer(...):

```
UINT CWnd::SetTimer

(

UINT nIDEvent, UINT nElapse,

void(CALLBACK EXPORT *lpfnTimer)(HWND, UINT, UINT, DWORD)

);
```

The function has three parameters. Parameter nIDEvent is an event ID. This ID can be any integer, and we need to use different ID for different event in order to distinguish between them. Parameter nElapse specifies time out period, whose unit is millisecond. Parameter lpfnTimer is a pointer to a callback function that will be used to handle time out message. We can also pass NULL to this parameter and add WM_TIMER message handler to receive this message.

In the sample, the IDs of the animate control and progress control are IDC_ANIMATE and IDC_PROGRESS. Also, the variables used to access them are m_animateCtrl and m_progressCtrl respectively.

Custom Resource

The AVI data can be included in the application as a resource. However, Developer Studio does not support this kind of resource directly. So we have to treat it as a custom resource. We can create AVI resource from a "*.avi" file through following steps: 1) Execute Insert | Resource command, then click "Import" button from the "Insert resource" dialog box. 2) From the popped up "File open" dialog box, browse and select a "*.avi" file and open it (we can use "5.17\CCtl\search.avi" or any other "*.avi" file for this purpose). 3) When we are asked to provide the resource type, input "AVI". 4) Name the resource ID as IDR_AVI.

Sample Implementation

In the dialog box's initialization stage, we need to initialize the animate control, progress control and set timer as follows:

(Code omitted)

First, function CAnimateCtrl::Open(...) is called to open the animation resource, then function CAnimateCtrl::Play(...) is called to play the AVI data. When doing this, we pass 0 to its first parameter and -1 to the second parameter, this will let the animation be played from the first frame to the last frame. The third parameter is also -1, this means the animation will be played again and again without being stopped.

Then we initialize the range of progress control from 0 to 50, incremental step 2, and a timer with time out period of 500 milliseconds is set.

Message WM_TIMER can be handled by adding message handlers, this can be easily implemented through using Class Wizard. In the sample, this member function is implemented as follows:

(Code omitted)

The only parameter of this function is nIDEvent, which indicates the ID of the timer that has just timed out. If we have two or more timers set within the same window, by examining this ID we know which timer has timed out. In the sample, when timer times out, we simply call function CProgressCtrl::StepIt() to advance the progress bar one step forward.

Summary:

1) A spin control must work together with another control, which is called the "Buddy" of the spin control. Usually the "Buddy" is an edit box, but it could be any other types of controls such as button or static control.

2) To set buddy automatically, we must make sure that the buddy window is the previous window of the spin control in Z-order.

3) The buddy can also be set by calling function CSpinButtonCtrl::SetBuddy(...).

4) If we set "Set buddy integer" style, the spin control will notify the buddy control to update its contents whenever the position of the spin control changes. If we set this style, the buddy edit box can display only integers.

5) If we want to customize the behavior of buddy control, we need to handle message UDN_DELTAPOS. This message will be sent when the position of the spin control changes. By doing this, we can let the buddy control display text strings or bitmap images.

6) Slider control shares the same message with scroll bars. By handling message WM_HSCROLL (for horizontally orientated sliders) and WM_VSCROLL (for vertically orientated sliders), we can trap the mouse activities on the slider.

7) List box can be implemented in different styles: single selection, multiple selection, extended selection. By default, the items in the list box will contain only characters, and they will be alphabetically sorted. These styles can be changed by calling member functions of CListCtrl.

8) A list box can be used to display directories and files contained in a specific directory by calling function CListCtrl::Dir(...).

9) We can handle LBN_... type messages to customize the default behavior of a list control.

10) A combo box is composed of an edit box and a list box. Because they are not created by MFC code, we can not access them through the normal method.

11) To trap RETURN, ESC keys for combo box, we need to override function CWnd:: PreTranslateMessage(...).

12) To implement subclass for edit box contained in a combo box, we need to call function CWnd:: SubclassWindow(...) instead of CWnd::SubclassDlgItem(...).

13) To create owner-draw list box or combo box, first we need to set "Owner draw" style, then override WM_MEASUREITEM and WM_DRAWITEM message handlers.

14) Image list is used by tree control, list control and tab control. Once an image list is assigned to a control, the images contained in the list can be accessed through their zero-based indices.

15) To use a tree control, we can create its resource in the dialog template. Then in the dialog's initialization stage, we can create the tree structure. A tree item can be added to the control by stuffing a TV_ITEM type object, then calling function CTreeCtrl::InsertItem(...).

16) We need to handle message TVN_ITEMEXPANDING or TVN_ITEMEXPANDED to customize a tree control's expanding and collapsing behaviors.

17) We need to set "Edit labels" style and handle TVN_ENDLABELEDIT message to enable label editing for tree control.

18) We need to handle messages TVN_BEGINDRAG, WM_MOSUEMOVE, and WM_LBUTTONUP to enable drag-n-drop for tree control.

19) The list box can be displayed in four different styles: Normal (big) icon style, small icon style, list style, and report style. We can select one style when the list box resource is being created. If we want to change the style dynamically, we need to call function ::SetWindowLong(...).

20) Because list control can be used to represent items in different styles, usually we need to prepare two image lists (big icon and small icon) for a list control.

21) To create a list control, we need to create columns first. For each column, we need to create sub-items for all the items contained in the list.

22) To create a column for a list control, we need to stuff an LV_COLUMN type object and call function CListCtrl::InsertColumn(...). To create an item, we need to stuff an LV_ITEM type object and call function CListCtrl::InsertItem(...). To set the rest sub-items, we need to stuff LV_ITEM type objects and call function CListCtrl::SetItem(...).

23) To use tab control, first we need to create tab control resource in the dialog template. Then in the dialog's initialization stage, we need to create and select the image list, stuff TV_ITEM type objects and call function CTabCtrl::InsertItem(...) to add items.

24) The animate control can be used to play AVI data. Because this is not a standard resource supported in Developer Studio, we need to create custom resource if we want to include AVI data in an application as a resource.

25) The progress control is used to indicate the progress of events. In order to synchronize the progress bar with the events, we need to advance the progress bar within the event's message handler.

BACK TO INDEX

&nb

Chapter 6 Dialog Box

Dialog box is very common for all types of applications, it is used in almost every program. Usually a dialog box is built from resources: we design everything in the dialog template, then use a CDialog derived class to implement it. We can call function CDialog::DoModal() to invoke the dialog box, use member variables to access the common controls, and add message handlers to process mouse or keyboard related events.

In this chapter we will discuss some topics on customizing normal dialog boxes. By using the methods introduced in this chapter, we are able to make our dialog boxes more user friendly.

6.1 Modeless Dialog Box

Modal and Modeless Dialog Box

There are two types of dialog boxes in Windows(system: modal dialog box and modeless dialog box.

A modal dialog box does not allow the user to switch away from it after it is invoked. We must first dismiss the dialog box before switching to any other window.

It is very easy to implement a modal dialog box: first create a dialog box template, then derive a new class from class CDialog; next we can use the new class to declare a variable which can be used to call function CDialog::DoModal(). Please note that when deriving the new class, we must make sure that it contains the ID of the dialog template.

We can check this by looking at the class definition. By default, there should be a member IDD that is assigned the ID of dialog template:

```
……

//{{AFX_DATA(CMLDialog)
enum { IDD = IDD_DIALOG };

//}}AFX_DATA

......
```

If no ID is assigned to this member, the dialog box will not be created correctly.

A modeless dialog box allows the user to switch to other windows without dismissing it first. Because of this, the variable used to implement the modeless dialog box should not go out of scope in the dialog box's lifetime. Usually we need to use member variable declared in the class to create modeless dialog box rather than using a local variable.

We can not call function CDialog::DoModal() to implement modeless dialog box, because this function is designed solely for modal dialog box. The correct functions that should be used for modeless dialog box are CDialog::Create(...) and CWnd::ShowWindow(...).

The following shows the prototypes of function CDialog::Create(...):

```
BOOL CDialog::Create(UINT nIDTemplate, CWnd* pParentWnd = NULL);

BOOL CDialog::Create(LPCTSTR lpszTemplateName, CWnd* pParentWnd =
NULL);
```

Both versions of this function have two parameters, the first of which is the template ID (either an integer ID or a string ID), the second is a CWnd type pointer which specifies the parent window of the dialog box.

Function CWnd::ShowWindow(...) has the following format:

```
BOOL CWnd::ShowWindow(int nCmdShow);
```

It has only one parameter, which can be set to SW_HIDE, SW_MINIMIZE, SW_SHOW... and so on to display the window in different styles.

For example, if class CMyDialog is derived from CDialog, and the dialog template ID is IDD_DIALOG, we can declare a variable m_dlg in any class (for example, CDocument) then do the following in a member function to implement a modeless dialog box:

```
m_dlg.Create(IDD_DIALOG);
m_dlg.ShowWindow(SW_SHOW);
```

Sample

Sample 6.1\DB demonstrates how to implement modeless dialog box. It is a standard SDI application created by Application Wizard. To make the modeless creation procedure simpler, a member function DoModeless() is implemented in the derived class so that it can be used just like function CDialog:: DoModal().

Please note that when the user clicks "OK" or "Cancel" button to dismiss the dialog box, the window will become hidden rather than being destroyed. The window will be destroyed only when the variable goes out of scope (e.g. when we use delete keyword to release the buffers if they are allocated by new key word, or after function returns if the variable is declared locally). So even after the dialog box is closed by clicking "OK" or "Cancel" button, it still can be restored by calling function CWind::ShowWindow(...) using SW_SHOW parameter.

In the sample application, a dialog template IDD_DIALOG_MODELESS is prepared for modeless dialog box implementation. A new class CMLDialog is derived from CDialog, and a CWnd type pointer m_pParent along with a member function DoModeless() are declared in the class:

(Code omitted)

The constructor of CMLDialog is modified as follows:

```
CMLDialog::CMLDialog(CWnd* pParent /*=NULL*/)

: CDialog(CMLDialog::IDD, pParent)

{

//{{AFX_DATA_INIT(CMLDialog)

//}}AFX_DATA_INIT

m_pParent=pParent;

}
```

When we implement a modal dialog box using class CDialog, the parent window needs to be specified only in the constructor. When calling CDialog::Create(...) to implement a modeless dialog box, we need to specify the parent window again even if it has been passed to the constructor. To let function DoModeless() has the same format with function CDialog::DoModal(), we store the pointer to the parent window in variable CMLDialog::m_pParent so that it can be used later in function DoModeless(). In the sample, function CMLDialog::DoModeless() is implemented as follows:

(Code omitted)

Since this function could be called after the dialog box has been invoked, first we need to check if a valid window has been created by calling function CWnd::GetSafeHwnd(). If the returned value is NULL, the window has not been created yet. In this case, we should call function CDialog::Create(...) to create the window. If the returned value is not NULL, there are two possibilities: the window may be currently active or hidden. We can call function CWnd::IsWindowVisible() to check the dialog box's visual state. If the dialog box is hidden, we should call CWnd::ShowWindow(...) to activate it.

In the sample, a command Dialog Box | Modeless is added to the mainframe menu IDR_MAINFRAME, whose ID is ID_DIALOGBOX_MODELESS. Also, a WM_COMMAND message handler is added for this command through using Class Wizard, and the corresponding member function is CDBDoc::OnDialogboxModeless(). The following is its implementation:

```
void CDBDoc::OnDialogboxModeless()

{

m_dlgModeless.DoModeless();

}
```

With the new class, it is equally easy to implement a modeless or modal dialog box. The only difference between creating two type of dialog boxes is that for modeless dialog box, the variable can not be declared locally.

6.2 Property Sheet

Property sheet provides a very nice user interface, it allows several dialog templates to be integrated together, and the user can switch among them by using tab control. This is especially useful if there are many common controls that need to be included in a single dialog template.

Because property sheet is very similar to dialog box, we can create a dialog box then change it to property sheet. The reason for creating property sheet this way is because currently Developer Studio does not support direct implementation of property sheet.

In MFC, there are two classes that should be used to implement property sheet: CPropertySheet and CPropertyPage. The former class is used to create a frame window that contains tab control, the second class is used to implement each single page.

To implement a property sheet, we first need to derive a class from CPropertySheet, then declare one or more CPropertyPage type member variables within it. Each variable will be associated with a dialog template. Because CPropertyPage is derived from class CDialog, all the public and protected members of CDialog are accessible in the member functions of CPropertyPage.

Sample 6.1-1\DB demonstrates how to create application based on property sheet. First it is generated as a dialog based application by using Application Wizard (the default classes are CDBApp and CDBDlg), then the base class of CDBDlg is changed from CDialog to CPropertySheet. Since class CPropertySheet does not have member IDD to store the dialog template ID, we need to delete the following line from class CDBDlg:

```
enum { IDD = IDD_DIALOG_DB };
```

The default dialog box template IDD_DIALOG_DB will not be used, so it is also deleted from the application resources. The following is the modified class:

(Code omitted)

We also need to find all the keyword CDialog in the implementation file of CDBDlg and change them to CPropertySheet. The changes should happen in the following functions: the constructor of CDBDlg, function DoDataExchange(...), OnInitDialog(), OnSysCommand(...), OnPaint(...), and message mapping macros.

Next we need to create each single page. The procedure of creating a property page is the same with creating a dialog box, except that when adding new class for a dialog box template, we must derive it from class CPropertyPage. In the sample, three dialog templates are added to the application, their IDs are ID_DIALOG_PAGE1, ID_DIALOG_PAGE2 and ID_DIALOG_PAGE3 respectively. Three classes CPage1, CPage2 and CPage3 are also added through using Class Wizard, which are all derived from CPropertyPage. When doing this, we need to provide the ID of the corresponding dialog template.

In class CDBDlg, a new member variable is declared for each page:

(Code omitted)

The pages should be added in the constructor ofCPropertySheet by calling function CPropertySheet::AddPage(...). The following is how each page is added in the sample:

(Code omitted)

Function CPropertySheet::AddPage(...) has only one parameter, it is a pointer to CPropertyPage type object.

These are the necessary steps for implementing property sheet. For each property page, we can also add message handlers for the controls, the procedure of which is the same with that of a standalone dialog box.

By default, the property sheet will be implemented in "tab" mode: there will be a tab control in the property sheet, which can be used to select property pages. The property sheet can also be implemented in "wizard" mode, in which case tab control will be replaced by two buttons (labeled with "Previous" and "Next"). In this mode, the pages can only be selected sequentially through button clickings.

To enable wizard mode, all we need to do is calling function CPropertySheet:: SetWizardMode()after all the pages have been added. For example, if we want to enable wizard mode in the sample, we should implement the constructor of CDBDlg as follows:

(Code omitted)

Sample 6.2-2\DB is the same with sample 6.2-1\DB, except that the property sheet is implemented in wizard mode.

If we need to implement a property sheet dialog box in an SDI or MDI application, most of the steps are still the same. We can start by creating a new CPropertySheet based class, then adding dialog templates and CPropertyPage based classes, using them to declare new variables in CPropertySheet derived class, calling function CPropertySheet::AddPage(...) in its constructor. We can call function CPropertySheet::DoModal() at anytime to invoke the property sheet.

6.3 Modeless Property Sheet

Because property sheet is very similar to dialog box, implementation of modeless property sheet is also similar to that of modeless dialog box: when invoking the property sheet dialog box, instead of calling function

CPropertySheet::DoModal(), we need to call CPropertySheet::Create(...) and CWnd:: ShwoWindow(...). We can use exactly the same method discussed in section 6.1 to implement modeless property sheet.

Sample 6.3\DB demonstrates how to implement modeless property sheet. It is a standard SDI application generated by Application Wizard. A command Property Sheet | Modeless is added to mainframe menu IDR_MAINFRAME, whose ID is ID_PROPERTYSHEET_MODELESS. A WM_COMMAND message handler is also added for this command, and the corresponding function is CDBDoc::OnPropertysheetModeless().

A new class CMLPropertySheet is defined to implement modeless property sheet, whose base class is CPropertySheet. Like what we did in sample 6.1\DB, function DoModeless() is declared in class CMLPropertySheet, which can be used to invoke the property sheet.

Three dialog box templates IDD_DIALOG_PAGE1, IDD_DIALOG_PAGE2 and IDD_DIALOG_PAGE3 are created to implement property pages. Also three new classes CPage1, CPage2 and CPage3 are derived from CPropertyPage. A CWnd type pointer m_pParentWnd and three other member variables declared by CPage1, CPage2 and CPage3 are added to class CMLPropertySheet. The following is the modified class:

(Code omitted)

In the constructor of CMLPropertySheet, we need to store the address of parent window to m_pParentWnd and add the property pages. The constructor of CPropertySheet has two versions, and we need to override both of them:

(Code omitted)

Next, function CMLPropertySheet::DoModeless() is implemented as follows:

(Code omitted)

Everything is the same with that of sample 6.1\DB, except that here we need to call function CPropertySheet::Create(...) instead of CDialog::Create(...).

By now class CMLPropertySheet is ready for use. We can declare a CMLPropertySheet type pointer in class CDBDoc:

(Code omitted)

The constructor and destructor of class CDBDoc are modified to initialize and release the buffers if necessary:

(Code omitted)

It is possible that the variable is not initialized when the application is closed, so we need to check if m_ptrDlg is NULL before deleting it.

Finally, function CDBDoc::OnPropertysheetModeless() is implemented as follows:

(Code omitted)

If m_ptrDlg is NULL, we need to initialize it. Then we call CMLPropertySheet::DoModless() each time the Property Sheet | Modeless command is executed. This member function will take care everything so there is no need for us to check the current state of the property sheet and implement different operations.

6.4 Sizes

In this section we are going to discuss some window sizes that is important for dialog boxes.

Initial Size

The initial size is the dimension of a dialog box when it first pops up. By default, a dialog box's initial size is determined from the font used by the dialog box and the size of its dialog template. If we want to make change to its initial size, we can call either CWnd::SetWindowPos(...) or CWnd::MoveWindow(...) within function CDialog::OnInitDialog(). The difference between above two functions is that CWnd:: SetWindowPos(...) allows us to change a window's X-Y position and Z-order, while CWnd::MoveWindow(...) allows us to move the window only in the X-Y plane.

Dialog Box Unit

When creating a dialog template, we can read its dimension in the status bar of Developer Studio (Figure 6-1). However this size is measured in dialog box unit rather than screen pixels. This means if we create a dialog template with a size of 100(100 (measured in dialog box unit), its actual size will not be 100 pixel (100 pixel. For any dialog template, its horizontal base unit is equal to the average width of the characters that is used by the template, and its vertical base unit is equal to the height of the font. The dialog box is measured by the base units: each horizontal base unit is equal to 4 horizontal dialog units and each vertical base unit is equal to 8 vertical dialog units. This is a very complex calculation, fortunately in class CDialog there is a member function CDialog::MapDialogRect(...) that can be used to implement the dimension conversion so we do not need to calculate the details.

If we want the initial size of a dialog box to be exactly the same with its template size, we need to call function CDialog::MapDialogRect(...) to convert its template size to screen pixels then call CWnd:: MoveWindow(...) to resize the dialog box before it is displayed.

Tracking Size and Maximized Size

There are two types of tracking sizes: minimum tracking size and maximum tracking size, which correspond to limit sizes that can be set to a window by dragging one of its resizable border. The maximized size of a window is the size when it is in the maximized state (when a window is maximized, it doesn't have to take up the whole screen). There is no "minimized size" here because when a window is minimized, it will become an icon.

These sizes can all be customized. To provide user defined sizes, we can override function CWnd:: OnGetMinMaxInfo(...), which will be called when any of the above sizes is needed by the system. We can provide our own sizes within the overridden function.

Function CWnd::OnGetMinMaxInfo(...) has the following format:

```
afx_msg void CWnd::OnGetMinMaxInfo(MINMAXINFO *lpMMI);
```

It is the handler of WM_GETMINMAXINFO message.

Whenever the system needs to know the tracking sizes or maximized size of a window, it sends a WM_GETMINMAXINFO message to it. In MFC, this message is handled by function CWnd:: OnGetMinMaxInfo(...). The input parameter of this function is a MINMAXINFO type pointer, if we want to customize the default implementation, we can change the members of MINMAXINFO. Structure MINMAXINFO is defined as follows:

```
typedef struct tagMINMAXINFO {

POINT ptReserved;

POINT ptMaxSize;

POINT ptMaxPosition;

POINT ptMinTrackSize;
POINT ptMaxTrackSize;

} MINMAXINFO;
```

Here members ptMinTrackSize and ptMaxTrackSize specify the minimum and maximum tracking sizes, ptMaxSize specifies maximized size, and ptMaxPosition specifies the upper-left corner position of a window when it is first maximized.

Sample

Sample 6.4\DB demonstrates how to customize these sizes. It is a dialog based application created from Application Wizard. In the sample, the dialog's minimum tracking size is set to its dialog template size. Also, the maximum tracking size and the maximized size are customized.

To let the dialog box be able to maximize and minimize, we must set the two styles: "Minimize Box", "Maximize Box". To let it be able to resize, we must also set "Resizing" style (Figure 6-2).

In the sample, message handler of WM_GETMINMAXINFO is added through using Class Wizard. The function is implemented as follows:

(Code omitted)

Here MIN_X_SIZE and MIN_Y_SIZE are defined as the dialog template size that is read from Developer Studio when the dialog resource is being edited. Because this size is the client area size of the dialog box (when a dialog box is created, caption bar, borders will be added), we need to add the dimensions of caption bar and border in order to make the dialog size exactly the same with its template size. The dimensions of caption bar and border can be retrieved by calling API function ::GetSystemMetrics(...) with appropriate parameters passed to it. This function allows us to retrieve many system configuration settings. The following is the function prototype and a list of commonly used parameters:

```
int ::GetSystemMetrics(int nIndex);
```

(Table omitted)

In the sample, the maximized size of the dialog is set to 1/4 of the desk top screen size. When the application is first maximized, it will be positioned at top-left corner (0, 0).

The dialog box's initial size is set in function CDialog::OnInitDialog():

(Code omitted)

The dialog box's initial size is a little bigger than its minimum tracking size.

The above sizes are not unique to dialog boxes. In fact, any window has the above sizes, and can be customized with the same method.

6.5 Customizing Dialog Box Background

Background Drawing

Generally all dialog boxes have a gray background. Sometimes it is more desirable to change dialog box's background to a custom color, or, we may want to paint the background using a repeated pattern. To customize a dialog box's background, we need to handle message WM_ERASEBKGND, and draw the custom background after receiving this message. All classes that are derived from CWnd will inherit function CWnd::OnEraseBkgnd(...), which has the following format:

```
BOOL CWnd::OnEraseBkgnd(CDC *pDC)

{

}
```

Here, pointer pDC can be used to draw anything on the target window. For example, we can create solid brush and paint the background with a custom color, or we can create pattern brush, and paint the background with certain pattern. Of course, bitmap can also be used here: we can draw our own bitmap repeatedly until all of the dialog box area is covered by the bitmap patterns.

Sample

Sampel 6.5\DB demonstrates background customization. It is a standard dialog-based application generated from Application Wizard. In the sample, instead of using a uniform color, the dialog box paints its background with a bitmap image (Figure 6-3).

Because WM_ERASEBKGND is not listed as a dialog box message, first we need to customize the filter settings for this application. We can do this by invoking Class Wizard, clicking "Class Info" tab then changing the default setting in combo box "Message Filter" from "Dialog" to "Window". By going back to "Message maps" page now, we can find WM_ERASEBKGND in "Message" window, and add a message handler for it. The function name should be CDBDlg::OnEraseBkgnd(...).

In the sample, a bitmap resource IDB_BITMAP_DLGBGD is added to the application, which will be used to draw the background of the dialog box. In function CDBDlg::OnEraseBkgnd(...), this bitmap is painted repeatedly until all dialog box area is covered by it:

(Code omitted)

First function CBitmap::LoadBitmap(...) is called to load the bitmap resource, then its dimension is retrieved by calling function CBitmap::GetBitmap(...). Next, function CWnd::GetClientRect(...) is called to obtain the size of the client area of the dialog box. Then we calculate the number of loops required to repeat drawing in both horizontal and vertical directions in order to cover all the client area. The results are stored in two local variables nHor and nVer. Then, a memory DC is created, and the bitmap image is selected into this DC. Next, function CDC::BitBlt(...) is called enough times to paint the bitmap to different locations of the dialog box. Finally a TRUE value is returned to prevent the background from being updated by the default implementation.

Changing the Background of Common Controls

If the dialog box includes some other common controls such as edit box, list box, check box or radio button, we will see the undesirable effect: the background of these controls is still painted with the default color, and this makes the appearance of the dialog box not harmonic.

To change the background color of the common controls, we need to handle message WM_CTLCOLOR. The message handler can be added through using Class Wizard, and the default member function looks like the following:

```
HBRUSH CDBDlg::OnCtlColor(CDC *pDC, CWnd *pWnd, UINT nCtlColor)

{

return CDialog::OnCtlColor(pDC, pWnd, nCtlColor);

}
```

This function has three parameters. The first parameter is a pointer to the device context of the target window; the second is a pointer to the common control contained in the dialog box whose background is to be customized; the third parameter specifies the control type, which could be CTLCOLOR_BTN, CTLCOLOR_EDIT, CTLCOLOR_LISTBOX..., indicating that the control is a button, an edit box, a list box, and so on.

We can return a brush handle that can be used to paint the background of the control. We can also let the control to have a transparent background, in this case we must return a NULL brush handle.

In order to demonstrate how to customize the background of the common controls, in the sample, an edit box, a check box, two radio buttons, a static text, a scroll bar, a list box and a simple combo box are added to the application. Also, WM_CTLCOLOR message handler is added and the corresponding function CDBDlg::OnCtlColor(...) is implemented as follows:

(Code omitted)

Stock Objects

In the above implementation, the background of different controls is painted using different brushes: the button and static control have a transparent background; the background of the list box and scroll bar is painted with a gray brush; the background of the message box is painted with a light gray brush. Here, all the brushes are obtained through calling function ::GetStockObject(...) rather than being created by ourselves.

In Windows(, there are a lot of predefined stock objects that can be used. These objects include brushes, pens, fonts and palette. The predefined brushes include white brush, black brush, gray brush, light gray brush, dark gray brush, and null (hollow) brush.

Function ::GetStockObject(...) will return a GDI object handle (a brush handle in our sample). If we attach the returned handle to a GDI object, we must detach it instead of deleting the object when it is no longer useful.

Text Foreground and Background

Now the edit box, static control and list box all have transparent background. But these controls also contain text. Since a character also has both foreground and background areas (Figure 6-4), if we don't set the text's drawing mode, its background area may be drawn with an undesirable color (Figure 6-5).

(Figure 6-4, 6-5 omitted)

We can call function CDC::SetBkMode(...) and use TRANSPARENT flag to set transparent background drawing mode for text, otherwise it will be drawn with the default background color.

The background of a 3-D looking pushdown button can not be changed this way. Also, if we include drop down or drop list combo box, the background color of its list box will not be customized by this method because it is not created as the child window of the dialog box. To modify it, we need to derive new class from CComboBox and override its OnCtlColor(...) member function.

6.6 Resizing the Form View

Form view is very similar to a dialog box. Usually we create form view from a dialog box template, which can contain all the standard common controls. While they are similar, a form view is usually created with a document/view structure, and has some properties that a standard dialog box lacks. For example, a form view will be automatically implemented with scroll bars. If the window size becomes smaller than the size of the dialog template, scroll bars will automatically be activated. They can be scrolled to allow the user to see the hidden part of the dialog.

Since a form view is usually resizable, we sometimes need to move and resize the common controls contained in the form view to make its appearance well balanced. For example, if we have an edit box embedded in the form view, instead of fixing its size, we may want to adjust it dynamically according to the dimension of the form view. This is usually a desired feature of form view because it will make the controls and the window well balanced.

Coordinates Conversion

Every window can be moved and resized by calling function CWnd::MoveWindow(...) or CWnd:: SetWindowPos(...). Also, a window's size and position can be retrieved by calling function CWnd:: GetClientRect(...) and CWnd::GetWindowRect(...). The points retrieved using the former function are measured in the client window's coordinate system, and the points retrieved from the latter function are measured in the screen (desktop) coordinate system. To convert coordinates from one system to another, we can call function CWnd::MapWindowPoints(...) or CWnd::ScreenToClient(...).

For example, if there are two windows: window A and window B, which are attached two CWnd type variables wndA and wndB. If we want to know the size and position of window A measured in window B's coordinate system, we can first obtain the size and position of window A in its local coordinate system, and then convert them to window B system as follows:

```
wndA.GetClientRect(rect);

wndA.MapWindowPoints(&wndB, rect);
```

Or we can find the position and size of window A in the screen coordinate system, and call CWnd:: ScreenToClient(...) to convert them to window B's coordinate system:

```
wndA.GetWindowRect(rect);

wndB.ScreenToClient(rect);
```

Sample

When the user resizes a window, a WM_SIZE message will be sent to that window. We can handle this message to resize and move the controls contained in the dialog template.

Sample 6.6\DB demonstrates how to resize the common controls contained in the form view dynamically. It is a standard SDI application generated from the Application Wizard. When generating the application, CFormView is selected as the base class of the view in the last step. After the application is generated, the following controls are added to the dialog template: an edit box, a static group control, two buttons. The IDs of these controls are IDC_EDIT, IDC_STATIC_GRP, IDC_BUTTON_A and IDC_BUTTON_B respectively.

If we compile and execute the application at this point, the application will behave awkwardly because if we resize the window, the sizes/positions of the controls will not change, this may make the window not well balanced (Figure 6-6).

We need to remember the original sizes and positions of the embedded controls and base their new sizes and positions on them. In the sample, four CRect type variables are declared in class CDBView for this purpose:

(Code omitted)

Also, a Boolean type variable m_bSizeAvailable is added to indicate if the original positions and sizes of the controls have been recorded.

There is no OnInitDialog() member function for class CFormView. The similar one is CView:: OnInitialUpdate(). This function is called when the view is first created and is about to be displayed. We can record the positions and sizes of the controls in this function.

Variable m_bSizeAvailable is initialized to FALSE in the constructor of class CDBView:

```
CDBView::CDBView()
```

```
: CFormView(CDBView::IDD)

{

//{{AFX_DATA_INIT(CDBView)

//}}AFX_DATA_INIT

m_bSizeAvailable=FALSE;

}
```

Member function OnInitialUpdate() can be added to class CDBView through using Class Wizard. In the sample, this function is implemented as follows:

(Code omitted)

Here we call function CWnd::GetWindowRect(...) and CWnd::ScreenToClient(...) several times to retrieve the sizes and positions of all the controls in the dialog template.

The handler of message WM_SIZE can also be added through using Class Wizard. The following is the implementation of this member function in the sample:

(Code omitted)

The new horizontal and vertical sizes of the client window (CDBView) is passed through parameters cx and cy. First we create a rectangle whose dimension is equal to the dimension of the dialog template. Then we compare its horizontal size to cx, and vertical size to cy. If cx is greater than the template's horizontal size, we move button A and button B in the horizontal direction, increase the horizontal size of edit box and static group control. If cx is not greater than the template's horizontal size, we put button A and button B to their original positions, set the horizontal sizes of edit box and static group control to their initial horizontal sizes (this is why we need to know each control's initial size and position). The same thing is done for vertical sizes. Finally, function CWnd::MoveWindow(...) is called to carry out the resize and reposition.

With the above implementation, the form view will have a well balanced appearance all the time.

6.7 Tool Tips

Tool Tip Implementation

Tool tip is a very nice feature, it gives the user quick hint on the functionality of a control. In MFC, tool bar is implemented with automatic tool tip feature: if we add a string whose ID is the same with a control's ID, that string will be used to implement the tool tip for that control. For dialog box, we also want the tool tip to be implemented in a similar way.

In a dialog box, all the controls (except static ones) can be enabled to display tool tips. The procedure of enabling tool tips is very simple: first call function CWnd::EnableToolTips(...) in the dialog box's initialization stage, then handle message TTN_NEEDTEXT. This message is sent to obtain a tool tip text for a specific control. The message handler has the following format:

```
OnToolTipNotify(UINT id, NMHDR *pNMHDR, LRESULT *pResult);

{

TOOLTIPTEXT *pTTT = (TOOLTIPTEXT *)pNMHDR;

UINT nID =pNMHDR->idFrom;

……

}
```

Here the first parameter id indicates the window that sent this notification, which is useless to us. The second parameter is a NMHDR type pointer, which must be cast to TOOLTIPTEXT type in order to process a tool tip notification. Structure TOOLTIPTEXT has the following format:

```
typedef struct {

NMHDR hdr;

LPTSTR lpszText;

WCHAR szText[80];

HINSTANCE hinst;

UINT uflags;

} TOOLTIPTEXT, FAR *LPTOOLTIPTEXT;
```

The ID of the target control (whose tool tip text is being retrieved) can be obtained from member hdr. From this ID we can obtain the resource string that is prepared for the tool tip. There are three ways to provide a tool tip string: 1) Prepare our own buffer that contains the tool tip text and assign the buffer's address to member lpszText. 2) Copy the tool tip text directly to member szText. 3) Stores the tool tip text in a string resource, assign its ID to member lpszText. In the last case, we need to assign member hinst the instance handle of the application, which can be obtained from function AfxGetResourceHandle(). Member uflsgs indicates if the control is a window or not.

Recall when we create tool bars and dialog bars in the first chapter, tool tips were all implemented in a very simple way: we provide a string resource whose ID is exactly the same with the control ID, and everything else will be handled automatically. When handling message TOOLTIPTEXT for dialog box, we can also let the tool tip be implemented in a similar way. In order to do this, we can assign the resource ID of the control to member lpszText and the application instance handle to member hinst. If there exists a string resource whose ID is the same with the control ID, that string will be used to implement the tool tip. Otherwise, nothing will be displayed because the string can not be found.

Sample

Sample 6.7\DB demonstrates how to implement tool tips for the controls contained in a dialog box. It is based on sample 6.6\DB, with tool tips enabled for the following three controls: ID_EDIT, ID_BUTTON_A and ID_BUTTON_B (Although sample 6.6\DB is a form view based application, the tool tip implementation is the same with that of a dialog box).

Three string resources are added to the application, whose IDs are IDC_EDIT, IDC_BUTTON_A and IDC_BUTTON_B. They will be used to implement tool tips for the corresponding edit box and buttons. In function CDBView::OnInitialUpdate(), the tool tips are enabled as follows

```
void CDBView::OnInitialUpdate()

{

......

EnableToolTips(TRUE);

}
```

Message handler of TTN_NEEDTEXT must be added manually. First we need to declare a member function OnToolTipNotify() in class CDBView:

(Code omitted)

Then, message mapping macros should be added to the implementation file:

```
BEGIN_MESSAGE_MAP(CDBView, CFormView)

……

ON_NOTIFY_EX(TTN_NEEDTEXT, 0, OnToolTipNotify)

END_MESSAGE_MAP()
```

Here, TTN_NEEDTEXT is sent through message WM_NOTIFY. Macro ON_NOTIFY_EX allows more than one object to process the specified message. If we use this macro, our message handler must return TRUE if the message is processed. If we do not process the message, we must return FALSE so that other objects can continue to process this message. Please note that in the above message mapping, the second parameter should always be 0.

Member function CDBView::OnToolTipNotify(...) is implemented as follows:

(Code omitted)

First the ID of the control is obtained. If the control is a window, this ID will be a valid handle. We can retrieve the control's resource ID by calling fucntion ::GetDlgCtrlID(...). Next, the resource ID is assigned to member lpszText, and the application's instance handle is assigned to member hinst.

With this method, we can only implement a tool tip which contains maximum of 80 characters. To implement longer tool tips, we need to provide our own buffer and assign its address to member lpszText. In this case, we do not need to assign the application's instance handle to member hinst.

After adding the above implementation, we can just add string resources whose IDs are the same with the resource IDs of the controls. By doing this, the tool tip will automatically implemented for them.

6.8 Tool Bar and Status Bar in Dialog Box

By default, the dialog box does not support tool bar and status bar implementation. Because a dialog box can contain various intuitive controls, it is often not necessary to implement extra tool bar and status bar. But sometimes the tool bar and status bar are helpful, especially when we want to implement a row of buttons with the same size. In this case, we can also easily implement the tool tips and flybys on the status bar.

Frame Window

In a standard SDI or MDI application, tool bar and status bar can be implemented by declaring CToolBar and CStatusBar type variables in class CMainFrame (They will be created in function CMainFrame::OnCreate(...)). In a dialog-based application, the frame window is the dialog box itself, so we need to embed CToolBar and CStatusBar type variables in the CDialog derived class and create them in function CDialog::OnInitDialog().

However, unlike CFrameWnd, class CDialog is not specially designed to work together with status bar and tool bar, so it lacks some features that are owned by class CFrameWnd: first, it does not support automatic tool tip implementation, so we have to write TTN_NEEDTEXT message handler for displaying tool tips; second, it does not support flyby implementation, so we also need to add other message handlers in order to enable flybys.

Flyby Related Messages

In MFC, there are two un-documented messages that are used for flyby implementation. When a flyby text for certain control needs to be displayed, the frame window will receive message WM_SETMESSAGESTRING. Also, when a flyby needs to be removed, the frame window will receive another message: WM_POPMESSAGESTRING.

Tool Bar Resource

Sample 6.8-1\DB demonstrates how to implement tool bar and status bar in a dialog based application. The sample is generated by Application Wizard, with a tool bar resource IDD_DB_DIALOG added later on. This ID is the same with the dialog template ID, which is convenient for tool bar implementation.

The tool bar contains four buttons, whose IDs are ID_BUTTON_YELLOW, ID_BUTTON_GREEN, ID_BUTTON_RED and ID_BUTTON_BLUE, and they are painted with yellow, green, red and blue colors respectively. Four string resources are also added to the application, they will be used to implement tool tips and flybys:

(Table omitted)

The sub-string before character '\n' will be used to implement flyby, and the sub-string after that will be used to implement tool tip. We will see that by letting the control and the string resource share a same ID, it is easier for us to implement both flybys and tool tips.

New CToolBar and CStatusBar type variables are declared in class CDBDlg:

```
class CDBDlg : public CDialog
{
......
protected:
......
CStatusBar m_wndStatusBar;
CToolBar m_wndToolBar;
......
};
```

Status Bar

A status bar is divided into several panes, each pane displays a different type of information. We can create as many panes as we like. When implementing a status bar, we must provide each pane with an ID. We can use these IDs to access each individual pane, and output text or graphic objects. Usually these IDs are stored in a global integer array. In the sample, the following array is declared for the status bar:

```
static UINT indicators[] =
{
AFX_IDS_IDLEMESSAGE,
IDS_MESSAGE
};
```

The status bar will have only two panes. Usually the first pane of the status bar is used to display flybys (In the idle state, "Ready" will be displayed in it). One property of status bar is that if we implement a string resource whose ID is the same with the ID of a pane contained in a status bar, the string will be automatically displayed in it when the application is idle. So here we can add two string resources whose IDs are AFX_IDS_IDLEMESSAGE and IDS_MESSAGE respectively. Since Developer Studio does not allow us to add a string resource starting with "AFX_", we may copy this string resource from any standard SDI application (An SDI application has string resource AFX_IDS_IDLEMESSAGE if it is generated by Application Wizard).

Adding Control Bars to Dialog Box

In function CDBDlg::OnInitDialog(), the following code is added for creating both tool bar and status bar:

(Code omitted)

Here, the procedure of creating the tool bar and status bar is almost the same with what we need to do for a standard SDI and MDI application in function CMainFrame::OnCreate(). The difference is that when implementing them in a dialog box, there is no need to set docking/floating properties for the control bars.

Function CWnd::RepositionBars(...) is also called to calculate the position of control bars then and reposition them according to the dimension of the client area. If we do not call this function, the status bar and tool bar may be randomly placed and thus can not be seen. When calling this function, we can use AFX_IDW_CNTROLBAR_FIRST and AFX_IDW_CONTROLBAR_LAST instead of providing actual IDs of the control bars.

Resizing the Client Area to Accommodate Control Bars

By compiling the and executing the application at this point, we will see that the dialog box is implemented with a tool bar and a status bar. The problem is: they occupy the client area without resizing the dialog box. If a control happens to be placed to the top or the bottom of the dialog template, it might overlap one of the control bars.

To solve this problem, we need to resize the dialog box and move the common controls to leave room for both tool bar and status bar.

Function CWnd::RepositionBars(...) has the following format:

```
void CWnd::RepositionBars

(
```

UINT nIDFirst, UINT nIDLast, UINT nIDLeftOver, UINT nFlag=CWnd::reposDefault,

LPRECT lpRectParam=NULL, LPCRECT lpRectClient=NULL

);

The function has six parameters, among them, nFlag, lprectParam and lpRectClient all have default values. When we called this function in the previous step, all the default values were used. This will pass CWnd::reposDefault to parameter nFlag, which will cause the default layout to be performed. If we pass CWnd::reposQuery to parameter nFlag, we can prepare a CRect type object and pass its address to lpRectParam to receive the new client area dimension (The client area is calculated with the consideration of control bars, their sizes are deducted from the original dimension of the client area). This operation will not let the layout be actually carried out. Based on the retrieved size, we can adjust the size of the dialog box and move the controls so that we can leave enough room to accommodate the tool bar and the status bar.

The following is the updated implementation of function CDBDlg::OnInitDialog():

(Code omitted)

First function CWnd::GetClientRect() is called and the dimension of client window is stored in rectOld. After the control bars are created, we call function CWnd::RepositionBars(...) and use flag CWnd::reposQuery to obtain the new layout dimension with the consideration of two control bars. The new layout dimension is stored in variable rectNew. The offset is calculated by deducting rectOld from rectNew. To access all the controls in the dialog box, we first call CWnd::GetWindow(...) using GW_CHILD flag to obtain the first child window of the dialog box, then call CWnd::GetNextWindow() repeatedly to find all the other child windows. Each child window is moved according to the offset dimension. Finally the dialog box is resized by calling function CWnd::RepositionBars(...) using the default parameters.

Tool Tip and Flyby Implementation

We need to add message handler for TTN_NEEDTEXT notification in order to implement tool tips. Also, we need to handle WM_SETMESSAGESTRING and WM_POPMESSAGESTRING in order to implement flybys on the status bar.

In the sample, three new message handlers are added to class CDBDlg:

(Code omitted)

Function CDBDlg::OnTooltipText(...) will be used to handle TTN_NEEDTEXT message, CDBDlg:: OnSetMessageString(...) and CDBDlg::OnPopMessageString(...) will be used to handle WM_SETMESSAGESTRING and WM_POPMESSAGESTRING respectively.

Message mapping macros are added to the implementation file as follows:

(Code omitted)

First the target window handle is obtained from idFrom member of structure NMHDR, and the ID of the button is retrieved by calling function ::GetDlgCtrlID(...). Then a string with the same ID is loaded from the resource into a CString type variable, and the sub-string after the '\n' character is copied into szText member of structure TOOLTIPTEXT. The sub-string before this character will be used to implement flyby.

The rest two functions are implemented as follows:

(Code omitted)

The control ID is sent through WPAMAM parameter of the message, so in the first function, we just use this ID to load a string from the resource, and display it in the first pane of the status bar by calling function CStatusBar::SetPaneText(...). For the second function, if there is no pop up message, we just return 0. Otherwise we call the first function to display an appropriate message.

Implementing Control Bars for Dialog Boxes Implemented in SDI or MDI Applications

We may think that by using the form view, it would be much easier for us to add tool bar and status bar to dialog box. So what's the meaning to implement them by ourselves? First a form view based application is implemented with document/view structure, and a pure dialog based application has fewer classes (only CWinApp and CDialog). Second, if we have a dialog box implemented in an SDI or MDI application, it is difficult for us to implement it as a form view.

Sample 6.8-2\DB demonstrates how to implement control bars for a dialog box contained in an SDI or MDI application. It is a standard SDI application generated by Application Wizard. First a dialog template IDD_DIALOG is added to the application, and a new CDialog based class CBarDialog is created through using Class Wizard. Like what we did in sample 6.8-1\DB, variables m_wndStatusBar and m_wndToolBar are declared in class CBarDialog, function CBarDialog::OnInitDialog() is modified to create the status bar and tool bar, and three message handlers are implemented for TTN_NEEDTEXT, WM_SETMESSAGESTRING and WM_POPMESSAGESTRING.

A new command Dialog | Bar Dialog is added to mainframe menu IDR_MAINFRM, whose command ID is ID_DIALOG_BARDIALOG. Its WM_COMMAND message handler can be added through using Class Wizard, in the sample, this handler is CDBDoc::OnDialogBardialog().

Function CDBDoc::OnDialogBardialog() can be implemented as follows:

```
void CDBDoc::OnDialogBardialog()

{

CBarDialog dlg;

dlg.DoModal();

}
```

If we do this, the tool bar and the status bar will be added to the dialog box. Also, the tool tips will work. However, there will be no flyby displayed in the status bar.

Problem

The reason for this is that when the application tries to display a flyby on the status bar, it will always try to put it on the status bar of the top-most parent window, which is the mainframe window in an SDI or MDI application. If the top-most window is inactive, the flyby will not be displayed.

When we invoke a modal dialog box, the mainframe window will always be inactivated. This is the reason why the flyby will be displayed neither on the status bar of the dialog box nor on the status bar of the mainframe window.

Work Around

One fix to this problem is to override the member function that is used to display the flyby. If we study the source code of class CControlBar, we will find that the flyby display is implemented as follows: after the mouse cursor enters the tool bar, a timer with time out period of 300 millisecond will be started. When this timer times out, the application checks to see if the cursor is still within the tool bar. If so, it kills the timer, starts another timer with a time out period of 200 millisecond. Next, it finds out the ID of the control that is under the mouse cursor and sends WM_SETMESSAGESTRING message to the mainframe window (if it is active).

The time out event is handled by function CControlBar::OnTimer(), we can override it and send WM_SETMESSAGESTRING message to the dialog box window.

The following is the original implementation of function CControlBar::OnTimer():

(Code omitted)

Note that in the above code fragment, function CWnd::GetTopLevelParent() is used to obtain the window where flyby should be displayed. If we replace it with CWnd::GetParent(), everything will be fixed.

Overriding CToolBar::OnTimer(...)

In the sample a new class CDlgToolBar is added to the application, whose base class is CToolBar. Within the new class, function OnTimer(...) is declared to override the default implementation:

(Code omitted)

Function CDlgToolBar::OnTimer(...) is implemented as follows:

(Code omitted)

This is just a copy of function CControlBar::OnTimer(...), except that here function CWnd:: GetTopLevelParent() is replaced by CWnd::GetParent().

An Alternate Solution

But this is not the best solution. Because the current implementation of function CControlBar:: OnTimer(...) is not guaranteed to remain unchanged in the future, there is a possibility that the above implementation will become incompatible with future versions of MFC.

The best solution is to set our own timer and simulate the default behavior of control bar. We can bypass all the implementation in the base class and set up our own 300 millisecond timer when the mouse cursor first enters the tool bar. When this timer times out, we check if the cursor is still within the tool bar. If so, we kill the timer and set another 200 millisecond timer. Whenever the timer times out, we check the position of mouse cursor and send WM_SETMESSAGESTRING message to the window that contains the tool bar.

Sample 6.8-3\DB demonstrates this method. It is based on sample 6.8-2\DB, with the WM_TIMER message handler removed from the application.

To detect mouse movement, we need to override function CWnd::PreTranslateMessage(...). Also, two timer IDs are defined to set timers:

```
#define ID_TIMER_DLGCHECK 500

#define ID_TIMER_DLGWAIT 501
```

The above IDs can be any integers. A Boolean type variable m_bTimerOn is declared in class CDlgToolBar. It will be used to indicate if the timer is currently enabled or not.

Variable m_bTimerOn is initialized in the constructor:

```
CDlgToolBar::CDlgToolBar():CToolBar()

{

m_bTimerOn=FALSE;

}
```

Function CDlgToolDar::PreTranslateMessage(...) is implemented as follows:

(Code omitted)

If the message is WM_MOUSEMOVE and the timer is off, this indicates that the mouse cursor has just entered the tool bar. We need to set timer ID_TIMER_DLGWAIT. Also, we need to set flag m_bTimerOn to TRUE.

Function CDlgToolBar::OnTimer(...) is implemented as follows:

(Code omitted)

We call function ::GetCursorPos(...) to retrieve the current mouse cursor position, then call function CWnd::ScreenToClient(...) to convert it to the tool bar coordinate system.

Next we call CWnd:: OnToolHitTest(...) to obtain the control ID of the button, then send WM_SETMESSAGESTRING message to the parent of the control bar. In case the current timer is ID_TIMER_DLGWAIT, we kill it and set timer ID_TIMER_DLGCHECK with a time out period of 200 millisecond. If the mouse cursor is not within the toolbar, this indicates that it has just been moved outside the control bar. In this case, we need to send message WM_POPMESSAGESTRING to the parent window, then kill the timer.

With this implementation, the flybys will work as if they were implemented in a standard control bar.

Summary:

1) To implement modeless dialog box, we need to declare CDialog type member variable and call CDialog::Create(...) instead of function CDialog::DoModal().

2) When a modeless dialog box is dismissed, the window becomes hidden rather than being destroyed. So if the user invoke the dialog again, we need to call function CWnd::ShowWindow(...) to activate the window rather than create it again.

3) We can decide the visual state of a window by calling function CWnd::IsWindowVisible(...).

4) Property sheet can be implemented as follows: 1) Derive a class from CPropertySheet.

5) Add dialog template for each property page. 3) Implement a CPropertyPage derived class for each dialog template created in step 2). 4) Use the classes created in step 3) to declare variables in the class derived from CPropertySheet. 5) In the constructor of CPropertySheet derived class, call CPropertySheet:: AddPage(...) for each page.

6) A property sheet can have either standard style or wizard style. To enable wizard style, we need to call function CPropertySheet::SetWizardMode().

7) To convert a dialog template dimension (measured in dialog box unit) to its actual screen size (measured in screen pixels), we need to call function CDialog::MapDialogRect(...).

8) Tracking sizes and maximized size of a window can be set by handling message WM_GETMINMAXINFO.

9) The background of a window can be customized by handling message WM_ERASEBKGND.

10) The background of controls contained in a dialog box can be customized by handling message WM_CTLCOLOR. When handling this message, we can provide a NULL (hollow) brush to make the background transparent.

11) Tool tips can be added for controls contained in a dialog box by calling function CWnd::EnableToolTips(...) and handling notification TTN_NEEDTEXT.

12) Tool bar and status bar can also be implemented in a dialog box. We must move the controls contained in the dialog box to accommodate the control bars. Also, we need to handle messages TTN_NEEDTEXT, WM_SETMESSAGESTRING and WM_POPMESSAGESTRING in order to implement tool tips and flybys.

BACK TO INDEX

&nb

Chapter 7 Common Dialog Boxes

Common dialog boxes are very useful in Windows(programming. We can use these dialog boxes to select files, colors, fonts, set up printer, do search and replace. Since these kind of operations are very common for all applications, common dialogs are pre-implemented by the operating system. We do not need to create them from dialog template if we want to use one of these dialog boxes.

7.1 File Open and Save Dialog Box

Implementing a Standard File Open Dialog Box

File dialog box is designed to let user pick up a file name for open, save, or other operations. In MFC, this type of common dialog boxes is implemented by class CFileDialog. The code used to implement a file open dialog box is very simple: we can just declare a CFileDialog type variable, then call function CFileDialog::DoModal() to implement the dialog box:

```
CFileDialog dlg(TRUE);

dlg.DoModal();
```

That's all we need to do. Since class CFileDialog does not have a default constructor, we must pass at least a Boolean type value to the first parameter of its constructor. If this value is TRUE, the dialog box will be an "Open" dialog box, if it is FALSE, the dialog box will be a "Save As" dialog box. Because the dialog template is already implemented by the operating system, we don't even need to design a single button for it. However, with the above simple implementation, what we can create is a very general file open dialog box: it does not have file filter, it does not display default file name, also, the initial directory is always the current working directory.

Structure OPENFILENAME

To customize the default behavior of file dialog box, we need to add extra code. Fortunately, this class is designed so that its properties can be easily changed by the programmer. We can make changes to its default file extension filter, default file name. We can also enable or disable certain controls in the dialog box, or even use our own dialog template.

Class CFileDialog has a very important member variable: m_ofn. It is declared by structure OPENFILENAME, which has the following format:

```
typedef struct tagOFN { // ofn
DWORD lStructSize;
HWND hwndOwner;
HINSTANCE hInstance;
LPCTSTR lpstrFilter;
LPTSTR lpstrCustomFilter;
DWORD nMaxCustFilter;
DWORD nFilterIndex;
LPTSTR lpstrFile;
DWORD nMaxFile;
LPTSTR lpstrFileTitle;
DWORD nMaxFileTitle;
LPCTSTR lpstrInitialDir;
LPCTSTR lpstrTitle;
DWORD Flags;
WORD nFileOffset;
WORD nFileExtension;
LPCTSTR lpstrDefExt;

DWORD lCustData;

LPOFNHOOKPROC lpfnHook;

LPCTSTR lpTemplateName;

} OPENFILENAME;
```

It has 20 members, which can all be used to customize the dialog box. In this and the following sections, we are going to see how to use them to change the default behavior of the file dialog box.

File Extension Filter

One of the most important members in this structure is lpstrFilter, which lets us specify a user defined filter for displaying files. Only those files whose extensions match one of the filters will be displayed in the dialog box. We can specify as many filters as we want. Each filter is made up of two parts: the text description and the filter string. The text description is used to give the user an idea about the type of the files, the filter string usually contains wildcard characters that can be used to specify file types.

For example, if we want to display only bitmap files, the description could be "Bitmap File (*.bmp)" and the filter string should be "*.bmp". Filters are separated by zeros. Within a filter, the description text and the filter string are also separated by a zero. For example, if we want to specify two filters, one is "*.cpp", another is for "*.htm", we should set the filter like this:

```
lpstrFilter="CPP File(*.cpp)\0*.cpp\0HTML File(*.htm)\0*.htm\0";
```

In the above statement, "CPP File(*.cpp)\0*.cpp\0" is the first filter and "HTML File(*.htm) \0*.htm\0" is the second filter.

A filter can select more than one type of files. If we specify this type of filter, the different file types should be separated by a ';' character. For example, in the above example, if we want the first filter to select both "*.cpp" and "*.h" file, its filter string should be "*.cpp;*.h".

Besides the standard filter, we can also specify a custom filter. In the file dialog boxes, the custom filter will always be displayed in the first place of the filter list. To specify a custom filter, we can store the filter string in a buffer, use member lpstrCustomFilter to store the buffer's address, and use member nMaxCustFilter to store the buffer's size.

If we have a list of filters, we can use only one of them at any time. Member nFilterIndex lets us specify which filter will be used as the initial one. Here the index to the first file filter is 1.

Retrieving File Names

After function CFileDialog::DoModal() is called, we can use CFileDialog::GetFileName() or CFileDialog::GetPathName() to retrieve the file name selected by the user. The difference between the two functions is that CFileDialog::GetFileName() returns full file name (including extension), and CFileDialog::GetFilePath() returns full path name (file name plus directory names). There are some other member functions that we can call to retrieve file extension, file title, etc.

File Open

Sample 7.1\CDB demonstrates how to use file dialog box and how to customize its standard styles. It is a standard SDI application generated by Application Wizard. After the application is generated, a new sub- menu File Dialog Box is added to mainframe menu IDR_MAINFRAME between View and Help. Two new commands File Dialog Box | File Open and File Dialog Box | File Save are added to this sub-menu, whose IDs are ID_FILEDIALOGBOX_FILEOPEN and ID_FILEDIALOGBOX_FILESAVE respectively. Two WM_COMMAND message handlers are added to class CCDBDoc (the document class) for the new commands through using Class Wizard, the corresponding function names are CCDBDoc::OnFiledialogboxFileopen() and CCDBDoc::OnFiledialogboxFilesave() respectively.

In the sample, command File Dialog Box | File Open is used to invoke a file open dialog box which has two filters and one custom filter. The message handler is implemented as follows:

(Code omitted)

Before we call function CFileDialog::DoModal(), the default behavior of file dialog box is modified. Here, three file filters are specified: the first filter selects "*.c", "*.cpp", "*.h" and "*.hpp" files; the second filter selects "*.doc" and "*.htm" files; the third filter selects all files. Since "1" is assigned to member nFilterIndex, the filter that is used initially would be "*.C;*.CPP;*.H;*.HPP".

A custom filter is also specified, which will select only files with "*.bmp" extension.

After calling function CFileDialog::DoModal(), if the user has picked up a file and clicked "OK" button, both file name and path name will be displayed in a message box.

File Save

When we ask the user to save a file, there are two more things that should be considered. First, we need to specify a default file name that will be used to save the data. Second, when the user uses "*.*" file filter, we may need to provide a default file extension.

We can specify the default file name by using members lpstrFile and nMaxFile of structure OPENFILENAME. We can store the default file name in a buffer, assign the buffer's address to member lpstrFile and the buffer's size to member nMaxFile. With this implementation, when the file save dialog box is invoked, the default file name will appear in "File Name" edit box. Also, we can use member lpstrDefExt to specify a default file extension. Please note that the maximum size of an extension string is 3 (If the string contains more than 3 characters, only the first three characters will be used).

The following is the WM_COMMAND message handler for command File Dialog Box | File Save:

(Code omitted)

In the sample, the default file name is set to "TestFile" (stored in buffer szFile). Also, default file extension is "DIB". All other settings are the same with the file open dialog box implemented above.

7.2 More Customizations

New Style and Old Style File Dialog Boxes

If we are writing applications for new operating systems such as Window95(, we can implement the file dialog box in two different styles. The new style, also called "Explorer-style" uses list control to display file names. The old style, which is the only available file dialog box in Windows3.1(, uses list box control. Two type of dialog boxes are shown in Figure 7-1.

(Figure 7-1 omitted)

The Explorer style file dialog box can display long file name, the old style dialog box will convert all long file names to 8.3 format (8 characters of file name + dot + 3 characters of extension). By default, class CFileDialog will implement Explorer-style file dialog box. If we want to create old style file dialog box, we must set changes to member Flags of structure OPENFILENAME.

Member Flags is a DWORD type value, which contains many 1-bit flags that can be set or cleared to change the styles of the file dialog box. By default, its OFN_EXPLORER bit is set, and this will let the dialog box have Explorer-style. If we set this bit to 0, the dialog box will be implemented in the old style.

Other Bits of Flags

We can use other bits of Flags to further customize the styles of file dialog box. The following lists three of them:

(Table omitted)

Dialog Box Title

The default titles of file dialog boxes are "File Open" and "Save As". We can change these titles by preparing text string in a buffer and assign its address to lpstrTitle member of structure OPENFILENAME.

Retrieving Multiple Path Names and File Names

If we allow the user to select more than one file, we need to call function CFileDialog:: GetStartPosition() and CFileDialog::GetNextPathName(...) to retrieve path name for each selected file. Here, the first function will return a POSITION type value, which could be used to call the second function. The returned value of the second function is a CString type value, which contains a valid file path name. Also, when calling the second function, the POSITION value will also be updated, which again can be used to get the next selected file path name. If all the selected files have been enumerated, a NULL value will be stored in the variable that holds the POSITION value.

However, there is no similar functions for retrieving all file names. To obtain all the selected file names, we need to access member lpstrFile of structure OPENFILENAME. After the user has made selections, the selected folder and file names will be stored in a buffer pointed by member lpstrFile. For Explorer-style dialog box, folder and file names are separated by '\0' characters; for old style dialog box, they are separated by SPACE character. In either case, folder name is always the first item contained in the buffer, which is followed by separator ('\0' or SPACE), file name, separator, and file name.... The position of the first file name is specified by member nFileOffset.

Sample

Sample 7.2\CDB demonstrates these styles. It is based on sample 7.1\CDB, with two new commands File Dialog Box | Customized File Open and File Dialog Box |

Customize File Open Old added to the application. The IDs of the two commands are ID_FILEDIALOGBOX_CUSTOMIZEDFILEOPEN and ID_FILEDIALOGBOX_CUSTOMIZEFILEOPENOLD respectively. Message handlers are added for them through using Class Wizard, the corresponding member functions are CCDBDoc:: OnFiledialogboxCustomizedfileopen() and CCDBDoc::OnFiledialogboxCustomizefileopenold().

For dialog box invoked by command File Dialog Box | Customized File Open, multiple file selection is enabled. Also, the dialog box has a "Help" button and a "Read only" check box. The following is the implementation of this command:

(Code omitted)

Two flags OFN_ALLOWMULTISELECT and OFN_SHOWHELP are set, this will enable multiple file selection and display the "Help" button. Also, flag OFN_HIDEREADONLY is disabled, this will display "Read only" check box in the dialog box. If the dialog box returns value IDOK (This indicates the user has pressed "OK" button), the first file name is obtained by doing the following:

```
lpstr=dlg.m_ofn.lpstrFile+dlg.m_ofn.nFileOffset;
```

Because the file names are separated by '\0' characters, we can use the following statement to access next file name:

```
lpstr+=strlen(lpstr);
```

Path names are obtained through calling function CFileDialog::GetStartPosition() and CFileDialog::GetNextPathName(...). The selected file and path names will be displayed in a message box.

Command File Dialog Box | Customize File Open Old has the same functionality, except that the dialog box is implemented in the old style. Before the dialog box is invoked, flag OFN_LONGNAMES is disabled, which will convert all long file names to "8.3 format". Because the file names are separated by SPACE rather than '\0' characters, the procedure of obtaining file names is a little different:

(Code omitted)

Instead of checking '\0', SPACE characters are checked between file names. Because SPACE character is not the end of a null-terminated string, we have to calculate the length for each file name.

If we compile and execute the application at this point, the dialog boxes invoked by the two newly added commands should let us select multiple files by using mouse along with CTRL or SHIFT key.

7.3 Selecting Only Directory

Sometimes we want to let the user select directories (folders) rather than files, for example, an installation application will probably ask the user to select the target directory where the files can be copied. In this case, we need to provide an interface for picking up only directories. Although this can be implemented through designing a new dialog box, it is not the best solution.

Sample 7.3\CDB demonstrates how to implement directory selection dialog box. It is based on sample 7.2\CDB, with two new commands added to the application: File Dialog Box | Dir Dialog and File Dialog Box | Dir Dialog Old. The IDs of the two new commands are ID_FILEDIALOGBOX_DIRDIALOG and ID_FILEDIALOGBOX_DIRDIALOGOLD, and their message handlers are CCDBDoc:: OnFiledialogboxDirdialog() and CCDBDoc::OnFiledialogboxDirdialogold(). The two commands will be used to implement directory selection dialog box in new and old styles respectively.

New Style

If we are writing code for Windows 95(or Windows NT4.0(, things become very simple. There are some API shell functions that can be called to implement a "folder selection" dialog box with just few simple steps. We can call function ::SHBrowseForFolder(...) to implement dialog box that let the user select folder, and call function ::SHGetPathFromIDList(...) to retrieve the folder that has been selected by the user.

The following is the prototype of function ::SHBrowseForFolder(...):

```
WINSHELLAPI LPITEMIDLIST WINAPI ::SHBrowseForFolder(LPBROWSEINFO
lpbi);
```

It has only one parameter, which is a pointer to structure BROWSEINFO. The structure lets us set the styles of folder selection dialog box, it has eight members:

```
typedef struct _browseinfo {

HWND hwndOwner;

LPCITEMIDLIST pidlRoot;

LPSTR pszDisplayName;
LPCSTR lpszTitle;

UINT ulFlags;

BFFCALLBACK lpfn;

LPARAM lParam;

int iImage;

} BROWSEINFO, *PBROWSEINFO, *LPBROWSEINFO;
```

Member hwndOwner is the handle of the window that will be the parent of folder selection dialog box, it can be NULL (In this case, the folder selection dialog box does not belong to any window). Member pidlRoot specifies which folder will be treated as "root" folder, if it is NULL, the "desktop" folder is used as the root folder. We must provide a buffer with size of MAX_PATH and assign its address to member pszDisplayName for receiving folder name. Member lpszTitle lets us provide a customized title for folder selection dialog box. Members lpfn, lParam and iImage let us further customize the behavior of dialog box by specifying a user-implemented callback function. Generally we can use the default implementation, in which case these members can be set to NULL. Member ulFlags lets us set the styles of folder selection dialog box. For example, we can enable computer and printer selections by setting BIF_BROWSEFORCOMPUTER and BIF_BROWSEFORPRINTER flags.

Function ::SHBrowseForFolder(...) returns a pointer to ITEMIDLIST type object, which can be passed to function ::SHGetPathFromIDList(...) to retrieve the selected folder name:

```
WINSHELLAPI BOOL WINAPI ::SHGetPathFromIDList(LPCITEMIDLIST pidl, LPSTR
pszPath);
```

We need to pass the pointer returned by function ::SHBrowserForFolder(...) to pidl parameter, then provide our own buffer whose address is passed to parameter pszPath for retrieving directory name.

Because these functions are shell functions, the buffers returned from them can not be released using normal method. Instead, it should be released by shell's task allocator.

Shell's task allocator can be obtained by calling function ::SHGetMalloc(...):

```
HRESULT ::SHGetMalloc(LPMALLOC *ppMalloc);
```

By calling this function, we can access shell's IMalloc interface (a shell's interface that is used to allocate, free and manage memory). We need to pass the address of a pointer to this function, then we can use method IMalloc::Free(...) to released the buffers allocated by the shell.

The following is the implementation of command File Dialog Box | Dir Dialog:

(Code omitted)

First function ::SHGetMalloc(...) is called to retrieve a pointer to shell's lMalloc interface, which will be used to release the memory allocated by the shell. Then, structure BROWSEINFO is filled. Both hwndOwner and pidlRoot are assigned NULL. By doing this, the dialog will have no parent, and the desktop folder will be used as its root folder. The title of dialog box is changed to "Select Directory". For member ulFlags, two bits BIF_RETURNFSANCESTORS and BIF_RETURNONLYFSDIRS are set to 1. This will allow the user to select only file system ancestors and file system directories. The returned buffer's address is stored in pointer pidl, which is passed to function ::SHBrowseForFolder(...) for retrieving the directory name. The selected directory name is stored in buffer szBuf, and is displayed in a message box. Finally, memory allocated by the shell is released by shell's lMalloc interface.

Old Style

If we are writing code for Win32 applications, the above-mentioned method does not work. We must use the old style file dialog box to implement folder selection.

The default dialog box has two list boxes, one is used for displaying directory names, the other for displaying file names. One way to implement directory selection dialog box is to replace the standard dialog template with our own. To avoid any inconsistency, we must include all the controls contained in the standard template in the custom dialog template (with the same resource IDs), hide the controls that we don't want, and override the default class to change its behavior.

To use a user-designed dialog template, we must: 1) Prepare a custom dialog template that has all the standard controls. 2) Set OFN_ENABLETEMPLATE bit for member Flags of structure OPENFILENAME, assign custom dialog template name to member lpTemplateName (If the dialog has an integer ID, we need to use MAKEINTRESOURCE macro to convert it to a string ID). 3) Assign the instance handle of the application to member hInstance, which can be obtained by calling function AfxGetInstanceHandle().

The standard file dialog box has two list boxes that are used to display files and directories, two combo boxes to display drives and file types, an edit box to display file name, a "Read only" check box, several static controls to display text, and "OK", "Cancel" and "Help" buttons. The following table lists their IDs and functions:

(Table omitted)

We must design a dialog template that contains exactly the same controls in order to replace the default template with it. This means that the custom dialog template must have static controls with IDs of 1088, 1089, 1090..., list boxes with IDs of 1120 and 1121, combo boxes with IDs of 1136 and 1137, and so on.

Although we can not delete any control, we can resize the dialog template and change the positions of the controls so that it will fit our use.

To let user select only folders, we need to hide following controls (along with the static control that contains text "Directories"): stc1, stc2, stc3, edt1, lst1, cmb1. We can call function CWnd::ShowWindow(...) and pass SW_HIDE to its parameter in the dialog initialization stage to hide these controls. Figure 7-3 shows the custom dialog template IDD_DIR implemented in the sample.

We must also fill edit box edt1 with a dummy string, because if it is empty or filled with "*.*", the file dialog box will not close when the user presses "OK" button.

By default, clicking on "OK" button will not close the dialog box if the currently highlighted folder is not expanded. In this case, clicking on "OK" button will expand the folder, and second clicking will close the dialog box. To overcome this, we need to call function CFileDialog::OnOK() twice when the "OK" button is clicked, this will close the dialog box under any situation.

We need to override class CFileDialog in order to change its default behavior. In the sample application, new class MCDialogBox is added by Class Wizard, whose base class is selected as CDialogBox. Three new member functions are added: constructor, MCFileDialog::OnOK() and MCFileDialog:: OnInitDialog(). The following is this new class:

(Code omitted)

Like class CFileDialog, the constructor of MCFileDialog has six parameters. The first parameter specifies if the dialog box is an "Open File" or a "Save As" dialog box, the rest parameters specify default file extension, default file name, style flags, filter, and parent window.

The constructor is implemented as follows:

(Code omitted)

Nothing is done in this function except calling the default constructor of the base class. Function MCFileDialog::InitDialog() is implemented as follows:

(Code omitted)

The controls that are useless for picking up folder are set hidden, and edt1 edit box is filled with a dummy string (it can be any string). The initial focus is set to the list box which will be used to display the directories. After calling function OnInitDialog() of base class (this will implement default dialog initialization), the first directory in the list box is highlighted.

The implementation of function MCFileDialog::OnOK() is very simple:

```
void MCFileDialog::OnOK()

{

CFileDialog::OnOK();

CFileDialog::OnOK();

}
```

The standard file dialog template can be found in file "Commdlg.dll", which is usually located under system directory. A copy of this file can be also found under Chap7\.

In the sample, command File Dialog Box | Dir Dialog Old is implemented as follows:

(Code omitted)

Flag OFN_EXPLORER must be disabled in order to implement old style file dialog box. The name of custom dialog template is assigned to member lpTemplateName of structure OPENFILENAME. After calling function MCFileDialog::DoModal(), the directory name is retrieved from member lpstrFile of structure OPENFILENAME.

Since member nFileOffset specifies position where file name starts, the characters before this address is the directory name followed by a SPACE character. We can change SPACE to '\0' character to let the string ends by the directory name.

This method is only available for the current version of Windows(, it may change in the future. Whenever possible, we should use the first method to implement folder selection.

7.4 Adding File Preview

The file dialog box will become more useful if we add a file preview window to it. With this feature, when the user highlights a file, part of file's contents will be displayed in the preview window so the user will have a better knowledge on the file before opening it. This can be easily implemented in the Explorer-style file dialog box because it provides an easy way to let us add extra controls to the default dialog box and write message handlers.

Adding Extra Controls

Explorer-style file dialog box has a nice feature that lets us add extra controls. We can design our own dialog template, and add other common controls. Instead of copying all the default controls contained in the standard dialog box, we can just create a static text control with symbolic ID of stc32. The relative positions between our controls and stc32 will be used to decide the layout of the file dialog box. When using this dialog template, we need to set OFN_ENABLETEMPLATE bit for member Flags of structure OPENFILENAME, and assign the name of custom dialog template to member lpTemplateName. Also, we need to derive a new class from CFileDialog in order to handle messages for the newly added controls.

Notification CDN_SELCHANGE

File selection activities are sent to the dialog box through WM_NOTIFY message. This message is primarily used by common controls to send notifications to the parent window. For example, in a file dialog box, if the user has changed the file selection, a CDN_SELCHANGE notification will be sent. Since message WM_NOTIFY is used for many purposes, after receiving it, we need to check LPARAM parameter and decide if the notification is CDN_SELCHANGE or not. In CWnd derived classes, WM_NOTIFY message is handled by function CWnd:OnNotify(...), we can override it to customize the default behavior of the file dialog box.

The following is the format of function CWnd::OnNotify(...):

```
BOOL CWnd::OnNotify(WPARAM wParam, LPARAM lParam, LRESULT *pResult);
```

To decide if this notification is CDN_SELCHANGE or not, first we need to cast lParam parameter to an OFNOTIFY type pointer. The following is the format of structure OFNOTIFY:

```
typedef struct _OFNOTIFY {

NMHDR hdr;

LPOPENFILENAME lpOFN;

LPTSTR pszFile;

} OFNOTIFY, FAR *LPOFNOTIFY;
```

The first member of OFNOTIFY is an NMHDR structure:

```
typedef struct tagNMHDR {

HWND hwndFrom;

UINT idFrom;

UINT code;

} NMHDR;
```

From its member code, we can judge if the current notification is CDN_SELCHANGE or not.

Sample

Sample 7.4\CDB demonstrates how to add a file preview window to Explorer-style file dialog box. If the user selects a file with "*.cpp" extension after the file dialog is activated, the contents of the file will be displayed in the preview window before it is opened.

In the sample, a new command File Dialog Box | Custom File Dlg is added to the application, whose command ID is ID_FILEDIALOGBOX_CUSTOMFILEDLG. Also, a WM_COMMAND message handler is added for this command through using Class Wizard, whose correspondung member function is CCDBDoc:: OnFiledialogboxCustomfiledlg().

A new dialog template IDD_COMDLG32 is also added, it contains a static text control stc32, and an edit box control IDC_EDIT (Figure 7-4). This edit box has "Disabled" and "Multiline" styles. This will implement a read only edit box that can display multiple lines of text. A new class MCCusFileDialog is derived from

FileDialog.

In this class, function OnNotity(...) is overridden.

The following is the definition of class MCCusFileDialog:

(Code omitted)

Function MCCusFileDialog::OnNotify(...) is implemented as follows:

(Code omitted)

First we check if the notification is CDN_SELCHANGE, if so, we retrieve the path name and the file extension by calling function CFileDialog::GetPathName() and CFileDialog::GetFileExt() respectively. If the file extension is "*.cpp", we call function CFile::Open(...) to open this file. When making this call, we use flag CFile::modeRead to open it as a read only file. Then function CFile::Read(...) is called to read the first 255 characters of the file into buffer szBuf. Finally, function CWnd::SetWindowText(...) is called to display these characters.

Command File Dialog Box | Custom File Dlg is implemented as follows:

(Code omitted)

The file dialog box is implemented using class MCCusFileDialog. To use the custom dialog template, OFN_ENABLETEMPLATE bit is set for member Flags, and the name of the custom dialog template is assigned to member lpTemplateName of structure OPENFILENAME. Function CFileDialog::DoModal() is called as usual.

If we want to modify the size and relative position of the preview window, we need to override function OnInitDialog(), and call function CWnd::MoveWindow(...) there to resize the edit box.

7.5 Color Dialog Box

Introduction

A color dialog box lets the user choose one or several colors from a pool of available colors. In Windows(, a valid color may be formed by combining red, green and blue colors with different intensities. There are altogether 16,777,216 possible colors in the system.

In MFC, color dialog box is supported by class CColorDialog. To create a color dialog box, we need to use class CColorDialog to declare a variable, then call

CColorDialog::DoModal() to activate the dialog box.

In the color dialog box, there are four ways to choose a color (Figure 7-5):

(Figure 7-5 omitted)

1) Select a color from those listed as Basic colors.

2) Select a color from those listed as Custom colors.

3) Select a color from "color matrix".

4) Input R, G, B values directly.

The selected color is shown in "Color | Solid" window. When a dialog box is implemented, there are two things that we can customize: the original selected color and the custom colors. If we do not change anything, the default selected color is black (RGB(0, 0, 0)). There are altogether 16 custom colors. By default, they are all initialized to white (RGB(255, 255, 255)) at the beginning.

Initializing Selected Color and Custom Colors

The default selected color can be set when the constructor of class CColorDialog is called. The function has the following format:

```
CColorDialog::CcolorDialog

(

COLORREF clrInit=0, DWORD dwFlags=0, CWnd *pParentWnd=NULL

);
```

There are three parameters here, the first of which is the default selected color, the second is the flags that can be used to customize the dialog box, and the third is a pointer to the parent window.

Class CColorDialog has a member m_cc, which is a CHOOSECOLOR structure:

```
typedef struct {

DWORD lStructSize;

HWND hwndOwner;
HWND hInstance;

COLORREF rgbResult;

COLORREF* lpCustColors;

DWORD Flags;

LPARAM lCustData;

LPCCHOOKPROC lpfnHook;

LPCTSTR lpTemplateName;

} CHOOSECOLOR;
```

We can change the custom colors by assigning the address of a COLORREF type array with size of 16 to member lpCustColors of this structure. The colors in the array will be used to initialize the custom colors.

After the dialog box is closed, we can call function CColorDialog::GetColor() to retrieve the selected color.

Also, if we've initialized custom colors, we can obtain the updated values by accessing member lpCustColors.

Sample

Sample 7.5\CDB demonstrates how to use color dialog box. First a function ColorDialog(...) is added to class CCDBDoc, which will implement a color dialog box whose default selected color and custom colors are customized. The following is its definition:

```
class CCDBDoc : public CDocument

{

……

public:

void ColorDialog(COLORREF colorInit, DWORD dwFlags=0);
......

}
```

This function has two parameters, the first one specifies the initially selected color, and the second one specifies the style flags. We will show how to use different style flags later, for the time being we just set all bits to 0. The function is implemented as follows:

(Code omitted)

We first create a color dialog box and use colorInit to initialize the selected color. Like OPENFILENAME, structure CHOOSECOLOR also has a Flags member that can be used to set the styles of color dialog box. In the above function new flags contained in parameter dwFlags are added to the default flags through bit-wise OR operation. Variable color is a COLORREF type array with a size of 16. We use a loop to fill each of its elements with a different color and pass the address of the array to member lpCustColors of structure CHOOSECOLOR. After calling CColorDialog::DoModal(), function CColorDialog::GetColor() is called to retrieve the selected color, whose RGB values are displayed in a message box. Besides this, the RGB values of custom colors are also displayed.

In the sample, a new sub-menu Color Dialog Box is added to IDR_MAINFRAME menu, and a command Color Dialog Box | Initialize is created. The ID of this command is ID_COLORDIALOGBOX_INITIALIZE, also, a WM_COMMAND message handler CCDBDoc::OnColordialogboxInitialize() is added through using Class Wizard.

The following is the implementation of this command:

```
void CCDBDoc::OnColordialogboxInitialize()

{

ColorDialog(RGB(255, 0, 0));

}
```

The selected color is initialized to red, and no additional styles are specified.

Full Open

Now we are going to customize the styles of color dialog box. First, we can create a fully opened dialog box by setting CC_FULLOPEN bit for member Flags of CHOOSECOLOR structure. Also, we can prevent the dialog from being fully opened by setting CC_PREVENTFULLOPEN bit. In the sample, two menu commands Color Dialog Box | Disable Full Open and Color Dialog Box | Full Open are added, in their corresponding message handlers, the color dialog boxes with different styles are implemented:

```
void CCDBDoc::OnColordialogboxDisablefullopen()

{

ColorDialog(RGB(0, 255, 0), CC_PREVENTFULLOPEN);

}

void CCDBDoc::OnColordialogboxFullopen()

{

ColorDialog(RGB(0, 255, 0), CC_FULLOPEN);

}
```

Now we can compile the application and try color dialog boxes with different styles.

7.6 Custom Dialog Box Template

Like file dialog box, we can use our own dialog template to implement color dialog box. When designing our own dialog template, we must include all the items contained in the standard color dialog box. We can change the position and resize some controls, and hide the ones that do not fit our use. To use the custom dialog template, we need to set CC_ENABLETEMPLATE bit for member Flags of structure CHOOSECOLOR, and assign the instance handle of the application to member hInstance. Here the instance handle of the application can be obtained by calling function AfxGetInstanceHandle(). We also need to assign the name of the custom dialog template to member lptemplateName.

Custom Dialog Template

Sample 7.6\CDB demonstrates how to implement color dialog box using user-designed dialog template. It is based on sample 7.5\CDB, with two new commands Color Dialog Box | Choose Base Color and Color Dialog Box | Choose Custom Color added to the application. For command Color Dialog Box | Choose Base Color, a color dialog box that only allows the user to choose color from base colors is implemented; for command Color Dialog Box | Choose Custom Color, the user is only allowed to choose color form custom colors. The IDs of two commands are ID_COLORDIALOGBOX_CHOOSEBASECOLOR and ID_COLORDIALOGBOX_CHOOSECOSTUMCOLOR, and their WM_COMMAND message handlers are CCDBDoc::OnColordialogboxChoosebasecolor() and CCDBDoc:: OnColordialogboxChoosecostumcolor() respectively.

Figure 7-6 shows the controls contained in the standard color dialog box.

(Figure 7-6 omitted)

The following table contains a list of ID values for these controls:

(Table omitted)

The standard dialog template can be copied from file "Commdlg.dll". By default, all the controls in this template will have a numerical ID. In order to make them easy to use, we can assign each ID a symbol, this can be done by inputting a text ID and assigning the control's original ID value to it in "ID" window of the property sheet that is used for customizing the styles of a control. For example, when we open "Text Properties" property sheet for control 720, its "ID" window shows "720". We can change it to "COLOR_BOX1=720". By doing this, the control will have an ID symbol "COLOR_BOX1", whose value is 720. In the sample, most of the controls are assigned ID symbols.

We can hide certain unnecessary controls by calling function CWnd::ShowWindow(...) in dialog box's initialization stage. However, there is an easier approach to it: we can resize the dialog template and move the unwanted controls outside the template (Figure 7-7). By doing this, these controls will not be shown in the dialog box, and therefore, can not respond to mouse clicking events.

However, in Developer Studio, a control is confined within the dialog template and is not allowed to be moved outside it. A workaround for this is to edit the resource file directly. Actually, a dialog template is based on a text format resource file. In our sample, all type of resources are stored in file "CDB.rc". By opening it in "Text" mode, we can find the session describing the color dialog template:

(Code omitted)

The first four lines specify the properties of this dialog template, which include its dimension, styles, caption, and font. Between "BEGIN" and "END" keywords, all controls included in the template are listed. Each item is defined with a type description followed by a series of styles. In the above example, the first item is a static text control (LTEXT), which contains text "Basic Colors" ("&Basic Colors:"). It has an ID of 65535 (-1), located at (4, 4), and its dimension is (140, 9).

In the above example, if we change a control's horizontal coordinate to a value bigger than 150 (Because the dimension of the dialog template is 150(124), it will be moved outside the template.

In the sample, two such dialog templates are prepared: "CHOOSECOLOR" and "CHOOSECUSCOLOR". In "CHOOSECOLOR", static text control COLOR_BOX1 is inside the template, and the area within this control will be used to draw base colors. For "CHOOSECUSCOLOR", static text control COLOR_CUSTOM1 is inside the template, the area within this control will be used to draw custom colors (COLOR_BOX1 and COLOR_CUSTOM1 are used to define an area where the controls will be created dynamically for displaying colors). In both cases, frame COLOR_CURRENT is inside the template, which will be used to display the current selected color.

Commands Implementation

Function CCDBDoc::OnColordialogboxChoosebasecolor() is implemented as follows:

(Code omitted)

There is nothing special for this function. The only thing we need to pay attention to is that we must set CC_FULLOPEN flag in order to display currently selected color. Otherwise control COLOR_CURRENT will not work.

The following is the implementation of function CCDBDoc::OnColordialogboxChoosecostumcolor():

(Code omitted)

We must provide custom colors. Otherwise they will all be initialized to white. With a color dialog box whose color matrix window can not be used, the custom colors provide the only way to let user pick up a color.

7.7 Font Dialog Box

Basics

Font dialog box lets user select a special font with different style combinations: boldface, italic, strikeout or underline. The font size and color can also be set in the dialog box.. In MFC, the class that is used to implement font dialog box is

CFontDialog. To create a standard font dialog box, all we need to do is declaring a CFontDialog type variable and using it to call function CFontDialog::DoModal(). Like classes CFileDialog and CColorDialog, class CFontDialog also contains a member variable that allows us to customize the default styles of color dialog box. This variable is m_cf, which is declared by structure CHOOSEFONT:

```
typedef struct {
DWORD lStructSize;
HWND hwndOwner;
HDC hDC;
LPLOGFONT lpLogFont;
INT iPointSize;
DWORD Flags;
DWORD rgbColors;
LPARAM lCustData;
LPCFHOOKPROC lpfnHook;
LPCTSTR lpTemplateName;
HINSTANCE hInstance;
LPTSTR lpszStyle;
WORD nFontType;
WORD ___MISSING_ALIGNMENT__;
INT nSizeMin;
INT nSizeMax;
} CHOOSEFONT;
```

There are several things that can be customized here. First, we can change the default font size range. By default, a valid size for all fonts is ranged from 8 to 72. We can set a narrower range by setting CF_LIMITSIZE bit of member Flags and assigning lower and higher boundaries to members nSizeMin and nSizeMax respectively. We can also specify font's initial color by setting CF_EFFECTS bit of member Flags and assigning a COLORREF value to member rgbColors (the initial color must be one of the standard colors defined in the font dialog box such as red, green, cyan, black...).

Structure LOGFONT

Also, we can specify an initially selected font (with specified size and styles) by assigning a LOGFONT type pointer to member lpLogFont. Here, structure LOGFONT is used to describe a font:

```
typedef struct tagLOGFONT {

LONG lfHeight;

LONG lfWidth;

LONG lfEscapement;

LONG lfOrientation;

LONG lfWeight;

BYTE lfItalic;

BYTE lfUnderline;

BYTE lfStrikeOut;

BYTE lfCharSet;

BYTE lfOutPrecision;

BYTE lfClipPrecision;

BYTE lfQuality;

BYTE lfPitchAndFamily;

CHAR lfFaceName[LF_FACESIZE];
} LOGFONT;
```

A font has many properties. The most important ones are face name, height, and font styles (italic, bolded, underlined or strikeout). In structure LOGFONT, these styles are represented by the following members: lfFaceName, lfWeight, lfItalic, lfUnderline and lfStrikeOut. A face name is the name of the font, it distinguishes one font from another. Every font has its own face name, such as "Arial", "System" and "MS Serif". The weight of a font specifies how font is emphasized, its range is from 0 to 1000. Usually we use predefined values such as FW_BOLD (font is bolded) and FW_NORMAL (font is not bolded) for convenience. Member lfItalic, lfUnderline and lfStrikeOut are all Boolean type, by setting them to TRUE, we can let the font to become italic, underlined, or strikeout. Member nSizeMin and nSizeMax can be used to restrict the size of a font.

In order to use LOGFONT object to initialize a font dialog box, we must first set CF_INITTOLOGFONTSTRUCT bit for member Flags of structure CHOOSEFONT, then assign the LOGFONT type pointer to member lpLogFont.

Retrieving Selected Font

After the font dialog box is closed, the font that is selected by the user can be retrieved through following member functions: CFontDialog::GetFaceName(), CFontDialog::IsBold(), CFongDialog:: IsItalic()....

Sample

Sample 7.7\CDB demonstrates how to use font dialog box. It is based on sample 7.6\CDB with a new command Font Dialog Box | Initialize added to the application. The ID of this command is ID_FONTDIALOGBOX_INITIALIZE, and its WM_COMMAND message handler is CCDBDoc:: OnFontdialogboxInitialize(). The command is implement as follows:

(Code omitted)

The initially selected font is "Times New Roman", whose color is yellow (RGB(255, 255, 0)), and has the following styles: italic, underlined, strikeout, bolded. The range of the font size is restricted between 20 and 48. After the user clicks button "OK", the properties of the selected font are retrieved through member functions of class CFontDialog, and are displayed in a message box.

7.8 Customizing Dialog Box Template

Like file and color dialog boxes, we can use custom dialog template to implement font dialog box. This gives us a chance to move, resize or hide some standard controls. To use a custom template, we need the following steps: 1) Design a new dialog template that contains all the controls in a standard font dialog box. 2) Set CF_ENABLETEMPLATE bit for member Flags of structure CHOOSEFONT, and assign custom template name to member lpTemplateName. 3) Assign application's instance handle to member hInstance.

The standard dialog template can be found in file "Commdlg.dll". It can be opened from Developer Studio in "Resources" mode.

All IDs of the controls are numbers. When writing code to access the controls, we can use these numbers directly, or we can assign each control a text ID like what we did for sample 7.6\CDB. Actually, these IDs are all pre-defined, whose symbolic IDs can be found in file "Dlgs.h" (we can find this file under ~DevStudio\VC\Include\ directory). We can check a control's ID value and search through this file to find out its symbolic ID. For example, in font dialog box, the combo box under window "Font:" has an ID of 1136 (0x470). In file "Dlgs.h", we can find:

```
......

//

// Combo boxes.

//

#define cmb1 0x0470

#define cmb2 0x0471

#define cmb3 0x0472

......
```

Value 0x0470 is used by cmb1, so we can access this combo box through using symbol cmb1.

Sample 7.8\CDB demonstrates how to use custom dialog template to implement font dialog box. It is based on sample 7.7\CDB. In the sample, a new command Font Dialog Box | Customize is added to the application, whose command ID is ID_FONTDIALOGBOX_INITIALIZE. If we execute this command, a font dialog box whose color selection feature is disabled will be invoked.

In order to implement this dialog box, we need to disable the following two controls: stc4 (Static control containing string "Color", whose ID value is 1091) and cmb4 (Combo box for color selection, whose ID value is 1139).

In the sample, the custom dialog template is IDD_FONT. It contains all the standard controls.

A new class MCFontClass is added to the application through using Class Wizard, its base class is CFontClass. In the derived class, function OnInitDialog is overridden, within which the above two controls are disabled:

(Code omitted)

The WM_COMMAND type message handler of command ID_FONTDIALOGBOX_INITIALIZE is CCDBDoc:: OnFontdialogboxCustomize(), which is also added through using Class Wizard. The following is its implementation:

(Code omitted)

With the above implementation, the font dialog box will not contain color selection feature.

7.9 Modeless Common Dialog Boxes

Tricks

It is difficult to implement modeless common dialog boxes. This is because all the common dialog boxes are designed to work in the modal style. Therefore, if we call function Create(...) instead of DoModal(), although the dialog box will pop up, it will not behave like a common dialog box. This is because function Create(...) is not overridden in a class such as CColorDialog, CFontDialog.

We need to play some tricks in order to implement modeless common dialog boxes. By looking at the source code of MFC, we can find that within function CColorDialog::DoModal() or CFontDialog:: DoModal(), the base class version of DoModal() is not called. Instead, API functions ::ChooseColor(...)and ::ChooseFont(...) are used to implement common dialog boxes.

There is no difference between the common dialog boxes implemented by API functions and MFC classes. Actually, using API function is fairly simple. For example, if we want to implement a color dialog box, we can first prepare a CHOOSECOLOR object, then use it to call function ::ChooseColor(...). But here we must initialize every member of CHOOSECOLOR in order to let the dialog box have appropriate styles.

By using this method we can still create only modal common dialog boxes. A "modeless" common dialog box can be implemented by using the following method:

1) Before creating the common dialog box, first implement a non-visible modeless window.

2) Create the modal common dialog box, use the modeless window as its parent.

Although the common dialog box is modal, its parent window is modeless. So actually we can switch away from the common dialog box (and its parent window) without closing it. Because the common dialog box's parent is invisible, this gives the user an impression that the common dialog box is modeless.

But if we call function DoModal() to implement the common dialog box, we are not allowed to switch away even it has a modeless parent window. We must call API functions to create this type of common dialog boxes.

Hook Function

We can provide hook function when implementing common dialog boxes using API functions. In Windows(programming, a hook function is used to intercept messages sent to a window, thus by handling these messages we can customize a window's behavior. Both structure CHOOSEFONT and CHOOSECOLOR have a member lpfnHook that can be used to store a hook function's address when the common dialog box is being implemented. If a valid hook function is provided, when a message is sent to the dialog box, the hook function will be called first to process the message. To enable hook function, we also need to set CF_ENABLEHOOK or CC_ENABLEHOOK bit for member Flags of structure CHOOSEFONT or CHOOSECOLOR. If the message is not processed in the hook function, the dialog's default behavior will not change.

A hook procedure usually looks like the following:

(Code omitted)

The first parameter is the handle of the destination window where the message is being sent. The second parameter specifies the type of message. The third and fourth parameters are WPARAM and LPARAM parameters of the message respectively. For different events, the messages sent to this procedure are different. For example, in the dialog's initialization stage, message WM_INITDIALOG will be sent, if we want to add other initialization code, we could implement it under "case WM_INITDIALOG" statement.

In MFC, this procedure is encapsulated by message mapping. We don't have to write hook procedure, because we can map any message to a specific member function. For example, in MFC, message WM_INITDIALOG is always mapped to function OnInitDialog(), so we can always do our initialization work within this member function.

When we use MFC classes to implement common dialog boxes, there is a hook function _AfxCommDlgProc(...) behind us. We never need to know its existence. However, we use it indirectly whenever a common dialog box is created. By looking at MFC source code, we will find that in the constructors of common dialog boxes, the address of this function is assigned to member lpfnHook.

To make full use of MFC resource, instead of creating a new hook function, we can use _AfxCommDlgProc(...) when calling API functions to implement common dialog boxes.

Using MFC Classes together with API Functions

When we use MFC class to create a window, the window's handle is automatically saved in a member variable. So we can always use the member functions to access this window. If we use API functions, no MFC class declared variables can participate in window creating activities, so we will not have any variable that can be used to call the member functions of MFC classes for accessing the window. This doesn't mean we are going to give up MFC implementation completely. Whenever possible, we want to use the member functions of CColorDialog or CFileDialog instead of implementing everything by our own. Actually, a window and a MFC class declared variable can be associated together after the window is created by calling function CWnd::Attach(...). The following is the format of this function:

```
BOOL CWnd::Attach(HWND hWndNew);
```

So long as we have a valid window handle, we can attach it to a MFC class declared variable.

Obtaining Handle

When we call function ::ChooseColor(...) or ::Choosefont(...), no window handle will be returned. The only place we can obtain dialog's handle is in the hook function (through parameter hdlg). We can attach this handle to a CColorDialog or CFontDialog declared variable after receiving message WM_INITDIALOG.

Accessing Member Variable from Static or Global Function

Because a hook function is a callback function, it must be either a static or global function. Therefore, we can not access the member variable of a class within the hook function. To let the window handle be attached to a member variable, we must pass its address to the callback function through a message parameter. By doing this, after receiving the message, the member variable can be accessed through a pointer within the callback function.

Both structure CHOOSECOLOR and CHOOSEFONT have a member lCustData, which allows us to specify a custom data that will be sent to the callback function along with WM_INITDIALOG message. The custom data will be contained in LPARAM message parameter. So if we assign the address of a variable declared by MFC class to member lCustData, we can receive it in the dialog box' initialization stage and call function CWnd:: Attach(...) to attach the window handle to this variable.

In other cases we would like to call the default hook procedure. We can store the address of the default hook procedure in a variable, and call it within the our hook procedure as follows:

(Code omitted)

The code listed above shows how to trap WM_INITDIALOG message and attach the window handle to a variable declared outside the hook function. Also, the default hook procedure is called to let other messages be processed normally. Here, lpFontfn is a global pointer that stores the address of the hook procedure.

Sample Implementation

Sample 7.9\CDB demonstrates how to implement modeless common dialog boxes. It is based on sampele 7.8\CDB, with two new commands added to the application: Color Dialog Box | Modeless and Font Dialog Box | Modeless, both of which can be used to invoke modeless common dialog boxs. For the former command, the user can select a color and switch back to the main window to see the effect without dismissing the dialog box. The IDs of the two commands are ID_COLORDIALOGBOX_MODELESS and ID_FONTDIALOGBOX_MODELESS, and their WM_COMMAND message handlers are CCDBDoc:: OnColordialogboxModeless() and CCDBDoc::OnFontdialogboxModeless() respectively.

A dummy dialog box is added to the application, whose resource ID is IDD_DIALOG_DUMMY. It will be used as the parent window of the common dialog boxes. Since this window is always hidden after being invoked, it does not matter what controls are included in the dialog template. The class that will be used to implement this dialog box is MCDummyDlg.

Two new variables are declared in class CCDBDoc for implementing modeless color dialog box:

(Code omitted)

Here, m_pColorDmDlg will be used to create dummy window, and m_pColorDlg will be used to create color dialog box.

At the beginning of file "CDBDoc. Cpp", a global hook procedure and a pointer are declared:

UINT CALLBACK ColorHook(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);

UINT (CALLBACK *lpColorfn)(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);

Function ColorHook is the custom hook procedure, and pointer lpColorfn will be used to store the address of the default hook procedure.

In function CCDBDoc::OnColordialogboxModeless(), first we need to initialize m_pColorDlg and m_pColorDmDlg, then create the dummy window:

(Code omitted)

Function CDialog::Create() is called to create a modeless dialog box, and function CWnd:: ShowWindow(...) (using parameter SW_HIDE) is called to hide this window.

Before the color dialog box is created, we must make some changes to structure CHOOSECOLOR:

(Code omitted)

The address of m_pColorDlg is stored as custom data, which will be sent to the hook function. The dummy window is designated as the parent window of the color dialog box and its handle is assigned to member hwndOwner of structure CHOOSECOLOR. A global COLORREF type array rgbColors is declared, which will be used to initialize the custom colors in the color dialog box. Also, custom dialog template "CHOOSECUSCOLOR" is used, which will allow the user to choose color from only custom colors.

The address of default hook procedure (which is contained in member lpfnHook of structure CHOOSECOLOR after the constructor of class CColorDlg is called) is stored in global variable lpColorfn, and the new hook procedure address is assigned to lpfnHook. Finally, API function ::ChooseColor(...) is called to invoke the color dialog box:

(Code omitted)

In the hook procedure, after receiving message WM_INITDIALOG, we can obtain the value of m_pColorDlg from LPARAM parameter and attach the color dialog box's window handle to it:

(Code omitted)

However, there are still some problems left to be solved. Since the dialog box is modeless now, we can execute command Color Dialog Box | Modeless again when the dialog box is being used. Also, in the new situation, the user is able to exit the application without closing the dialog box first.

To avoid the dummy dialog box and the color dialog box from being created again while they are active, we have to check m_pColorDlg and m_pColorDmDlg variables. First, if they are NULL, it means the variables have not been initialized, we need to allocate buffers and create the window. If they are not NULL, there are two possibilities: 1) The dialog box is currently active. 2) The dialog box is closed. Obviously we don't need to do anything for the first case. For the second case, we need to reinitialize the two variables and create the window again. Since the window handle is attached to the variable in the hook procedure, we need to detach it before releasing the allocated buffers. For the above reasons, the following is added to the beginning of function CCDBDoc::OnColordialogboxModeless():

(Code omitted)

We need to destroy the windows by ourselves if the user exits the application without first closing the color dialog box. We can override a member function OnCloseDocument(), which will be called when the document is about to be closed. This function can be easily added through using the Class Wizard. The following shows how it is implemented in the sample:

(Code omitted)

Again, function CWnd::Detach() is called before the buffers are released.

Applying Selected Color Instantly

To make the sample more practical, another feature is added to it: when the user picks up a color, the client window of the application will be painted with that color instantly. To implement this, a new COLORREF type variable m_colorCur is added to class CDBDoc, which will be used to store the current color of the client window:

```
class CCDBDoc : public CDocument
{
protected:
……
COLORREF m_colorCur;
……
}
```

The color is initialized in the constructor as follows:

```
CCDBDoc::CCDBDoc()
{
……
m_colorCur=RGB(0, 255, 255);
}
```

Member function CCDBDoc::SetCurrentColor(...) and CCDBDoc::GetCurrentColor() are added to let us access variable m_colorCur outside the document:

```
class CCDBDoc : public CDocument
{
……
public:
void SetCurrentColor(COLORREF);
COLORREF GetCurrentColor(){return m_colorCur;}
......

}
```

Function CCDBDoc::GetCurrentColor(...) is implemented as an inline function, and function CCDBDoc::SetCurrentColor(...) is implemented as follows:

(Code omitted)

Here we check if the new color is the same with the old color, if not, we update member m_colorCur, and repaint the client window by calling function CDocument::UpdateAllViews(...).

When initializing the color dialog box, we need to use m_colorCur to set the initially selected color before the color dialog box is created. The following change is made for this purpose:

Before change:

```
void CCDBDoc::OnColordialogboxModeless()

{

......

m_pColorDlg=new CColorDialog();

......

}
```

After change:

```
void CCDBDoc::OnColordialogboxModeless()

{

......

m_pColorDlg=new CColorDialog(m_colorCur);

......

}
```

Function CCDBView::OnDraw(...) is modified to paint the client window with color CCDBDoc:: m_colorCur:

(Code omitted)

First we find out the size of the client window, then call function CCDBDoc::GetCurrentColor() to retrieve the current color, and call function CDC::FillSolidRect() to fill the window with this color.

When the user selects a new color, we need to call function CCDBDoc::SetCurrentColor(...) to update the current color. In order to do this, we need to trap message WM_LBUTTONUP in the hook procedure, obtain the selected color and update variable CCDBDoc::m_colorCur. For this purpose: the following is added to function ColorHook(...):

(Code omitted)

Since ColorHook(...) is not a member function of class CCDBDoc, we can not access its member function directly from the hook procedure. So here AfxGetMainWnd() is called first to obtain the mainframe window, then CFrameWnd::GetActiveDocument() is called to obtain the active document attached to it. This method can also be used to access the active document from other classes.

When calling function CCDBDoc::SetCurrentColor(...), we use IDs COLOR_RED, COLOR_GREEN and COLOR_BLUE to retrieve the current values contained in the edit boxes (see Figure 7-6). In a standard color dialog box, these edit boxes will be shown only when the dialog box is fully opened. Although this is not the case in the sample, we still can retrieve the contents of them even they can not be seen. Also, we use CDialog::GetDlgItemInt(...) to convert characters to integers when retrieving the color values.

In the sample, modeless font dialog is implemented in a similar way.

Summary:

1) Three classes that can be used to implement common file dialog box, common color dialog box and common font dialog box are CFileDialog, CColorDialog and CFontDialog respectively. To implement a standard common dialog box, we need to use the corresponding class to initialize a variable, then call function DoModal() to invoke the dialog box.

2) We can customize the default behavior of common dialog boxes by modifying the members of structure OPENFILENAME, CHOOSECOLOR or CHOOSEFONT.

3) File dialog box can be implemented either in an "Explorer" style or an "Old" style.

4) There are some shell functions that can be called to implement folder selection dialog box. If we want to implement the old-style interface, we must use custom dialog template and override class CFileDialog.

5) To use custom dialog template, we need to first design a dialog template that contains all the standard controls, then set "enable template" flag and assign the template name to member lpTemplateName.

6) To add extra controls to an "Explorer" style file dialog box, we need to design a dialog template that contains static control with ID of stc32. We do not need to replicate all the standard controls.

7) MFC classes do not support modeless common dialog boxes. To implement this type of dialog boxes, we need to create a modeless parent window for the common dialog box and hide the parent window all the time. This will give the user an impression that the common dialog box is modeless. Also, we need to call API functions instead of MFC member functions to create the common dialog box.

BACK TO INDEX

Chapter 8 DC, Pen, Brush and Palette

8.0 Device Context & GDI Objects

Starting from this chapter, we are going to study topics on GDI (Graphics Device Interface) programming.

Situation

GDI is a standard interface between the programmer and physical devices. It provides many functions that can be used to output various objects to the hardware (e.g. a display or a printer). GDI is very important because, as a programmer, we may want our applications to be compatible with as many peripherals as possible. For example, almost every application need to write to display, and many applications also support printer output. The problem here is that since a program should be able to run on different types of devices (low resolution displays, high resolution displays with different color depth, etc.), it is impossible to let the programmer know the details of every device and write code to support it beforehand.

The solution is to introduce GDI between the hardware and the programmer. Because it is a standard interface, the programmer doesn't have to have any knowledge on the hardware in order to operate it. As long as the hardware supports standard GDI, the application should be able to execute correctly.

Device Context

As a programmer, we do not output directly to hardware such as display or printer. Instead, we output to an object that will further realize our intention. This object is called device context (DC), it is a Windows(object that contains the detailed information about hardware. When we call a standard GDI function, the DC implements it according to hardware attributes and configuration.

Suppose we want to put a pixel at specific logical coordinates on the display. If we do not have GDI, we need the following information of the display in order to implement this simple operation:

1) Video memory configuration. We need this information in order to convert logical coordinates to physical buffer address.

2) Device type. If the device is a palette device, we need to convert a RGB combination to an index to the color table and use it to specify a color. If the device is a non-palette device, we can use the RGB combination directly to specify a color.

Because the actual devices are different form one type to another, it is impossible for us to gather enough information to support all the devices in the world. So instead of handling it by the programmer, GDI functions let us use logical coordinates and RGB color directly, the conversion will be implemented by the device driver.

GDI Objects

In Windows(, GDI objects are tools that can be used together with device context to perform various drawings. They are designed for the convenience of programmers. The following is a list of some commonly used GDI objects:

(Table omitted)

The above GDI objects, along with device context, are all managed through handles. We can use the handle of an object to identify or access it. Besides the handles, every GDI object has a corresponding MFC class. The following is a list of their handle types and classes:

(Table omitted)

As a programmer, most of the time we need to output to a specific window rather than the whole screen. A DC can be obtained from any window in the system, and can be used to call GDI functions. There are many ways to obtain DC from a window, the following is an incomplete list:

1) Call function CWnd::GetDC(). This function will return a CDC type pointer that can be used to perform drawing operations within the window.

2) Declare CClientDC type variable and pass a CWnd type pointer to its constructor. Class CClientDC is designed to perform drawing operations in the client area of a window.

3) Declare CWndowDC type variable and pass a CWnd type pointer to its

constructor. Class CWindowDC is designed to perform drawing operations in the whole window (including client area and non-client area).

4) In MFC, certain member functions are designed to update application's interface (i.e. CView:: OnDraw(...)). These functions will automatically be called when a window needs to be updated. For this kind of functions, the device context will be passed through one of function's parameters.

5) If we know all the information, we can create a DC by ourselves.

Using DC with GDI Objects

Before calling any function to perform drawing, we must make sure that an appropriate GDI object is being selected by the DC. For example, if we want to draw a red line with a width of 2, we must select a solid red pen whose width is 2. The following steps show how to use DC together with GDI objects:

1) Obtain or create a DC that can be used to perform drawing operations on the target window.

2) Create or obtain an appropriate GDI (pen, brush, font...) object.

3) Select the GDI object into the DC, use a pointer to store the old GDI object.

4) Perform drawing operations.

5) Select the old GDI object into the DC, this will select the new GDI object out of the DC.

6) Destroy the GDI object if necessary (If the GDI object was created in step 2 and will not be used by other DCs from now on).

The following sections will discuss how to use specific GDI objects to draw various kind of graphic objects.

8.1 Line

Creating Pen

Sample 8.1\GDI demonstrates how to create a pen and use it to draw lines. The sample is a standard SDI application generated from Application Wizard.

To draw a line, we need the following information: starting and ending points, width of the line, pattern of the line, and color. There are several types of pens that can be created: solid pen, dotted pen, dashed pen. Besides drawing patterns, each pen can have a different color and different width. So if we want to draw two types of lines, we need to create two different pens.

In MFC, pen is supported by class CPen, it has a member function CPen::CreatePen(...), which can be used to create a pen. This function has several versions, the following is one of them:

```
BOOL CPen::CreatePen(int nPenStyle, int nWidth, COLORREF crColor);
```

Parameter nPenStyle specifies the pen style, it can be any of the following: PS_SOLID, PS_DASH, PS_DOT, PS_DASHDOT, PS_DASHDOTDOT, PS_NULL, etc. The meanings of these styles are self-explanatory. The second parameter nWidth specifies width of the pen. Please note that if we create pen with a style other than PS_SOLID, this width can not exceed 1 device unit. Parameter crColor specifies color of the pen, which can be specified using RGB macro.

In MFC's document/view structure, the data should be stored in CDocument derived class and the drawing should be carried out in CView derived class (for SDI or MDI applications). Since CView is derived from CWnd, we can obtain its DC by either calling function CWnd::GetDC() or declare a CClientDC type variable using the window's pointer. To select a GDI object into the DC, we need to call function CDC::SelectObject(...). This function returns a pointer to a GDI object of the same type that is being selected by the DC. Before we delete the GDI object, we need to use this pointer to select the old GDI object into the DC so that the new DC will be selected out of the DC.

We can not delete a GDI object when it is being selected by a DC. If we do so, the application will become abnormal, and may cause GPF (General protection fault) error.

Drawing Mode

Besides drawing lines, sample 8.1\GDI also demonstrates how to implement an interactive environment that lets the user use mouse to draw lines anywhere within the client window. In the sample, the user can start drawing by clicking mouse's left button, and dragging the mouse with left button held down, releasing the button to finish the drawing. When the user is dragging the mouse, dotted outlines will be drawn temporarily on the window. After the ending point is decided, the line will be actually drawn (with red color and a width of 1). In order to implement this, the following messages are handled in the application: WM_LBUTTONDOWN, WM_MOUSEMOVE and WM_LBUTTONUP.

Before the left mouse button is released, we have to erase the old outline before drawing a new one in order to give the user an impression that the outline "moves" with the mouse cursor. The best way to implement this type of operations is using XOR drawing mode. With this method, we can simply draw an object twice in order to remove it from the device.

XOR bit-wise operation is very powerful, we can use it to generate many special drawing effects. Remember when we perform a drawing operation, what actually happens in the hardware level is that data is filled into the memory. The original data contained in the memory indicates the original color of pixels on the screen (if we are dealing with a display device). The new data can be filled with various modes: it can be directly copied into the memory; it can be first combined with the data contained in the memory, then the combining result is copied into the memory. In the latter case, the combination could be bit-wise AND, OR, XOR, or simply a NOT operation on the original data. So by applying different drawing modes, the output color doesn't have to be the color of the pen selected by the DC. It could be either the combination of the two (original color and the pen color), or it could be the complement of the original color.

If we draw an object twice using XOR operation mode, the output color will be the original color on the target device. This can be demonstrated by the following equation:

A^B^B=A^(B^B)=A^0=A

Here A is the original color, and B is the new color. The above equation is valid because XORing any number with itself results in 0, and XORing a number with 0 does not change this number.

When using a pen, we can select various drawing modes. The following table lists some modes that can be used for drawing objects with a pen (In the table, P represents pen color, O represents original color on the target device, B represents black color, W represents white color, and the following symbols represent bit-wise NOT, AND, OR and XOR operations respectively: ~, &, |, ^):

(Table omitted)

To set the drawing mode, we need to call function CDC::SetROP2(...), which has the following format:

```
int CDC::SetROP2(int nDrawMode);
```

The function has only one parameter, which specifies the new drawing mode. It could be any of the modes listed in the above table.

Storing Data

When the user finishes drawing a line, the starting and ending points will be stored in the document. At this time, instead of drawing the new line directly to the window, we need to update the client window and let the function CView::OnDraw(...) be called. In this function, all the lines added previously will be drawn again, so the client will always become up-to-date.

We must override function CView::OnDraw(...) to implement client window drawing for an application. This is because the window update may happen at any time. For example, when the application is restored from the icon state, the whole portion of the window will be painted again. The system will draw the non-client area such as caption bar and borders, and it is the application's responsibility to implement client area drawing, which should be carried out in function CView::OnDraw(...) In MFC, if the application does not implement this, the client window will be simply painted with the default color. As a programmer, it is important for us to remember that when we output something to the window, it will not be kept there forever. We must redraw the output whenever the window is being updated.

This forces us to store every line that has been drawn by the user in document, and redraw all of them when necessary. This is the basic document/view structure of MFC: storing data in CDocument derived class, and representing it in function CView::OnDraw(...).

In the sample, the lines are stored in an array declared in the document class CGDIDoc. Also, some new functions are declared to let the data be accessible from the view:

(Code omitted)

A CPtrArray type variable m_paLines is declared for storing lines. Class CPtrArray allows us to add and delete pointer type objects dynamically. This class encapsulates memory allocation and release, so we do not have to worry about memory management when adding new elements. Three new public member functions are also declared, among them CGDIDoc::AddLine(...) can be used to add a new line to m_paLines, CGDIDoc::GetLine(...) can be used to retrieve a specified line from m_paLines, and CGDIDoc:: GetNumOfLines() can be used to retrieve the total number of lines stored in m_paLines. Here lines are stored in CRect type variables, usually this class is used to store rectangles. In the sample, the rectangle's upper-left and bottom-right points are used to represent a line.

Although class CPtrArray can manage the memory allocation and release for storing pointers, it does not release the memory for the stored objects. So in class CGDIDoc's destructor, we need to delete all the objects stored in the array as follows:

(Code omitted)

Here we just delete the first object and remove it from the array repeatedly until the size of the array becomes 0.

In CView derived class, we can call function CView::GetDocument() to obtain a pointer to the document and use it to access all the public variables and functions declared there.

Recording One Line

When the left button is pressed down, we need to trace the mouse's movement until it is released. During this period, it is possible that mouse may move outside the client window. To enable receiving mouse message even when it is outside the client window, we need to call function CWnd::SetCapture() to set window capture. This will direct all the mouse related messages to the window that has the capture no matter where the mouse is. After the left button is released, we need to call API function ::ReleaseCapture() to release the capture.

In class CGDIView, some new variables are declared to record the starting and ending points of the line and the window's capture state:

(Code omitted)

Variables m_ptStart and m_ptEnd are used to record the starting and ending points of a line, m_bCapture indicates if the window has the capture. Variable m_bNeedErase indicates if there is an old outline that needs to be erased. After the user presses down the left button, the initial mouse position will be stored in variable m_ptStart. Then as the user moves the mouse with the left button held down, m_ptEnd will be used to store the updated mouse position. A new line is defined by both m_ptStart and m_ptEnd. The only situation that we don't need to erase the old line is when m_ptEnd is first updated (before this, it does not contain valid data).

Two Boolean type variables are initialized in the constructor of CGDIView:

```
CGDIView::CGDIView()

{

m_bCapture=TRUE;

m_bNeedErase=FALSE;
```

}

Message handlers of WM_LBUTTONDOWN, WM_MOUSEMOVE and WM_LBUTTONUP can all be added through using Class Wizard. In the sample, these member functions are CGDIView::OnLButtonDown(...), CGDIView::OnMouseMove(...) and CGDIView::OnLButtonUp(...) respectively.

Function CGDIView::OnLButtonDown(...) is implemented as follows:

(Code omitted)

When the left button is pressed down, we need to store the current mouse position in m_ptStart. The second parameter of CView::OnLButtonDown(...) is the current mouse position measured in window's own coordinate system, which can be stored directly to m_ptStart. After that, we can set window capture and set flag m_bCapture.

When mouse is moving, we need to check if the left button is being held down. This can be implemented by examining the first parameter (nFlags) of function CView::OnMouseMove(...). If its MK_LBUTTON bit is set, the left button is being held down. We can also check other bits to see if SHIFT, CTRL key or mouse right button is being held down.

If the left button is being held down, we need to draw the outline using the dotted pen. If there already exists an old outline, we need to erase it before putting a new one. To draw a line, we need to first call function CDC::MoveTo(...) to move the DC's origin to the starting point of the line, then call CDC::LineTo(...) to complete drawing. We don't need to call CDC::MoveTo(...) each time when drawing several connected line segments continuously. After function CDC::LineTo(...) is called, the DC's origin will always be updated to the end point of that line. The following function shows how the mouse moving activities are handled in the sample:

(Code omitted)

We declare CClientDC type variable to obtain the device context of the client window. First function CPen::CreatePen(...) is called to create a dotted pen with black color. Then this pen is selected into the device context and the old pen is stored in ptrPenOld. Next, the device context's drawing mode is set to R2_XORPEN, and the original drawing mode is stored in nMode. If this is the first time the function is called after the user pressed mouse's left button, m_bNeedErase flag must be FALSE. In this case we need to set it to TRUE. Otherwise if m_bNeedErase is TRUE, we need to first draw the old outline (This will erase the outline). Next the current mouse position is stored in variable m_ptEnd, and CDC::MoveTo(...), CDC::LineTo(...) are called to draw the new outline. Finally device context's old drawing mode is resumed. Also, the old pen is selected back into the DC (which will also select the new pen out of the DC).

For WM_LBUTTONUP message handler, we also need to erase the old line if necessary. Because the new line is fixed now, we need to add new data to the document and update the client window. The following function shows how this message is handled:

(Code omitted)

Here a dotted pen is created again to erase the old line. Then function CView::GetDocument() is called to obtain a pointer to the document. After the line information is retrieved from the mouse position, function CGDIDoc::AddLine(...) is called to add new data to the document. Then flag m_bNeedErase is cleared, and window capture is released. Finally function CWnd::Invalidate() is called to update the client window. By default, this action will cause function CView::OnDraw(...) to be called.

For SDI and MDI applications, function CView::OnDraw(...) will be added to the project at the beginning by Application Wizard. In order to implement our own interface, we need to rewrite this member function as follows:

(Code omitted)

In the above function CPen::CreatePen(...) is called to create a red solid pen whose width is 1 device unit. Since the window DC is passed as a parameter to this funciton, we do not need to call CWnd::GetDC() or declare CClientDC type variable to obtain window's device context. After the pen is selected into the DC, the drawing mode is set to R2_COPYPEN, which will output the pen's color to the target device. Next function CGDIDoc::GetNumOfLines() is called to retrieve the total number of lines stored in the document. Then a loop is used to draw every line in the client window. Finally the DC's original drawing mode is resumed and the new pen is selected out of it.

8.2 Rectangle and Ellipse

It is easy to implement rectangle drawing after we understand the previous sample. To draw a rectangle, we need to call function CDC::Rentangle(...) and pass a CRect type value to it. The rectangle's border will be drawn using current pen selected by the DC, and its interior will be filled with the currently selected brush.

We can declare a brush type variable using class CBrush, and create various kind of brushes by calling any of the following functions: CBrush::CreateSolidBrush(...), CBrush::CreateHatchBrush(...), CBrush::CreatePatternBrush(...). In the sample, only solid brush is used. In Windows(, there are several pre-implemented default GDI objects that can be retrieved and used at any time. These objects include pens, brushes, fonts, etc. We can get any object by calling ::GetStockObject(...) API function. The returned value is a handle that could be attached to a GDI variable (declared by MFC class) of the same type. For example, the following code shows how to obtain a gray brush and attach it to a CBrush type variable:

```
CBrush brush;

brush.Attach(::GetStockObject(GRAY_BRUSH));

//After some drawing

brush.Detach();
```

We need to detach the object before the GDI variable goes out of scope.

When a rectangle is not finally fixed, we may want to draw only its border and leave its interior unpainted. To implement this, we can select a NULL (hollow) brush into the device context. A hollow brush can be obtained by calling function ::GetStockObject(...) using HOLLOW_BTRUSH or NULL_BRUSH flag.

Sample 8.2-1\GDI demonstrates how to implement an interactive environment to let the user draw rectangles. It is an SDI application generated by Application Wizard. Like what we implemented in sample 8.1\GDI, first some member variables and functions are declared in the document for storing rectangles:

(Code omitted)

In the destructor, all the objects in array m_paRects are deleted:

(Code omitted)

In class CGDIView, some new variables are declared, they will be used to record rectangles, erasing state and window capture state:

(Code omitted)

Two Boolean type variables are initialized in the constructor:

(Code omitted)

Message handlers for WM_LBUTTONDOWN, WM_MOUSEMOVE and

WM_LBUTTONUP are added to class CGDIView through using Class Wizard. They are implemented as follows:

(Code omitted)

Also, function CGDIView::OnDraw(...) is modified as follows:

(Code omitted)

With the above implementation, the application will be able to let the user draw rectangles.

With only minor modifications we can let the application draw ellipses. Sample 8.2-2\GDI demonstrates how to implement interactive ellipse drawing. It is based on sample 8.2-1\GDI.

To draw an ellipse, we need to call function CDC::Ellipse(...)and pass a CRect type value to it. This is exactly the same with calling function CDC::Rectangle(...). So in the previous sample, if we change all the "Rectangle" keywords to "Ellipse", the application will be implemented to draw ellipses instead of rectangles.

In sample 8.2-2\GDI, function CDC::Ellipse(...) is called within CGDIView::OnMouseMove(...) and CGDIView::OnDraw(...). The following shows the modified portion of two functions:

(Code omitted)

In CGDIView::OnDraw(...), function CBrush::CreateSolidBrush(...) is also changed to CBrush:: CreateHatchBrush(...). By doing this, a different brush will be used to fill the interior of ellipse.

8.3 Curve

We can call function CDC::PolyBezier(...) to draw curves. This function has two parameters:

```
BOOL CDC::PolyBezier(const POINT* lpPoints, int nCount);
```

The first parameter lpPoints is a pointer to an array of points, which specify the control points that can be used for drawing a curve. The second parameter nCount specifies how many control points are included in the array. We need at least four control points to draw a curve, although the last three points could be the same (In this case, a straight line will be drawn).

Sample 8.3\GDI demonstrates how to implement an interactive environment that can let the user draw curves. It is a standard SDI application generated by Application Wizard. First new variable and functions are declared in the document, which will be used to store the data of curves:

(Code omitted)

We use a CDWordArray type variable m_dwaPts to record control points, this class can be used to record DWORD type value, which is 32-bit integer. Because a point contains two integers, we need two DWORD type variables to store one point. So in CGDIDoc::AddPoint(...), function CDWordArray::Add(...) is called twice to add both x and y coordinates to the array. Function CGDIDoc::GetNumOfPts() returns the number of control points, which is obtained through dividing the size of the array by 2. Function CGDIDoc::GetOnePt() returns a specified control point, which is obtained from two consecutive elements contained in array m_dwaPts.

Curve drawing is implemented in function CGDIView::OnDraw(...) as follows:

(Code omitted)

Since we need 4 control points to draw a curve, the number of curves we can draw should be equal to the number of points stored in array CGDIDoc::m_dwaPts divided by 4. In the function, a loop is used to draw each single curve. Within each loop, four control points are retrieved one by one, and stored in a local CPoint type array pt. After all four control points are retrieved, function CDC::PolyBezier(...) is called to draw the curve. Here, we also need to create a pen and select it into the DC before any drawing operation is performed.

The rest thing we need to implement is recording control points. In order to do this, we need to handle two mouse related messages: WM_LBUTTONUP and WM_MOUSEMOVE. In the sample, their message handlers are added through using Class Wizard, the corresponding member functions are CGDIView::OnLButtonUp(...) and CGDIView::OnMouseMove(...) respectively.

Since a curve needs four control points, we use mouse's left button up event to record them. In the application, a counter is implemented to count how many control points have been added. Before a new curve is added, this counter is set to 0. As we receive message WM_LBUTTONUP, the counter will be incremented by 1. As it reaches 4, we finish recording the control points, store the data in the document and update the client window.

In the sample, to implement curve drawing, some new variables are declared in class CGDIView as follows:

(Code omitted)

We are familiar with variables m_bCapture and m_bNeedErase. Here, variable m_ptCurve will be used to record temporary control points, and m_nCurrentPt will act as a counter.

Some of the variables are initialized in the constructor:

(Code omitted)

Since m_nCurrentPt starts from 0, we need to count till it reaches 3. In function CGDIView:: OnLButtonUp(...), if the value of m_nCurrentPt becomes 3, we will call CGDIDoc::AddPoint(...) four times to add the points stored in array m_ptCurve to the document, then update the client window. We also need to reset flags m_nCurrentPt, m_bNeedErase and m_bCapture because the drawing is now complete. The following is a portion of function CGDIView::OnLButtonUp(...) that demonstrates how to record the last point and store data in the document:

(Code omitted)

If m_nCurrentPt is 0, this means it is the first control point. In this case, we do not need to draw or erase anything. However, we need to set the window capture.

If m_nCurrentPt is not 0, we need to erase the previous curve, record the new point, assign its value to the rest points (This is for the convenience of drawing curve outline, because when we draw a curve, we always need four control points), and draw the curve outline.

Then the implementation of CGDIView::OnMouseMove(...) becomes easy. We just need to check if there is an existing curve outline. If so, we need to erase it before drawing a new curve outline. Otherwise we just draw the curve outline directly and set m_bNeedErase flag:

(Code omitted)

And that is all we need to do. If we compile and execute the application at this point, we will be able to draw curves using mouse.

8.4 Other Shapes

There are many other shapes that we can draw using device context. These shapes include chord, focus rectangle, round rectangle, pie, polygon, 3D rectangle, and many others. Sample 8.4\GDI demonstrates how to use member functions of CDC to draw different shapes. It is a standard SDI application generated by Application Wizard. In the sample, only function CGDIView::OnDraw(...) is modified, where we demonstrate how to draw the above-mentioned shapes:

(Code omitted)

The following is the explanation of the functions used above:

BOOL CDC::TextOut(int x, int y, const CString& str);

This function outputs a text string at the specified x-y coordinates. The string is stored in str parameter. Text color and background color can be set by using functions CDC::SetTextColor(...) and CDC::SetBkColor(...). Background mode (specifies if text has transparent or opaque background) can be set by using function CDC::SetBkMode(...).

BOOL CDC::Chord(LPCRECT lpRect, POINT ptStart, POINT ptEnd);

This function draws a chord formed from an ellipse and a line segment. Parameter lpRect specifies the bounding rectangle of an ellipse, ptStart and ptEnd specify the starting and ending points of a line segment (they do not have to be on the ellipse). The border of the chord will be drawn using the currently selected pen, and the interior will be filled with the currently selected brush (Figure 8-1).

void CDC::DrawFocusRect(LPCRECT lpRect);

This function draws a rectangle specified by lpRect parameter. The rectangle will have a dotted border. If we call this function twice for the same rectangle, the rectangle will be removed. This is because when drawing the rectangle, the function uses bit-wise XOR mode.

BOOL CDC::RoundRect(LPCRECT lpRect, POINT point);

The function draws a rectangle specified by lpRect with rounded corners. Parameter point specifies width and the height of the ellipse that will be used to draw rounded corners (Figure 8-2). The border of the rectangle will be drawn with currently selected pen and its interior will be filled with the currently selected brush.

BOOL CDC::Pie(LPCRECT lpRect, POINT ptStart, POINT ptEnd);

The function draws a pie that is formed from an ellipse and two line segments. The ellipse is specified by parameter lpRect, and the two line segments are formed by the center of ellipse and ptStart, ptEnd respectively (Figure 8-3). The pie will be drawn in the counterclockwise direction. The border of the pie will be drawn using currently selected pen and its interior will be filled with the currently selected brush.

```
BOOL CDC::Polygon(LPPOINT lpPoints, int nCount);
```

This function draws a polygon. Parameter lpPoints is an array of points specifying the vertices of the polygon. Parameter nCount indicates the number of vertices. The border of the polygon will be drawn using currently selected pen and its interior will be filled with the currently selected brush.

```
void CDC::Draw3dRect(LPCRECT lpRect, COLORREF clrTopLeft, COLORREF
clrBottomRight);
```

This function draws a rectangle specified by lpRect. Its upper and left borders will be drawn using color specified by clrTopLeft, and its bottom and right borders will be drawn using color specified by clrBottomRight.

8.5 Flood Fill

Flood fill is very useful for applications such as graphic editors. By using this method, we can easily fill an irregular area with a specified color. In class CDC, there are two member functions that can be used to implement flood fill:

```
BOOL CDC::FloodFill(int x, int y, COLORREF crColor);
```

```
BOOL CDC::ExtFloodFill(int x, int y, COLORREF crColor, UINT nFillType);
```

There is a slight difference between the two functions. For function CDC::FloodFill(...), the filling starts from the point that is specified by parameters x and y using the brush being selected by the DC, and stretches out to all directions until a border whose color is the same with parameter crColor is encountered. The second function allows us to select filling mode, which is defined by parameter nFillType. Here we have two choices: FLOODFILLBORDER and FLOODFILLSURFACE. The first filling mode is exactly the same with that of function CDC::FloodFill(...). For the second mode, the filling starts from the point specified by parameters x and y, and stretches out to all directions, fills all the area that has the same color with crColor, until a border with different color is encountered.

Samples 8.5-1\GDI and 8.5-2\GDI demonstrate how to implement flood fill. They are based on sample 8.1\GDI. In the samples a new command is added to the application, it can be used by the user to fill any closed area with gray color. This closed area can be formed from the lines drawn by the user.

First a button ID_FLOODFILL is added to the tool bar IDR_MAINFRAME. We will use this button to indicate if the application is in the line drawing mode or flood filling mode. By default the button will stay in its normal state, at this time the user can use mouse to draw lines in the client window. If the user clicks this button, the application will toggle to flood filling mode, at this time, if the user clicks mouse within the client window, flood filling will happen.

In the sample, both WM_COMMAND and UPDATE_COMMAND_UI message handlers are added for command ID_FLOODFILL, the corresponding functions are CGDIDoc::OnFloodfill() and CGDIDoc:: OnUpdateFloodfill(...) respectively. Also, a Boolean type variable m_bFloodFill is declared in class CGDIDoc, which is used to indicate the current mode of the application (line drawing mode or flood filling mode):

(Code omitted)

Function CGDIDoc::GetFloodFill() allows m_bFloodFill to be accessed outside class CGDIDoc. Variable m_bFloodFill is initialized in the constructor of class CGDIDoc as follows:

```
CGDIDoc::CGDIDoc()
{
m_bFloodFill=FALSE;
}
```

Functions CGDIDoc::OnFloodfill() and CGDIDoc::OnUpdateFloodfill(...) are implemented as follows:

```
void CGDIDoc::OnFloodfill()
{
m_bFloodFill=m_bFloodFill ? FALSE:TRUE;
}
void CGDIDoc::OnUpdateFloodfill(CCmdUI* pCmdUI)
```

```
{

pCmdUI->SetCheck(m_bFloodFill);

}
```

We need to implement flood filling in response to left button up events when CGDIDoc::m_bFloodFill is TRUE. So we need to modify function CGDIView::OnLButtonDown(...). In the sample, we first check flag CGDIDoc::m_bFloodFill. If it is set, we create a gray brush, select it into the DC and implement the flood filling. Otherwise we prepare for line drawing as we did before:

(Code omitted)

In the other two mouse message handlers CGDIView::OnMouseMove(...) and CGDIView:: OnLButtonUp(...), we also need to check flag CGDIDoc::m_bFloodFill. If it is not set, we can proceed to implement line drawing:

(Code omitted)

This will enable flood filling.

Sample 8.5-2\GDI is exactly the same with sample 8.5-1\GDI except that here we use function CDC:: ExtFloodFill(...) instead of CDC::FloodFill(...). The following is the difference between the function calling in two samples:

8.5-1\GDI:

```
dc.FloodFill(point.x, point.y, RGB(255, 0, 0));
```

8.5-2\GDI:

```
dc.ExtFloodFill(point.x, point.y, dc.GetPixel(point), FLOODFILLSURFACE);
```

We call function CDC::GetPixel(...) to retrieve the color of the pixel specified by point, and fill the area that has the same color with it. The flood filling will stretch out to all directions until borders with different colors are reached. For samples 8.5-1\GDI and 8.5-2\GDI, we don't see much difference between function CDC::ExtFloodFill(...) and CDC::FloodFill(...). However, in the sample, if the lines can be drawn with different colors, we need to use CDC::ExtFloodFill(...) rather than CDC::FloodFill(...) to implement flood filling.

8.6 Pattern Brush

Pattern brush is very useful in filling an area with a specific pattern. We already know how to create a hatch brush, which could have several simple patterns. By using pattern brush, we are able to create a brush with any custom pattern.

The pattern brush can be created from a bitmap with a dimension of 8(8. If we use a larger bitmap image, only the upper-left portion of the image (with a size of 8(8) will be used. The bitmap used to create pattern brush must be stored in a CBitamp declared variable.

The simplest way of implementing pattern brush is to prepare a bitmap resource, load it with a CBitmap type variable, then create the brush. The function that can be used to create pattern brush is CBrush:: CreatePatternBrush(...), which has the following format:

```
BOOL CBrush::CreatePatternBrush(CBitmap* pBitmap);
```

Sample 8.6\GDI demonstrates how to create and use pattern brush. It is based on sample 8.2-1\GDI. In the new sample, pattern brush is used to fill the interior of the rectangles instead of solid brush.

First an 8(8 bitmap resource IDB_BITMAP_BRUSH is added to the application. Here function CGDIView:: OnDraw(...) is modified as follows: 1) A new local CBitmap type variable bmp is declared, which is used to call function CBitmap::LoadBitmap(...) for loading bitmap IDB_BITMAP_BRUSH. 2) The original statement brush.CreateSolidBrush(...) is changed to brush.CreatePatternBrush(...). The following is the modified function CGDIView::OnDraw(...):

(Code omitted)

With the above change, we can see the effect of pattern brush.

8.7 Color Approximation

Palette Device vs. Non-Palette Device

When creating a pen or a brush, we need to specify a color. The pen and brush will use this specified color to perform drawing operations. However, sometimes it is impossible for the system to create the exact same color as we specified, because the actual color that can be displayed on the screen depends on the hardware limitations. The total number of colors that can be displayed at any time depends on the screen resolution and the amount of video memory. For example, if we have a screen whose resolution is 1024(768, there are altogether

786432 pixels. If we use red, green and blue combination to specify a pixel, and each of the three basic colors (red, green and blue) can range from 0 to 255 (256 steps), we nee 3 bytes (24 bits) to specify one pixel on the screen. For screen with a resolution of 1024(768, the total memory needed is 786,432(3=2,359,296=2.4 Mega Bytes (In reality, situation is more complex so a video card of this type requires more memory).

If the hardware has this amount of memory, there will be no problem in displaying any color specified by the R, G, B combination. This kind of device is called non-palette device. The advantage of non-palette device is that it is fast and does not cause color distortion. The disadvantage is obvious: it is expensive.

Contrary to the above approach, palette devices use fewer bytes to represent a pixel. A very common approach is to represent each pixel using only one byte (8 bits). In this case, with the above assumption, the total amount of video memory needed will be 786,432 bytes.

However, 8 bits is not enough to specify all possible R, G, B (each ranges from 0 to 255) combinations. The solution here is to use a color table, which contains 256 different R, G, B combinations. In this case, the data stored in the video memory actually represents an index to the color table. This kind of devices is called palette device. Although it introduces much inconvenience to the programmer and may cause color distortion, it is still widely used in all types of systems because of its inexpensive price.

Color Approximation

Since the index has only 8 bits, the size of this color table is limited to contain no more than 256 colors. In Windows(, in order to maintain some standard colors (i.e., the caption bar color, border color, menu color, etc.), some entries of this color table are reserved for solely storing system colors. The colors stored in these entries are called Static Colors. For the rest entries of the color table, any application may fill them with custom colors. When we specify an R, G, B combination and use it to draw geometrical objects, the actual color appeared on the screen depends on the available colors contained in the color table. If the specified color can not be found in the color table, Windows(uses two different approaches to do the color approximation: for brush, it uses dithering method to simulate the specified color using the colors that can be found in the color table (For example, if color gray can not be found in the color table, the system combines black and white to simulate it); for pen, the n

earest color that can be found in the color table will be used instead of the specified color.

Sample

Sample 8.7-1\GDI and 8.7-2\GDI demonstrate two different color approximation approaches. Sample 8.7-1\GDI is a standard SDI application generated from Application Wizard. In function CGDIView:: OnDraw(...), the client window area is painted with blue colors that gradually change from dark blue (black) to bright blue. The GDI object used here is brush.

We need to create 256 different brushes using colors from RGB(0, 0, 0), RGB(0, 0, 1), RGB(0, 0, 2)... to RGB(0, 0, 255). Also, we need to divide the client window into 256 rows. For each row, a different brush can be used to fill it. This will generate a visual effect that the color changes gradually from dark blue to bright blue. The following is the modified function CGDIView::OnDraw(...):

(Code omitted)

First function CWnd::GetClientRect(...) is called to retrieve the dimension of the client window, which is stored in variable rect. Then the rectangle's vertical size is shrunk to 1/256 of its original size. Next a 256-step loop is used to fill the client window. Within each loop, a different blue brush is created, and function CDC::FillRect(...) is called to fill the rectangle. Before calling this function, we do not need to select the brush into DC, this is because the second parameter of function CDC::FillRect(...) is a CBrush type pointer (The DC selection happens within the function). The difference between function CDC:: Rectangle(...) and CDC::FillRect(...) is that the former will draw a border with the currently selected pen while the latter does not draw the border.

Sample 8.7-2\GDI is based on sample 8.7-1\GDI. Here within function CGDIView::OnDraw(...), pens with different blue colors are created and used to draw lines for painting each row. The following is the modified function CGDIView::OnDraw(...):

(Code omitted)

Instead of creating brushes, solid pens are created to paint the client window area. The width of these pens is the same with the height of each row, and their colors change gradually from dark blue to bright blue. Within each loop, before calling functions to draw a line, we select the pen into DC, after the line is drawn, we select the pen out of the DC. Because a new pen is created in each loop, function CPen::DeleteObject() is called at the end of the loop. This will destroy the pen so that variable pen can be initialized again.

Adjusting Display Settings

The two programs must be run on a palette device with a setting of "256 Color" in order to see the effects of color approximation. Generally this can be adjusted through changing the settings of the system.

We can do this by first opening application Control Panel (Click "Start" menu, go to "Settings", select "Control Panel"). Then we ned to double click icon "Display" and click "Setting" tab from the popped up property sheet. There are a lot of choices in the combo box that is labeled "Color Palette". Usually it contains entries such as "24bit", "256 color", "16 Color", etc. The actual available selections depend on the capability of the video card. If we select "24bit" true color setting, no dithering effect will be seen in the system. Here we need to choose "256 color" setting in order to test our samples (Figure 8-4).

Results

Figure 8-5 and Figure 8-6 show the results from the two color approximation approaches (Please note that if the samples are executed on non-palette device, we may not see the approximation effect).

User can avoid color approximation by increasing the color depth of the system. One simple way to do so is to reduce the screen resolution (e.g. from 1024(768 to 800(600): after the total number of pixels is reduced, the number of colors supported by the system will probably increase.

As a programmer, we need to prepare for the worst situation and make our application least susceptible to the system setting. To achieve this, we need to implement local palette.

8.8 Logical Palette

Palette

Palette is another type of GDI objects, it encapsulates a color palette that can be used to store custom colors in an application. Although programmer can use logical palette like using an actual palette, it is not a real palette. Within any system, there is only one existing palette, which is the physical palette (or system palette). This is why the palettes created in the applications are called logical palettes. The advantage of using a logical palette is that the colors defined in the logical palette can be mapped to the system palette in an efficient way so that least color distortion can be achieved. There is no guarantee that all the colors implemented in the logical palette will be displayed without color distortion. Actually, the ultimate color realizing ability depends on the hardware. For example, if our hardware support only 256 colors and we implement a 512 logical palette, some colors defined in the logical palette will inevitably be distorted if we display all the 512 colors on the screen at the same time.

Color Mapping

When we implement a logical palette, operating system maps the colors contained in the logical palette to the system palette using the following method: for every entry in the logical palette, the system first finds out if there exists a color in the system palette that is exactly the same with the color contained in this entry. If so, it will be mapped to the corresponding entry of the system palette. If no such entry is found in the system palette, the system will find out if there is any entry in the system palette that is not occupied by any logical palette. If such an entry exists, the color in the logical palette will be filled into that entry. If there is no such entry available, the system find out the nearest color in the system palette and map the entry in the logical palette to it.

For a 256 color system, the operating system reserve 20 static colors as system colors, which can not be used to fill new colors. This assures that the default colors of window border, title, button do not change when we implement logical palettes. These static colors always occupy the first 10 and last 10 entries of the system palette. The rest 236 entries can be used fill any color.

Foreground and Background Palette

When creating a logical palette, we can either implement it as a "foreground" or a "background" palette. For foreground logical palette, the operating system maps the colors contained in the logical palette to the system palette only when the application is active (has the current focus). For the background logical palette, the operating system also tries to map the colors contained in the logical palette to the system palette even it is not active (does not have the current focus). In this case, the application in the foreground has the highest priority in mapping its colors to the system palette. If there is still entries left unused, they may be occupied by the applications in the background. If no such entry is left, the colors in the background palette will be mapped to the system palette by finding the nearest colors.

Creating Logical Palette

To create a logical palette, we need to call function CPalette::CreatePalette(...), which has the following format:

```
BOOL CPalette::CreatePalette(LPLOGPALETTE lpLogPalette);
```

The only parameter of this function is a LOGPALETTE type pointer. The following is the format of structure LOGPALETTE:

```
typedef struct tagLOGPALETTE {

WORD palVersion;
WORD palNumEntries;

PALETTEENTRY palPalEntry[1];

} LOGPALETTE;
```

This structure has three members: palVersion specifies the Windows(version of this structure, which must be set to 0x300; palNumEntries specifies the total number of entries contained in this palette, and palNumEntries is the first element of a PALETTEENTRY type array, which stores the color table. Structure PALETTEENTRY has four members:

```
typedef struct tagPALETTEENTRY {

BYTE peRed;

BYTE peGreen;

BYTE peBlue;

BYTE peFlags;

} PALETTEENTRY;
```

Members peRed, peGreen and peBlue specify RGB intensities, and peFlags can be assigned NULL if we want to create a normal palette (we will see how to set this flag to create a special palette later). To create a logical palette, we need to allocate enough buffers for storing LOGPALETTE structure and colors, and call function CPalette::CreatePalette(…).

Using Logical Palette

Like other GDI objects, we must select a logical palette into the DC in order to use it. To select palette into the DC, we need to call function CDC::SelectPalette(…), which has two parameters:

```
CPalette *CPalette::SelectPalette(CPalette* pPalette, BOOL bForceBackground);
```

The first parameter is a CPalette type pointer, which stands for the logical palette we are going to use. The second parameter is a Boolean type variable indicating if the palette will be selected as a background palette or not.

We need to select the palette out of the DC after using it.

Realizing Palette

The color mapping does not happen if we do not realize the logical palette. We need to realize the palette in the following situations: 1) After the palette is created and about to be used. 2) After the colors in the logical palette have changed. 3) After the colors contained in the system palette have changed. 4) After the application has regained the focus.

To realize the palette, we need to call function CDC::RealizePalette() to force the colors in the logical palette to be mapped to the system palette. The format of this function is very simple:

```
UINT CDC::RealizePalette();
```

The returned value indicates how many entries were mapped to the system palette successfully.

Macro PALETTEINDEX

When a logical palette is selected into the DC, we need to use index to the color table to reference a color in the palette. In this case, we can use macro PALETTEINDEX to convert an index to R, G, B combination.

Sample

Sample 8.8-1\GDI and 8.8-2\GDI are based on sample 8.7-1\GDI and 8.7-2\GDI respectively. They demonstrate how to implement logical palette to avoid color distortion. In the two samples, the logical palettes contain all colors that will be used to paint the client window. These colors will be mapped to the colors contained in the system palette.

In both samples, first a CPalette type variable m_palDraw is declared in class CGDIView as follows:

```
class CGDIView : public CView

{

......

protected:
```

```
CPalette m_palDraw;

......

};
```

The palette is created in the constructor of class CGDIView as follows:

(Code omitted)

In the function a LOGPALETTE type pointer is declared, we need to use it to store the buffers allocated for creating the logical palette. The buffers are allocated through using "new" method. The following isthe formulae that is used to calculate the size of the total bytes needed for creating logical palette:

size of structure LOGPALETTE + (number of entries-1)((size of structure PALETTEENTRY)

Next the palette entries are filled with 256 colors ranging from dark blue to bright blue. After the logical palette is created, these buffers can be released.

The following is the updated function CGDIView::OnDraw(...) in sample 8.8-1\GDI:

(Code omitted)

Each time the client window needs to be painted, we first call function CDC::SelectPalette(...) to select the logical palette into the device context and call function CDC::RealizePalette() to let the colors in the logical palette be mapped to the system palette. When creating the brushes, instead of using RGB macro to specify an R, G, B combination, we need to use PALETTEINDEX macro to indicate a color contained in the logical palette.

Sample 8.8-2\GDI is implemented almost in the same way: first a CPalette type variable m_palDraw is declared in class CGDIView, and the palette is created in its constructor. In function CGDIView::OnDraw(...), before the client window is painted, the logical palette is selected into the device context. The following is the modified function CGDIView::OnDraw(...):

(Code omitted)

This function is similar to that of sample 8.8-1\GDI: when creating a pen, we use PALETTEINDEX macro to indicate a color rather than using an R, G, B combination.

The samples will improve a lot with the above implementations (If the samples are executed on non-palette device, there will be no difference between the samples here and those in the previous section). However, we may still notice a tiny step between two contiguous blues. This is because for a video device of this type (256 color), usually the color depth is 18 bits on the hardware level. This means red, green and blue colors each uses only 6 bits (instead of 8 bits). So instead of displaying 256-level blue colors, only 64-level blue colors can be implemented.

8.9 Monitoring System Palette

We know that when a logical palette is realized, the colors contained in the system palette may change. If we have several applications each implementing a logical palette, the system palette will change if any application calls function CDC::RealizePalette(). In Windows(programming, there is a way for us to monitor the colors contained in the system palette: we can create a logical palette and explicitly map a logical palette entry to a fixed system palette entry instead of let the mapping be implemented automatically. By doing this, we have a way of knowing what colors are contained in certain entries of the system palette.

Remember in the previous samples, when stuffing structure PALETTEENTRY, we set NULL to its peFlags member. Actually, it can also be set to one of the following three values: PC_EXPLICIT, PC_NOCOLLAPSE and PC_RESERVED.

If peFlags is set to PC_EXPLICIT, we can create a palette whose entries are mapped directly to the specified entries in the system palette. In this case, the low order of structure PALETTEENTRY (the combination of peRed and peGreen members) indicates an index to the system palette, and member peBlue has no effect.

Because non-static entries in the system palette may change constantly, if we use such a palette to implement drawing, the colors we draw will also change constantly with the system palette.

Sample 8.9\GDI demonstrates how to implement such kind of palette. It is a standard SDI application generated by Application Wizard. In the sample, the palette is implemented and is used to paint the client window of the application. The client window is divided into 256 rectangles, and each rectangle is painted using the color contained in one of the system palette entry. So when a logical palette is realized from other application, we can see the changes from the client window.

In the sample, first a CPalette type variable m_palSys is declared in class CGDIView:

```
class CGDIView : public CView

{

……

protected:

CPalette m_palSys;

……

};
```

In the constructor, the logical palette is created as follows:

(Code omitted)

The procedure above is almost the same with creating a normal logical palette. The only difference here is that when stuffing structure PALETTEENTRY, the low order word of the structure is filled with an index to the system palette entry (from 0 to 255), and member peFlags is set to PC_EXPLICIT.

Function CGDIView::OnDraw(...) is implemented as follows:

(Code omitted)

The client area is divided into 256 (16(16) rectangles. For each rectangle, a brush using a specific logical palette entry is created. Then function CDC::Rectangle(...) is called to draw the rectangle. Because no pen is selected into the DC, the default pen will be used to draw the border of the rectangles.

The application can be compiled and executed at this point. There are 256 rectangles in the client window, each represents one color contained in the system palette. By paying attention to the first 10 and last 10 entries, we will notice that the colors contained in these entries never change (because they are static colors). Other colors will change from time to time as we open and close graphic applications. We can use samples 8.8-1\GDI or 8.8-2\GDI to test this.

Please note that if the sample is executed on non-palette device, the logical palette will not represent the system palette. This is because on the hardware level, palette does not exist at all.

8.10 Palette Animation

Flag PC_RESERVED

Another interesting flag we can use when creating a logical palette is PC_RESERVED, which can be used to implement palette animation. If we have a logical palette with this flag set for some entries, the colors in these entries will only be mapped to the unused entries of the system palette. If the mapping is successful, when we change the colors in the logical palette, the entries in the system palette will also change. If any portion of window is painted with such entries, this change will affect that portion. This is the reason why a logical entry with PC_RESERVED flag can not be mapped to an occupied entry.

The following table lists the difference between a normal logical palette entry and an entry with PC_RESERVED flag:

(Table omitted)

While we can change the color of any area in a window by painting it again (using a different brush or pen), the above mentioned method has two advantages:

1) If we have several areas painted with the same color, the new method will cause all of them to change at the same time once the old color is replaced with a new one in the logical palette. For the traditional method, we have to draw each area one by one to make this change, it takes longer time.

2) If we draw each area one by one, the change takes place in software level. For the new method, the color is filled to the system palette directly (This change happens at the hardware level), which is extremely fast.

Animation

With this method, it is very easy to implement an animation effect. For example, considering an array of four rectangles that are filled with the following four different colors respectively: red, green, blue and black. If we paint the four rectangles with green, blue, black, red next time, and blue, black, red, green next next time, and so on..., this will give us an impression that the rectangles are doing rotating shift. One way to implement this effect is to redraw four rectangles again and again using different colors (which means using different entries to draw the same rectangle again and again). Another way is to switch the colors in the palette directly.

Sample 8-10\GDI demonstrates how to implement palette animation. It is a standard SDI application generated from Application Wizard. In the sample, the client area is divided into 236 columns, each row is painted with a different color. The colors in the logical palette will be shifting all the time, and we will see that the colors in the client window will also shift accordingly.

Among 256 system palette entries, only 236 of them contain non-static colors, so in the sample, a logical palette with 236 entries is created. The colors contained in this palette change gradually from red to green, and from green to blue. Figure 8-7 shows the RGB combination of each entry (i.e., entry 0 contains RGB(255, 0, 0), entry 79 contains RGB(0, 255, 0)...).

Sample

First three variables are declared in class CGDIView:

```
class CGDIView : public CView
{
......
protected:
int m_nEntryID;
PALETTEENTRY m_palEntry[236*2-1];
CPalette m_palAni;
......
};
```

Variable m_palAni is used to create the animation palette, and array m_palEntry will be used to implement color shifting. When we shift the palette entries, it would be much faster if we use function memcpy(...) to copy all the entries in just one stroke. To achieve this, we keep all the colors in a PALETTEENTRY type array whose size is twice the logical palette size minus one. The original 236 colors are stored in entries 0, 1, 2... to 235, and entries 236, 237, 238...470 store the same colors as those contained in entries 0, 1, 2... 234. Also, we use variable m_nEntryID to indicate the current starting entry of the logical palette. For example, if m_nEntryID is 2, the logical palette should be filled with colors contained in entries 2 to 237 of m_palEntry. If m_nEntryID reaches 236, we need to reset it to 0. Figure 8-8 demonstrates this procedure.

First, 236 different colors are filled into entries from 0 to 235 for variable m_palEntry in the constructor of class CGDIView as follows:

(Code omitted)

This distribution of RGB values is the same with the graph shown in figure 8-8. Next, the colors contained in entries from 0 to 234 are copied to entries from 236 to end:

(Code omitted)

Next, we use entries from 0 to 235 contained in m_palEntry to create logical palette (using variable m_palAni), and assign 0 to variable m_nEntryID:

(Code omitted)

Note when filling array m_palEntry, we need to assign PC_RESERVED flag to member peFlags of structure PALETTEENTRY for every element. This flag will be copied to array pointed by lpLogPal when we actually create the palette. As we refill the palette entries again and again, these flags should remain unchanged all the time. Otherwise, the entries can not be changed dynamically.

To realize the animation, we need to implement a timer, and change the palette when it times out. The best place to start the timer is in function CView::OnInitialUpdate(), where the view is just created. In the sample, this function is added through using Class Wizard, and is implemented as follows:

```
void CGDIView::OnInitialUpdate()

{

SetTimer(TIMER_ANIMATE, 100, NULL);

CView::OnInitialUpdate();

}
```

Macro TIMER_ANIMATE is defined at the beginning of the implementation file, which acts as the ID of the timer:

```
#define TIMER_ANIMATE 500
```

We can use any integer as the timer ID.

Next, a WM_TIMER type message handler is added to class CGDIView through using Class Wizard, which is implemented as follows:

(Code omitted)

Like other drawing operations, before implementing palette animation, we must select the palette into target DC and realize it. After the animation is done, we need to select the palette out of DC. The palette animation is implemented through calling function CPalette::AnimatePalette(...), which has the following format:

```
void CPalette::AnimatePalette

(

UINT nStartIndex, UINT nNumEntries, LPPALETTEENTRY lpPaletteColors

);
```

The first parameter nStarIndex is the index indicating the first entry that will be filled with a new color, the second parameter nNumEntries indicates the total number of entries whose color will be changed, and the last parameter is a PALETTEENTRY type pointer which indicates buffers containing new colors.

If we call this function and change the contents of a logical palette, only those entries with PC_RESERVED flags will be affected.

The last thing we need to do is painting the client area using animation palette in function CGDIView:: OnDraw(...):

(Code omitted)

This is very straight forward, we just divide the client area into 236 columns and paint each column using one color contained in the logical palette. Note in function CGDIView::OnTimer(...), after palette animation is implemented, we do not need to call function CWnd::Invalidate() to update the client window.

Now the application can be compiled and executed. If we execute this application along with sample 8.9\GDI, we will see how system palette changes when the animation is undergoing (We can put sample 8.9\GDI in background to monitor the system palette): as the colors in the client window shifts, the colors in the system palette will also change.

Palette animation can also be used to implement special visual effect on bitmaps such as fade out. This function only works on palette type video device.

Please note that if the sample is executed on non-palette device, the animation effect can not be seen. This is because there is no palette on the hardware level.

8.11 Find Out Device Capability

There are many type of devices, each has a different capability. Some use palette to implement colors, others use 24 bits to store an RGB value directly. Before a program is about to run, it might be a good idea to find out the capabilities of hardware and use different approaches to implement colors.

In MFC, there is a function that can be used to detect the capabilities of a device:

```
int CDC::GetDeviceCaps(int nIndex);
```

By passing different flags to parameter nIndex, we can retrieve different attributes of a device. The following is a list of some important flags that can be used:

(Table omitted)

Sample 8.11\GDI demonstrates how to check the abilities of a device. It is a standard SDI application generated from Application Wizard. In the sample, ability checking is implemented in the initialization stage of the client window.

For this purpose, function CGDIView::OnInitialUpdate() is added to the application through using Application Wizard. In this function, various device abilities are checked and the result is displayed in a message box:

(Code omitted)

Here we check if the device is a palette device or not. If it is a palette device, we further find out the maximum number of colors it supports, the number of static colors reserved, and the actual color resolution for each pixel. Also, the device's horizontal and vertical sizes are retrieved.

This function is especially useful in finding out if the device is a palette device or not. With this information, we can decide if the logical palette should be used in the application.

The following lists the capabilities of certain device:

It is a palette device

The device supports 256 colors

There are 20 static colors

Color resolution is 18 bits

Horizontal size is 270 mm, 1024 pixels

Vertical size is 203 mm, 768 pixels

Summary:

1) Before drawing anything to a window, we must first obtain its device context. There are many ways of obtaining a window's DC (Calling function CWnd::GetDC(), declaring CClientDC or CWindowDC type variables, etc.).

2) A client DC can be used to paint a window's client area, and a window DC can be used to paint the whole window (client and non-client area).

3) Pen can be used to draw line, the border of rectangle, polygon, ellipse, etc. A pen can have different styles: solid pen, dotted pen, dashed pen, etc.

4) Brush can be used to fill the interior of rectangle, polygon, ellipse, etc. A brush can have different patterns: solid, hatched, etc.

5) We can use an 8(8 image to create pattern brush.

6) On a palette device, there are two color approaching methods: dithering and using the nearest color.

7) Logical palette can be used to avoid color distortion. To use a logical palette, we need to create it, select it into DC, and realize the palette.

8) System palette can be monitored by creating a logical palette whose entries are set to PC_EXPLICIT flags.

9) Palette animation can be implemented by creating a logical palette whose entries are set to PC_RESERVED flag.

10) The abilities of a device can be retrieved by calling function CDC::GetDeviceCaps().

BACK TO INDEX

Chapter 9 Font

Font is another very important GDI object, every application deals with font. Usually a system contains some default fonts that can be used by all the applications. Besides these default fonts, we can also install fonts provided by the thirty party. For word processing applications, using font is a complex issue. There are many things we need to take care. For example, when creating this type of applications, we need to think about the following issues: how to display a font with different styles; how to change the text alignment; how to add special effects to characters.

9.1 Outputting Text Using Different Fonts

When we implement a font dialog box, all the available fonts contained in the system will be listed in it. We can select font size, font name, special styles, and text color.

Like other GDI objects such as pen and brush, we need to create font with specific styles and select it into a DC in order to use it for outputting text. In MFC, the class that can be used to implement font is CFont. To create a font, we can call either CFont::CreateFont(...) or CFont::CreateFontIndirect(...), whose formats are listed as follows:

(Code omitted)

The first function has many parameters and the second one needs only a LOGFONT type pointer. The results of the two member functions are exactly the same, every style we need to specify for a font in the first function has a corresponding member in structure LOGFONT:

```
typedef struct tagLOGFONT{

LONG lfHeight;

LONG lfWidth;

LONG lfEscapement;
LONG lfOrientation;

LONG lfWeight;

BYTE lfItalic;

BYTE lfUnderline;

BYTE lfStrikeOut;

BYTE lfCharSet;

BYTE lfOutPrecision;

BYTE lfClipPrecision;

BYTE lfQuality;

BYTE lfPitchAndFamily;

TCHAR lfFaceName[LF_FACESIZE];

} LOGFONT;
```

Here, member lfFaceName specifies the font name; lfHeight and lfWidth specify font size; lfWeight, lfItalic, lfUnderline and lfStrikeOut specify font styles. Besides these styles, there are two other styles that can be specified: lfEscapement and lfOrientation.

Under Windows 95, lfEscapement and lfOrientation must be assigned the same value when a font is being created. If they are non-zero, the text will have an angle with respect to the horizontal border of the window when it is displayed (Figure 9-1). To display text this way, we must assign the angle to both lfEscapement and lfOrientation when creating the font, the unit of the angle is one tenth of a degree. Please note that only True Type fonts can have such orientation.

After a font is created, it can be selected into DC for outputting text. After the text output is over, it must be selected out of the DC. This procedure is exactly the same with other GDI objects.

Sample 9.1\GDI demonstrates how to create and use a font with specified styles. It is a standard SDI application generated by Application Wizard. In the sample, the user can choose any available font in the system and set its styles (bold, italic, underline, etc). The face name of the font will be displayed in the client window using the selected font, and the user can also set the escapement of the font.

Two variables are added to class CGDIDoc: CGDIDoc::m_fontDraw and CGDIDoc::m_colorFont. The first variable is declared as a CFont type variable, it will be used to create the font. The second variable is declared as a COLORREF type variable, it will be used to store the color of the text.

Besides font color, we also need to consider the background color of the text. A text can be displayed with either a transparent or opaque background. In the latter case, we can set the background to different colors. In order to display text in different styles, another Boolean type variable CGDIDoc:: m_bTransparentBgd is declared, it will be used to indicate if the background is transparent or opaque.

The following is the modified class CGDIDoc:

(Code omitted)

Besides the three new member variables, there are also three new member functions added to the class. These functions allow the information stored in CGDIDoc to be accessible outside the class, they are CGDIDoc ::GetCurrentFont(), CGDIDoc::GetFontColor() and CGDIDoc::GetBgdStyle().

The above variables are initialized in the constructor of class CGDIDoc:

(Code omitted)

When a DC is created, it selects the default font, pen, brush and other GDI objects. So here we create a DC that does not belong to any window, and call function CDC::GetCurrentFont() to obtain its currently selected font (which is the default font). Then function CFont::GetLogFont(...) is called to retrieve the font information, which is stored in a LOGFONT type object. With this object, we can create a system default font by calling function CFont::CreateFontIndirect(...). By default, the font color is set to black and the text background mode is set to transparent.

We need to provide a way of letting user modify the font styles. This can be easily implemented by using a font common dialog box. In the sample, two commands are added to the application: Font | Select and Font | Escapement and Orientation, whose IDs are ID_FONT_SELECT and ID_FONT_STYLE respectively. Also, message handlers are added through using Class Wizard, the corresponding functions are CGDIDoc::OnFontStyle() and CGDIDoc::OnFontSelect().

Function CGDIDoc::OnFontSelect() lets the user select a font, set its styles, and specify the text color. It is impelemented as follows:

(Code omitted)

A font common dialog is implemented to let the user pick up a font. If a font is selected, function CFontDialog::GetCurrentFont(...) is called to retrieve the information of the font, which is stored in a LOGFONT type object. Because member m_fontDraw is already initialized, we need to delete the old font before creating a new one. The font is created by calling function CFont::CreateFontIndirect(...). After this the color of the font is retrieved by calling function CFontDialog::GetColor(), and stored in the variable CGDIDoc::m_colorFont. Finally function CDocument::UpdateAllViews(...) is called to update the client window of the application.

Since font common dialog box does not contain escapement and orientation choices, we have to implement an extra dialog box to let the user set them. In the sample, dialog template IDD_DIALOG_STYLE is added for this purpose. Within this template, besides the default "OK" and "Cancel" buttons, there are two other controls included in the dialog box: edit box IDC_EDIT_ESP, which allows the user to set escapement angle; check box IDC_CHECK, which allows the user to select text background style (transparent or opaque). A new class CStyleDlg is added for this dialog template, within which two variables m_lEsp (long type) and m_bBgdStyle (Boolean type) are declared. Both of them are added through using Class Wizard, and are associated with controls IDC_EDIT_ESP and IDC_CHECK respectively.

Command Font | Escapement and Orientation is implemented as follows:

(Code omitted)

First function CFont::GetLogFont(...) is called to retrieve the information of the current font, which is then stored in a LOGFONT type object. Before the dialog box is invoked, its members CStyleDlg::m_lEsp and CStyleDlg::m_bBgdStyle are initialized so that the current font's escapement angle and background style will be displayed in the dialog box. After function CDialog::DoModal() is called, the new value of CStyleDlg::m_lEsp is stored back to members lfEscapement and lfOrientation of structure LOGFONT, and the new value of CStyleDlg::m_bBgdStyle is stored to variable CGDIDoc::m_bTransparentBgd. Then the old font is deleted and the new font is created. Finally, function CGDIDoc::UpdateAllViews(...) is called to update the client window.

When we call function CDocument::UpdateAllViews(...), the associated view's member function OnDraw(...) will be called automatically. So we need to modify this function to display the font specified by variable CGDIDoc::m_fontDraw. The following is the implementation of function CGDIView::OnDraw():

(Code omitted)

First, function CGDIDoc::GetCurrentFont() is called to retrieve the currently selected font from the document, then function CFont::GetLogFont(...) is called to retrieve the information of this font (the face name of the font will be used as the output string). Next, the font is selected into the target DC. Also, CGDIDoc::GetFontColor() is called to retrieve the current font color, and CDC::SetTextColor(...) is called to set the text foreground color. Then, CGDIDoc::GetBgdStyle(...) is called to see if the text should be drawn with an opaque or transparent background, and CDC::SetBkMode(...) is called to set the background style. Next, the text background color is set to the inverse of the foreground color by calling function CDC::SetBkColor(...) (If text background style is transparent, this operation has no effect). Finally, function CDC::TextOut(...) is called to display font's face name in the client window, and the font is selected out of the DC.

The default font displayed in the client window should be "System", which is not a True Type font. To see how a text can be displayed with different escapement angles, we need to choose a True Type font such as "Arial". Please note that the unit of the escapement angle is on tenth of a degree, so if we want to display the text vertically, escapement angle should be set to 900.

9.2 Enumerating Fonts in the System

Font Types

There are three type of fonts in Windows(system: raster font, vector font and True Type font. The difference among them is how the character glyph is stored for each type of fonts. For raster fonts, the glyph is simply a bitmap; for vector fonts, the glyph is a collection of end points that define the line segments; for true type fonts, the glyph is a collection of line and curve commands. The raster fonts can not be drawn in a scaled size, they are device dependent (the size of the output depends on the resolution of the device). The vector fonts and true Type Fonts are device independent because they are scalable, however, drawing True Type fonts is faster than drawing vector fonts. This is because the glyph of True Type fonts contains commands and hints.

Enumerating Font Family

We can find out all the font families installed in a system by calling API function ::EnumFontFamilies(...), which has the following format:

```
int EnumFontFamilies

(
HDC hdc, LPCTSTR lpszFamily, FONTENUMPROC lpEnumFontFamProc, LPARAM lParam

);
```

The first parameter hdc is the handle to a device context, which can be obtained from any window.

The second parameter lpszFamily specifies the font family name that will be enumerated, it could be any of "Decorative", "Dontcare", "Modern", "Roman", "Script" and "Swiss". To enumerate all fonts in the system, we just need to pass NULL to it.

The third parameter is a pointer to callback function, which will be used to implement the actual enumeration. This function must be provided by the programmer.

The final parameter lParam is a user-defined parameter that allows us to send information to the callback function.

The callback function must have the following format:

```
int CALLBACK EnumFontFamProc

(

ENUMLOGFONT FAR *lpelf, NEWTEXTMETRIC FAR *lpntm, int FontType, LPARAM lParam

);
```

Each time a new font family is enumerated, this function is called by the system. So the function will be called for all the available font types (e.g. if there are three types of fonts in the system, this funciton will be called three times by the system). The font's information is stored in an ENUMLOGFONT type object that is pointed by lpelf, and the font type is specified by FontType parameter. We can check RASTER_FONTTYPE or TRUETYPE_FONTTYPE bit of FontType to judge if the font is a raster font or a true type font. The final parameter lParam will be used to store the information that is passed through the user defined parameter (lParam in the function ::EnumFontFamilies(...)).

Sample 9.2\GDI demonstrates how to enumerate all the valid fonts in the system. It is a standard SDI application generated by Application Wizard. The application will display all the available fonts in the client window after it is executed. Because there are many types of fonts, the view class of this application is derived from CScrollView (This can be set in the final step of Application Wizard).

First, an int type array is declared in class CGDIView:

```
class CGDIView : public CScrollView

{

protected:

int m_nFontCount[3];

……

}
```

The first element of this array will be used to record the number of raster fonts in the system, the second and the third elements will be used to store the number of vector and true type fonts respectively. The array is initialized in the constructor as follows:

```
CGDIView::CGDIView()

{

m_nFontCount[0]=m_nFontCount[1]=m_nFontCount[2]=0;

}
```

We need to create the callback function in order to implement the enumeration. In the sample, a global function ::EnumFontFamProc(…) is declared as follows (This function can also be declared as a static member function):

```
int CALLBACK EnumFontFamProc(LPLOGFONT, LPNEWTEXTMETRIC, DWORD,
LPVOID);
```

This function is implemented as follows:

(Code omitted)

We will use user-defined parameter lParam in the function ::EnumFontFamilies(…) to pass the address of CGDIView::m_nFontCount into the callback function, so that we can fill the font's information into this array when the enumeration is undergoing. In the callback function, the address of CGDIView:: m_nFontCount is received by parameter pFontCount, which is then cast to an integer type pointer. The font type is retrieved by examining parameter FontType, if the font is a raster font, the first element of CGDIView:: m_nFontCount will be incremented; if the font is a true type font, the third element will be incremented; in the rest case, the font must be a vector font, and the second element will be incremented.

The best place to implement the font enumeration is in function CView::OnInitialUpdate(), when the view is first created. In the sample, a client DC is created and function ::EnumFontFamilies(...) is called. When doing this, we pass the address of CGDIView::m_nFontCount as a user-defined parameter:

(Code omitted)

Still, we need to display the result in function CView::OnDraw(...). In the sample, this function is implemented as follows:

(Code omitted)

We just display three lines of text indicating how many fonts are contained in the system for each different font family.

Enumerating Font

Apart from the above information (how many fonts there are for each font family), we may further want to know the exact properties of every font type (i.e., face name). To implement this, we need to allocate enough memory to store the information of all fonts. Here, the size of this buffer depends on the number of fonts whose properties are to be retrieved. Since each font need a LOGFONT structure to store all its information, we can use the following formulae to calculate the required buffer size:

(sizeof structure LOGFONT) * (number of fonts)

For this purpose, in the sample, another two variables are declared in class CGDIView as follows:

```
class CGDIView : public CScrollView

{

protected:

......
LPLOGFONT m_lpLf[3];
CFont *m_ptrFont;
……
}
```

Array m_lpLf will be used to store LOGFONT information, and m_ptrFont will be used to store CFont type variables. The variables are initialized in the constructor as follows:

```
CGDIView::CGDIView()
{
……
m_lpLf[0]=m_lpLf[1]=m_lpLf[2]=NULL;
m_ptrFont=NULL;
}
```

We need to provide another callback function to retrieve the actual information for each font type. In the sample, this callback function is declared and implemented as follows:

```
int CALLBACK EnumFontProc(LPLOGFONT, LPNEWTEXTMETRIC, DWORD, LPVOID);
```

(Code omitted)

Three static variables are declared here to act as the counters for each type of fonts. When this function is called, the information of the font is copied from lplf to the buffers allocated in CGDIView:: OnInitialUpdate(), whose address is passed through user-defined parameter.

In function CGDIView::OnInitialUpdate(), after the font families are enumerated, we need to allocate enough memory, implement the enumeration again for every single type of font:

(Code omitted)

After obtaining the information for each type of font, we create a font using this information by calling function CFont::CreateFontIndirect(...). The addresses of these font objects are stored in array CGDIView::m_ptrFont.

In function CGDIView::OnDraw(...), the face names of all fonts are output to the client window:

(Code omitted)

For each font family, all the font face names are listed. Three loops are used for this purpose. Within each loop, one of the enumerated font is selected into the target DC, and function CDC::TextOut(...) is called to output the font's face name to the window. To avoid text from overlapping one another, a local variable nYPos is used as the vertical orgin of the output text, which will increment each time after a line of text is output to the window.

Because the memory is allocated at the initialization stage, we need to free it when the application exits. In the sample, WM_DESTROY message handler is added to class CGDIView through using Class Wizard, and the corresponding member function is implemented as follows:

(Code omitted)

The application is now ready to enumerate all the available fonts in the system.

9.3 Output Text Using CDC::ExtTextOut(...)

Function CDC::ExtTextOut(...)

Usually we use function CDC::TextOut(...) to output text. There is another powerful function CDC:: ExtTextOut(...), which allows us to output the text to a specified rectange. We can use either transparent or opaque drawing mode. In the latter case, we can also specify a background color. Besides this, we can set the distances between neighboring characters of the text. One version of function CDC::ExtTextOut(...) has the following format:

```
BOOL CDC::ExtTextOut

(

int x, int y,

UINT nOptions, LPCRECT lpRect, const CString &str, LPINT lpDxWidths

);
```

Like CDC::TextOut(...), the first two parameters x and y specify the position of the output text. The third parameter nOptions indicates the drawing mode, it could be any type of combination between ETO_CLIPPED and ETO_OPAQUE flags (Either flag bit can be set or not set, altogether there are four possibilities). Style ETO_CLIPPED allows us to output text within a specified rectangle, and restrict the drawing within the rectangle even if the size of the text is bigger than the rectangle. In this case, all interior part of the rectangle not occupied by the text is treated as background. The third parameter is a CString type value that specifies the actual text we want to output. The last parameter is a pointer to an array of integers, which specify the distances between origins of two adjacent characters. This gives us the control of placing each character within a text string to a specified place. If we pass NULL to this parameter, the default spacing method will be applied.

A very typical use of this function is to implement a progress bar with percentage displayed in it (Figure 9-2). The progress bar is divided into two parts. For one part the text color is white and the background color is blue, for the other part the text color is blue and the background color is white.

New Class

Sample 9.3\GDI demonstrates how to implement this percentage bar. It is a standard SDI application generated from Application Wizard.

First a new class CPercent is added to the application through using Class Wizard, this class will be used to implement the percentage bar. Here, the base class is selected as CStatic.

The purpose of choosing CStatic as the base class is that by doing this, we can easily use subclass method to change a static control contained in dialog box to a percentage bar. Of course, we can choose other type of controls such as CButton to change a button to a percentage bar.

Two variables m_nRange and m_nCurPos along with two functions are added to class CPercent. Also, WM_PAINT message handler is added to the class through using Class Wizard, and the corresponding member funtion is OnPaint(). The following is this new class:

(Code omitted)

Variable m_nRange indicates the range of the percentage bar, and m_nCurPos indicates the current position. They are initialized in the constructor:

```
CPercent::CPercent()
```

```
{

m_nRange=100;

m_nCurPos=0;

}
```

Funtions CPercent::SetPercentage(...) and CPercent::SetPosition(...) allow us to change the value of m_nCurPos, and CPercent::SetRange(...) allows us to change the total range.

Within function CPercent::OnPaint(), we will draw the percentage bar using the values of m_nRange and m_nCurPos.

We need to check if m_nRange is zero. If so, we can not draw the percentage bar. If not, we need to first find out the size of the window (the static control window) within which the percentage bar will be drawn. This information is stored in a local variable rect. Next, we create the text string and store it in another local variable szStr, whose format is "XXX%" (XXX represents a number between 0 and 100). To place the text in the center of the rectangle, we need to know its dimension.

To retrieve the dimension of a text string, we need to call function CDC::GetTextExtent(...) and pass the actual string to it. The function will return a CSize type value that specifies the dimension of this text.

The percentage bar is divided into two portions. On the left side of the rectangle, the text color is white and the background color is blue. The following portion of function CPercent::OnPaint() shows how to create text string, set foreground and background colors, and retrieve the dimension of the text:

(Code omitted)

Next, the dimension of the left side rectange of the percentage bar is stored in local variable rectHalf. Then function CDC::ExtTextOut(...) is called to draw the left part of the percentage bar (Mode ETO_CLIPPED is used here, it will restrict the drawing within the rectangle). Because ETO_OPAQUE flag is also used, the text will be drawn with white color and the rest part of rectangle (specified by rectHalf) will all be painted blue:

(Code omitted)

Then we swap the text and background colors, store the right side rectangle in variable rectHalf, and call CDC::ExtTextOut(...) again to draw the rest part of the percentage bar:

(Code omitted)

The last two statements resume the original text color and background color for the device context.

Implementing Percentage Bar

It is very simple to use class CPercent to implement subclass for a static control contained in a dialog box. In the sample, a dialog template IDD_DIALOG is added to the application, it contains an "OK" button and a picture control whose ID is IDC_STATIC_PROG. The control has a modal frame whose color is set to "Gray", this will let the percentage bar have a 3-D effect. The above two styles can be set in the "Picture Properties" property sheet (See Figure 9-3 and Figure 9-4).

Class CProgDlg is added to the application for template IDD_DIALOG. In the class, a CPercent type variable m_perBar along with an integer type variable m_nPercent are declared:

```
class CProgDlg : public CDialog

{

......

protected:

CPercent m_perBar;

int m_nPercent;

......

}
```

Variable m_perBar will be used to implement percentange bar, and m_nPercent will be used to record the current position of the percentage bar.

Variable m_nPercent is initialized in the constructor:

(Code omitted)

In the sample, WM_INITDIALOG message handler is added to the application through using Class Wizard, and funtion CProgDlg::OnInitDialog() is implemented as follows:

(Code omitted)

Control IDC_STATIC_PROG is changed to a progress bar through implementing subclass, then a timer is started to generate events that will be handled to advance the percentage bar.

To handle time out events, in the sample, a WM_TIMER message handler is added throgh using Class Wizard. The corresponding member function CProgDlg::OnTimer(...) is implemented as follows:

(Code omitted)

If timer times out, we advance the percentage bar one step forward (1%); if the percentage bar reaches 100%, we reset it to 0%.

We must destroy timer when the application exits. The best place of doing this is when we receive WM_DESTROY message. This message handler can also be added through using Class Wizard. In the sample, the corresponding member function is implemented as follows:

```
void CProgDlg::OnDestroy()

{

CDialog::OnDestroy();

KillTimer(TIMER_ID);

}
```

For the purpose of testing the percentage bar, a new command Dialog | Progress is added to the application, whose command ID is ID_DIALOG_PROGRESS. A WM_COMMAND message handler is added to class CGDIDoc for this command, and the corresponding member function CGDIDoc::OnDialogProgress() is implemneted as follows:

```
void CGDIDoc::OnDialogProgress()

{

CProgDlg dlg;
dlg.DoModal();

}
```

After all these implementations, we can execute command Dialog | Progress to test the percentage bar.

9.4 One-Line Text Editor, Step 1: Displaying a Static String

From now on we are going to implement a very simple one-line text editor: it will display only one line text that can be edited through using mouse and keyboard. We will implement many features of a standard editor such as font selection, changing text styles. The sample application in this section does not introduce any new concept, it is the base of later sections.

Sample 9.4\GDI is a standard SDI application generated by Application Wizard. Because the horizontal size of the text string may be bigger than that of the client window as the user input more and more characters, we need to add scroll bars to the application. In order to do this, we can choose CScrollView as the base class of the client window in the final step of Application Wizard.

The sample does nothing but displaying a static text in the client window. Other features of text editor will be implemented in later sections.

Two variables m_szText and m_ftDraw along with two member functions are declared in class CGDIDoc:

(Code omitted)

Variable m_szText will be used to store the text string, and m_ftDraw will be used to store the font used for text drawing. Functions CGDIDoc::GetText() and CGDIDoc::GetFont() provide a way of accessing the two member variables outside class CGDIDoc.

Because we still do not have an interactive input environment, in the constructor, variable m_szText is initialized to a fixed string:

```
CGDIDoc::CGDIDoc()

{

m_szText="This is just a test string";

}
```

In the sample, function CGDIDoc::OnNewDocument() is modified as follows:

(Code omitted)

This function will be called when the document is initialized. Within the function, variable m_ftDraw is used to create a default font. Since the document is always created before the view, creating the font in this function will guarantee that m_ftDraw will be a valid font when the view is created. This procedure can also be done in the constructor.

To let the user select different types of fonts, a new command Dialog | Font is added to application's mainframe menu IDR_MAINFRAME. The resource ID of this command is ID_DIALOG_FONT and the corresponding message handler is CGDIDoc::OnDialogFont(), which is implemented as follows:

(Code omitted)

After a new font is selected by the user, we delete the old font and create a new one, then call function CDocument::UpdateAllViews(...) to update the client window.

On the view side, we need to modify function CGDIView::OnDraw(...). In this function, the text string and the font are retrieved from the document, and are used to draw text in the client window:

(Code omitted)

With the above implementation, the application will display a static string. Although we still can not input any character, the font for drawing the text can be changed through executing Dialog | Font command.

9.5 One Line Text Editor, Step 2: Adding Caret

Caret Functions

Caret is a very important feature for text editor, it indicates the current editing position. This makes the interface more user friendly. Because there are many types of fonts in the system, and for each font, the width of different characters may vary, we need to make careful calculation before moving the caret to the next position.

Every class derived from the CWnd supports caret, the steps of implementing caret are as follows: 1) Create a caret with specific style. 2) Show the caret. 3) Destroy the caret before the window is destroyed.

The following three member functions can be used to create a caret:

```
void CWnd::CreateSolidCaret(int nWidth, int nHeight);

void CWnd::CreateGrayCaret(int nWidth, int nHeight);

void CWnd::CreateCaret(CBitmap *pBitmap);
```

The first member function allows us to create a solid caret, here parameters nWidth and nHeight specify the dimension of the caret. Similarly, the second function can be used to create a gray caret. The last function can create a caret from a bitmap so that the caret can have a custom pattern.

After the caret is created, we can call function CWnd::ShowCaret() to display the caret or call function CWnd::HideCaret() to hide the caret.

The difficult thing on managing caret is to set its position. Because every character may have a different width, when the user presses arrow keys, we can not advance the caret with fixed distance each time. We must move the caret forward or backward according to the width of the character located before (or after) the caret. In order to do this, we can either calculate the new caret position each time, or we can store the starting position of each character in a table and obtain the caret position from it whenever the caret needs to be moved. In the sample, the latter solution is used.

Sample

Sample 9.5\GDI demonstrates how to implemnt caret. It is based on sample 9.4\GDI.

First, some new variables and functions are added to class CGDIDoc for caret implementation:

(Code omitted)

Two variables m_nCaretIndex and m_nCaretVerSize are added. The first variable is the index indicating the position of the caret. The second variable is the caret's vertical size. This is necessary because for fonts with different sizes, we need to create different carets, whose vertical size should be the same with the current font's height.

Five functions are added to the class, among them, function CGDIDoc::GetCaretVerSize() provides us a way of obtaining the current caret's vertical size in class CGDIView; and function CGDIDoc:: GetCaretPosition()

converts the caret index to a position within the client window (The function returns a POINT type value). It is implemented as follows:

(Code omitted)

The caret position is calculated through using function CDC::GetTextExtent(...), which will return the vertical and a horizontal size of a text string. We need a DC to select the current font in order to calculate the text dimension. In the sample, first a DC that does not belong to any window is created, then the current font is selected into this DC, and function CString::Left() is called to obtain a sub-string whose last character is located at the current caret position. The horizontal size obtained from function CDC:: GetTextExtent(...) for the sub-string is the caret's horizontal position. Because we have only one line text, the vertical position of the caret is always 0.

This function may be called within the member functions of CGDIView to retrieve the current caret position. The other two functions, CGDIDoc::ForwardCaret() and CGDIDoc::BackwardCaret() can be called to move the caret forward or backward. They are implemented as follows:

(Code omitted)

Instead of calculating the actual position of the caret, we just increment or decrement the caret index. The range of this index is from 0 to the total number of characters (If there are five characters, we have six possible positions for displaying the caret). If the index goes beyond the limit, we set it back to the boundary value.

At the end of above two functions, function CGDIDoc::GetCGDIView() is called to access class CGDIView, then CGDIView::RedrawCaret() is called to update the caret. This will cause the caret to be displayed in a new position. To access a view from the document, we need to call function CDocument:: GetFirstViewPosition() and then call CDocument::GetNextView(...) repeatedly until we get the correct view. For an SDI application, we need to call this function only once. However, some applications may have more than one view attached to the document (Like an MDI application). In this case, we need to use RUNTIME_CLASS macro to judge if the class is the one we are looking for. In the sample, CGDIDoc:: GetCGDIView() is implemented as a general function, it can also be used in an MDI application to obtain a specific view from the document. The following is its implementation:

(Code omitted)

We will implement function CGDIView::RedrawCaret() later.

In the sample, member variable m_nCaretIndex is initialized in the constructor along with m_szText:

```
CGDIDoc::CGDIDoc()

{

m_szText="This is just a test string";

m_nCaretIndex=0;

}
```

Also, when the document is first created or when a new font is selected, we need to update the value of m_nCaretVerSize, so functions CGDIDoc::OnNewDocument() and CGDIDoc::OnDialogFont() are updated as follows:

Old version of CGDIDoc::OnNewDocument():

(Code omitted)

New version of CGDIDoc::OnNewDocument():

(Code omitted)

Old version of CGDIDoc::OnDialogFont():

(Code omitted)

New version of CGDIDoc::OnDialogFont():

(Code omitted)

Function CDC::GetOutputTextMetrics(...) is called to obtain the information of the selected font. It will return a lot of information about the font such as its height, average width. This information is stored in a TEXTMETRIC type object, and the font's height can be retrieved from its tmHeight member.

In the above function, CGDIView::CreateNewCaret(...) is called to create a new caret. We will implement this function in the next step. As we will see, passing TRUE to this function will cause the old caret to be destroyed automatically.

In class CGDIView, two new functions are declared (They are called from

CGDIDoc::ForwardCaret(), CGDIDoc::BackwardCaret(), CGDIDoc::OnDialogFont()):

(Code omitted)

Function CGDIView::RedrawCaret() will erase the current caret and draw it at the new position. Function CGDIView::CreateNewCaret() will create a new caret and destroy the old one if necessary. The following code fragment shows their implementations:

(Code omitted)

In both functions, we retrieve the caret position from the document. Before moving the caret, we first call function CWnd::HideCaret() to hide the caret. After setting the new position, we call CWnd::ShowCaret() to show the caret again. Also, function CWnd::CreateSolidCaret(...) is called to create the caret, since we pass 0 to its horizontal dimension, the horizontal size of the caret will be set to the default size. The vertical size is retrieved from the document.

We need to create the caret once the view is created, so the default function CGDIView:: OnInitialUpdate() is modified as follows:

(Code omitted)

Here we just call function CGDIView::CreateNewCaret(...) and pass a FALSE value to its parameter because there is no caret needs to be destroyed.

Now we must respond to the events of left arrow and right arrow key strokes. As we know, when a key is pressed, the system will send WM_KEYDOWN message to the application, with the key code stored in WPARAM parameter. Under Windows(, all keys are defined as virtual keys, so there is no need for us to check the actual code sent from the keyboard. In order to know which key was pressed, we can examine WPARAM parameter after WM_KEYDOWN message is received. Here, the virtual key code of the left arrow key and right arrow key are VK_LEFT and VK_RIGHT respectively.

In the sample, WM_KEYDOWN message handler is added to class CGDIView through using Class Wizard, and the corresponding function CGDIView::OnKeyDown(...) is implemented as follows:

(Code omitted)

In this function, WPARAM parameter is mapped to nChar parameter. If the key stroke is from left arrow key, we call CGDIDoc::BackwardCaret() to move the caret leftward. If the key stroke is from right arrow key, we call

CGDIDoc::ForwardCaret() to move the caret rightward.

9.6 One Line Text Editor, Step 3: Enabling Input

Sample 9.6\GDI is based on sample 9.5\GDI, it allows the user to input characters.

New Member Functions

We need to trap keyboard stroking events in order to let the user input characters. Since our data is stored in the document, we need to first provide some member functions that can be called from the view to let the new characters be added. For this purpose, two new functions are declared in class CGDIDoc:

(Code omitted)

Function CGDIDoc::AddChar(...) allows us to insert characters to the string at the position indicated by the caret, and function CGDIDoc::DeleteChar(...) allows us to delete the character before or after the caret. Let's first take a look at the implementation of function CGDIDoc::AddChar(...):

(Code omitted)

We divide the text string into two parts, the first part is the sub-string before the caret, and the second part is the sub-string after the caret. The new characters are inserted between the two sub-strings. Parameter uChar indicates the new character, and uRepCnt specifies how many characters will be added. After the character is added, we update the view and move the caret forward.

For function CGDIDoc::DeleteChar(...), it can be used for two situations: one corresponds to "BACK SPACE" key stroke, the other corresponds to "DELETE" key stroke. If parameter bBefore is true, the character before the current caret should be deleted. Otherwise, the character after it needs to be deleted. The following is the implementation of function CGDIDoc::DeleteChar(...):

(Code omitted)

To delete the character before the current caret, we divide the text into two sub-strings, delete the last character of the first sub-string, and re-combine them. Then we update the view, and move caret one character left. When deleting the character after the caret, we do not need to change the position of the caret.

Message WM_CHAR

Now we need to use the above two member functions. In the sample, message WM_CHAR is handled to implement keyboard input. The difference between WM_CHAR and WM_KEYDOWN messages is that WM_CHAR is sent only for printable characters along with the following five keys: ESCAPE, TAB, BACK SPACE and ENTER. Message WM_KEYDOWN will be sent for all types of key strokes.

In the sample, the message handler of WM_CHAR is CGDIView::OnChar(...), it is implemented as follows in the sample:

(Code omitted)

We neglect the ENTER, TAB and ESCAPE key strokes. For BACK SPACE key stroke, we delete the character before the current caret. For all other cases, we insert character at the current caret position.

The DELETE key stroke can not be detected by this message handler, we need to trap and handle it in function CGDIView::OnKeyDown(...):

(Code omitted)

Of course the printable key strokes will also be detected by this message handler. However, if we handle character input in this function, we need to first check if the character is printable. This will make the program a little bit complex.

9.7 One Line Text Editor, Step 4: Caret Moving & Cursor Shape

Sample 9.7\GDI is based on sample 9.6\GDI.

New Functions

Besides moving the caret before or after one character at a time, we sometimes need to move the caret to the next or the previous word. This will give the user a faster way of putting the caret at the appropriate position. The method of moving caret to the next or previous word is almost the same with moving it to the next or previous character, the only difference between them is how to calculate the new caret position. Because words are separated by blanks, if we want to move caret one word leftward or rightward, we can just find the previous or next blank, calculate the distance, then move the caret to the new position.

Also, we may want to let the user use HOME key and END key to move the caret to the beginning or the end of the text. Still, almost every text editor supports changing the caret position with a single mouse clicking.

The following new functions are declared in class CGDIDoc to implement above-mentioned functionalities:

(Code omitted)

As implied by the function names, CGDIDoc::HomeCaret() will move the caret to the beginning of the text, CGDIDoc::EndCaret() will move the caret to the end of the text. The implementation of these two functions is very simple, all we need to do is setting m_nCaretIndex to a proper value then updating the caret:

(Code omitted)

For other two functions CGDIDoc::ForwardCaretToBlank() and CGDIDoc::BackwardCaretToBlank(), we need to find out the position of the nearest blank, and set the value of m_nCaretIndex to it:

(Code omitted)

Within the two functions, a local variable szSub is used to store the sub-string before or after the caret, and function CString::Find(...) or CString::ReverseFind(...) is called to find the position of the nearest blank. In case the blank is not found, we need to move the caret to the beginning or end of the text. If it is found, we just increment or decrement m_nCaretIndex by an appropriate value.

On the view side, we need to move the caret when any of the following keys is pressed together with CTRL: HOME, END, Left and Right ARROW. These keystroke events can be trapped by handling WM_KEYDOWN message. To detect if the CTRL key is held down, we can call API function ::GetKeyState(...) to check the key state.

Function ::GetKeyState(...) can be used to check the current state of any key. We need to pass the virtual key code (such as VK_CONTROL, VK_SHIFT...) to this function when making the call. The returned value is a SHORT type integer. The high order of this value indicates if the key is held down, the lower order indicates if the key is toggled, which is applicable to keys such as CAPS LOCK, NUM LOCK or INSERT.

Moving Caret Using Keyboard

Function CGDIView::OnKeyDown(...) is modified as follows to add the new features to the application:

(Code omitted)

Changes are made to VK_LEFT and VK_RIGHT cases. First we call

::GetKeyState(...) using virtual key code VK_CONTROL and extract the high order byte from the return value. If it is non-zero, function CGDIDoc:: BackwardCaretToBlank() or CGDIDoc::ForwardCaretToBlank() is called to move the caret to the nearest blank. Other wise we move the caret one character leftward or rightward.

In case the key is VK_END or VK_HOME, we call function CGDIDoc::EndCaret() or CGDIDoc ::HomeCaret() to move the caret to the beginning or end of the text.

Moving Caret Using Mouse

Moving the caret by mouse clicking is a little bit complex. Because all we can obtain from the message parameter is the mouse's current position, we need to convert it to a caret index before moving the caret. To do so, we need to go over all the starting positions of characters, calculate the absolute distance between each character and the current mouse cursor. The index that results in the smallest distance will be used as the new caret index. In the sample, function CGDIDoc::SetCaret(...) is declared for this purpose, which converts geometrical coordinates to caret index and move the caret to the new place. The following is its implementation:

(Code omitted)

In order to find out all the possible caret positions, we need to obtain the dimension of different sub- strings. For example, the caret positions of text "abcde" can be calculated from the dimensions of following sub-strings: "a", "ab", "abc", "abcd", "abcde". In the above function, first a DC that does not belong to any window is created, then the current font is selected into this DC, and function CDC::GetTextExtent(...) is called to obtain the dimension of each sub-string. Because we have only one line text, only the horizontal size is meaningful to us.

A loop is implemented to do the comparison. For the nth loop, we create a sub-string that contains text's first character to nth character, obtain its dimension, and calculate the distance from the position of the last character of the sub-string to the current position of mouse cursor. After the loop finishes, we choose the smallest distance and set m_nCaretIndex to the corresponding caret index.

Cursor Shape

For a text editor, when the mouse is over its editable area, usually the mouse cursor should be changed to an insertion cursor. This indicates that the user can input text at this time. This feature can also be implemented in our sample application.

To set mouse cursor's shape, we need to call API function ::SetCursor(...). The input parameter to this function is an HCURSOR type handle.

A cursor can be prepared as a resource and then be loaded before being used. After the cursor is loaded, we can pass its handle to function ::SetCursor(...) to change the current cursor shape. Besides the cursor prepared by the user, there also exist some standard cursors that can be loaded directly.

We can call function CWinApp::LoadCursor(...) to load a user-defined cursor, and call function CWinApp::LoadStandardCursor(...) to load a standard cursor. The following table lists some of the standard cursors:

(Table omitted)

In the sample, an HCURSOR type variable is declared in class CGDIView, and the insertion cursor is loaded in function CGDIView::OnInitialUpdate():

```
class CGDIView : public CScrollView

{

protected:

HCURSOR m_hCur;

……

}

void CGDIView::OnInitialUpdate()

{

……

m_hCur=AfxGetApp()->LoadStandardCursor(IDC_IBEAM);

}
```

We need to respond to WM_SETCURSOR message in order to change the cursor shape. By handling this message, we have a chance to customize the default cursor when it is within the client window of the application. Upon receiving this message, we can check if the mouse position is over the editable text. If so, we need to change the cursor by calling API function ::SetCursor(...). In this case, we need to return a TRUE value and should not call the default message handler. If the cursor should not be changed, we need to call the default message handler to let the cursor be set as usual.

To check out if the cursor is over editable text, we need to know the text dimension all the time. In the previous steps, we already have a variable CGDIDoc::m_nCaretVerSize that is used to store the vertical size of the caret (also the text), so here we just need another variable to store the horizontal size of the text. In the sample, a new variable n_nTextHorSize is declared in class CGDIDoc for this purpose:

(Code omitted)

Besides the new variable, function CGDIDoc::GetTextHorSize() is also added, which lets us access the value of CGDIDoc::m_nTextHorSize outside class CGDIDoc.

We need to set the value of m_nTextHorSize when the document is first initialized and when the font size is changed (In the following two functions, m_nTextHorSize is assigned a new value):

(Code omitted)

Message handler of WM_SETCURSOR can be added through using Class Wizard. The corresponding function CGDIView::OnSetCursor(...) is implemented as follows:

(Code omitted)

We call ::GetCursorPos(...) and CWnd::ScreenToClient(...) to retrieve the current cursor position in the client window's coordinate system. Then functions CGDIDoc::GetTextHorSize() and CGDIDoc:: GetCaretVerSize() are called to retrieve the dimension of the text. If the mouse cursor is within this rectangle, we call ::SetCursor(...) to change it to insertion cursor. In this case, we must return a TRUE value to avoid this message from being further processed (by default the mouse cursor will be set to arrow cursor).

Handling WM_LBUTTONDOWN to Move Caret

The caret can be moved when the current mouse cursor is an insertion cursor. To implement this, we need to call function CGDIDoc::SetCaret(...) after receiving WM_LBUTTONDOWN message. In the sample, this message handler is added through using Class Wizard, and the corresponding function CGDIView:: OnLButtonDown(...) is implemented as follows:

(Code omitted)

In this function, first we check if the mouse cursor is the insertion cursor. If not, it means that the mouse is not over the text string. If so, we call function CGDIDoc::SetCaret(...) and pass current mouse position to it. This will cause the caret to move to the new position.

9.8 One Line Text Editor, Step 5: Selection

Sample 9.8\GDI is based on sample 9.7\GDI.

Highlighting the Selected Text

The next feature we will add is to let the user select text using mouse. If this is implemented, it is easy for us to add cut, copy and paste functionalities.

To add the selection feature, we need two new text indices: one indicates the beginning of the selection, one indicates the end of the selection. In the sample, two variables along with two functions are added to class CGDIView for this purpose:

(Code omitted)

Here, variables m_nSelIndexBgn, m_nSelIndexEnd, functions CGDIDoc::GetSelIndexBgn() and CGDIDoc::GetSelIndexEnd() are added to the class. Before the selection is made, there may exist two situations: the text is currently being selected or there is no character being selected. To distiguish between the two situations, let's define that if any of m_nSelIndexBgn and m_nselIndexEnd is -1, or their values are the same, it indicates that there is no text being selected. The two variables are initialized in the constructor as follows:

(Code omitted)

In CGDIView::OnDraw(...), we need to retrieve the values of m_nSelIndexBgn and m_nSelIndexEnd, swap forground and background colors for the selected text when outputting text string to the client window. The following is the modified function CGDIView::OnDraw(...):

(Code omitted)

Two local variables nSelIndexBgn and nSelIndexEnd are declared here, they are used to store the values of CGDIDoc::m_nSelIndexBgn and CGDIDoc::m_nSelIndexEnd retrieved from the document. If the value of CGDIDoc::m_nSelIndexEnd is less than the value of CGDIDoc::m_nSelIndexBgn (In this case, the selection is made from right to left), we need to swap their values.

If there is no currently selected text, we simply call CDC::TextOut(...) as usual to output the plain text. Otherwise we swap the two indices if m_nSelIndexEnd is less than m_nSelIndexBgn, and set the text alignment by calling function CDC::SetTextAlign(...) using TA_UPDATECP flag. This will cause the output origin to be updated to the end of the text after each CDC::TextOut(...) call. If we do not set this alignment, the coordinates passed to function CDC::TextOut(...) indicate a position relative to the upper-left corner of the window. With TA_UPDATECP alignment style, when we call function CDC::TextOut(...), the coordinates passed to this function will be interpreted as a position relative to the new origin (which is the end of the text that is output by function CDC::TextOut(...) last time). This is very useful if we want to output several segments of strings. In the sample, the old alignment flag is stored in variable uTextAlign, and is restored after the text is output.

We divide the text string into three segments: the first segment starts from the beginning of the text and ends at the beginning of the selection. We output this sub-string using normal text and background colors. The second segment is the selected portion, before drawing this sub-string we need to swap the text and background colors so that the selected part will be drawn highlighted. The rest part is the third sub-string, which is also drawn using the normal text and background colors. Each time function CDC::TextOut(...) is called, the output coordinates are specified as (0, 0). If the alignment flag is not set to TA_UPDATECP, the three segments will all be drawn starting from the same origin.

Setting Selection Indices

Now we can change the values of CGDIDoc::m_nSelIndexBgn and CGDIDoc::m_nSelIndexEnd to highlight any portion of the text string. From user's point of view, this should happen when the mouse is clicked and dragged over the text. In order to implement this, we need to respond to WM_LBUTTONDWON, WM_LBUTTONUP and WM_MOUSEMOVE messages.

When left button is pressed down, we need to first reset the values of CGDIDoc::m_nSelIndexBgn and CGDIDoc::m_nSelIndexEnd to -1, because we need to unselect any currently highlighted text. Then we can update the value of CGDIDoc::m_nSelIndexBgn to the current caret index. When mouse moves with the left button held down, we need to set the value of CGDIDoc::m_nSelIndexEnd to the current caret index (the caret will move with the mouse cursor). The same thing needs to be done when mouse's left button is released. For these purposes, a new function ResetSelection() is declared in class CGDIDoc, which will be called to reset the selection indices. Also, function CGDIDoc::SetCaret(...) is modified. The following is a portion of the updated class CGDIDoc:

(Code omitted)

Function CGDIDoc::ResetSelection() is implemented inline, it just resets the values of m_nSelIndexBgn and m_nSelIndexEnd to -1, then updates the view window. The following shows the modified portion of function CGDIDoc::SetCaret(...):

(Code omitted)

We want to use this function to set both m_nSelIndexBgn and m_nSelIndexEnd, so that it can be called in response to any of the three mouse messages. If the value of m_nSelIndexBgn is -1, it means there is no currently selected text. In this case, we need to update m_nSelIndexBgn (This is the situation that the left button of the mouse is pressed down). In other cases (If m_nSelIndexBgn is not -1, the function must be called in response to mouse moving or left button up event), we need to update the value of m_nSelIndexEnd, then update the client window.

Handling Mouse Events

First function CGDIView::OnLButtonDown(...) is modified as follows:

Old Version:

(Code omitted)

New Version:

(Code omitted)

The only thing added to this function is resetting the selection indices stored in the document. Other changes are implemented in the updated function CGDIView::SetCaret(...).

Two other message handlers for WM_LBUTTONUP and WM_MOUSEMOVE are added through using Class Wizard. The corresponding functions are CGDIView::OnLButtonUp(...) and CGDIView::OnMouseMove(...) respectively.

Function CGDIView::OnMouseMove(...) is implemented as follows:

(Code omitted)

We find out if the left button is held down when the mouse is moving by checking MK_LBUTTON bit of parameter nFlags. If so, function CGDIDoc::SetCaret(...) is called to set the selection and update the client window. Function CGDIView::OnLButtonUp(...) is implemented exactly the same except that we don't need to check the status of left button here:

(Code omitted)

With the above knowledge, it is very easy for us to implement selection by using left/right ARROW key when SHIFT key is held down. To implement this, we need to check SHIFT key's status when either left or right ARROW key is pressed. If it is held, we can update the selection indices and redraw the client window.

9.9 One Line Text Editor, Step 6: Cut, Copy and Paste

Global Memory

Cut, copy and paste are supported almost by every application. They provide a way to exchange data among different applications. We must use globally shared data to implement clipboard data transfer. Once we send some data to the clipboard, it becomes public and can be accessed by all the programs. Any process in the system can clear the clipboard.

Because of this, we must allocate global memory to store our data in the clipboard. In Windows(programming, the following API functions can be used to manage global memory:

```
HGLOBAL ::GlobalAlloc(UINT uFlags, DWORD dwBytes);

LPVOID ::GlobalLock(HGLOBAL hMem);

HGLOBAL ::GlobalReAlloc(HGLOBAL hMem, DWORD dwBytes, UINT uFlags);

BOOL ::GlobalUnlock(HGLOBAL hMem);

HGLOBAL ::GlobalFree(HGLOBAL hMem);
```

Global memory is also managed through using handle. Unlike memory allocated using new key word, function ::GlobalAlloc(...) returns an HGLOBAL type handle to the allocated memory block if we allocate non-fixed global memory.

Generally, before accessing a non-fixed block of global memory, we must lock it by calling function ::GlobalLock(...), which will return the address of the memory block. After reading from or writing to this memory block, we need to call function ::GlobalUnlock(...) to unlock the memory again. We can free a block of global memory by calling function ::GlobalFree(...) (We can not free a block of memory when it is being locked).

We can also allocate fixed global memory, in which case the address of the memory will be returned by function ::GlobalAlloc(...) directly, and we do not need lock or unlock operation in order to access the memory.

Parameter nFlags of function ::GlobalAlloc(...) specifies how the memory will be allocated. There are many possible choices. For example, we can make the memory movable or fixed, and fill all the buffers with zero. The following is a list of some commonly used flags:

(Table omitted)

The most commonly used flag is GHND, which specifies that the memory block should be movable and all buffers should be initialized to zeros. For the clipboard usage, we also need to specify flag GMEM_SHARE, which will enhance the performance of clipboard operation.

Actually, in Win32 programming, the memory block allocated by one process can not be shared by other processes. Flag GMEM_SHARE exists just for the backward compatibility purpose. For a general application, we can not allocate a block of memory using flag GMEM_SHARE and share it with other applications. In Win32, this flag is solely used for clipboard and DDE (see chapter 15) implementation.

The memory size that will be allocated is specified by dwBytes parameter of function ::GlobalAlloc(...).

Apart from these functions, there is another set of functions whose functionality is exactly the same, the only difference is that they have a different set of function names:

HLOCAL ::LocalAlloc(UINT uFlags, UINT uBytes);

LPVOID ::LocalLock(HLOCAL hMem);

HLOCAL ::LocalReAlloc(HLOCAL hMem, UINT uBytes, UINT uFlags);

BOOL ::LocalUnlock(HLOCAL hMem);

HLOCAL ::LocalFree(HLOCAL hMem);

Everything is exactly the same except that all the "Global" keywords are changed to "Local" here. These functions are originated from the old Win16 programming, which uses 16-bit memory mode. In that case the memory can be allocated either from the local heap or global heap. In Win32 programming, there is only one heap, so two sets of functions become exactly the same. They exist just for the compatibility purpose. We can use either of them in our program. We can even call ::GlobalAlloc(...) to allocate memory and release it using

::LocalFree(...).

Clipboard Funcitons

To copy our own data to the clipboard, we need to first prepare the data. The following lists the necessary steps for allocating memory blcok and fill it with our data: 1) Allocate enough buffers by calling function ::GlobalAlloc(...). 2) Lock the memory by calling function ::GlobalLock(...), which will return a pointer that can be used to access the memory buffers. 3) Fill these buffers with data. 4) Call ::GlobalUnlock(...) to unlock the memory.

We need to use a series of functions in order to put the data to the clipboard: 1) First we need to call function CWnd::OpenClipboard(...), which will let the clipboard be owned by our application (Only the window that owns the clipboard can modify the data contained in the clipboard, any other application is forbidden to access the clipboard during this period). 2) Before putting any data to the clipboard, we must call ::EmptyClipboard() to clear any existing data. 3) We can call ::SetClipboardData() to put new data to the clipboard. 4) Finally we need to call ::CloseClipboard() to close the clipboard, this will let the clipboard be accessible to other windows.

When calling function ::SetClipboardData(...), besides passing the handle of the global memory, we also need to specify the data format. There are many standard clipboard data formats such as CF_TEXT, CF_DIB, which represent text data and bitmap data respectively. We can also define our own data format by calling function ::RegisterClipboardFormat(...).

To copy data from the clipboard, we need to open the clipboard first, then call function ::GetClipboardData(), which will return a global memory handle. With this handle, we can call ::GlobalLock(...) to lock the memory, copy the data from the global memory to our own buffers, call ::GlobalUnlock(...) to unlock the memory, and close the clipboard. We can not free the global memory obtained from the clipboard because after the clipboard is closed, some other applications may also want to access it.

Deleting Selected Text

Sample 9.9\GDI is based on sample 9.8\GDI, it allows the user to cut or copy the selected text to the clipboard, and paste the data from clipboard.

When we cut data to the clipboard, we also need to delete the selected text. So first a new function DeleteSelection() is declared in class CGDIDoc, it can be called to delete the currently selected text:

```
class CGDIDoc : public CDocument
```

```
{
......
public:
......
BOOL DeleteSelection();
......
}
```

Function CGDIDoc::DeleteSelection() is implemented as follows:

(Code omitted)

When there is no currently selected text, the function does nothing. Otherwise we proceed to delete the selected text.

Because the ending selection index may be less than the beginning selection index, first we set the value of local variable nSel to the smaller selection index, and set the number of selected characters to another local variable nNum. Then the unselected text is combined together, and the caret index is adjusted. Next the caret and the client window are updated. Finally, both selection indices are set to -1, this indicates that currently there is no text being selected.

We can call this function when DELETE key is pressed to delete the selected text, also we can call it when the selected text is being cut to the clipboard. In the sample, function CGDIDoc::DeleteChar() is modified as follows:

(Code omitted)

Since this member function may be called when either BACK SPACE or DELETE key is pressed, we need to delete the selected text in both cases. If deleting the selected text is successful, the function will return. Otherwise it means there is no currently selected text, so we go on to delete a single character.

Message Handlers for Cut, Copy Paste Commands

In the sample, both WM_COMMAND and UPDATE_COMMAND_UI message hanlders are added for command ID_EDIT_CUT, ID_EDIT_COPY and ID_EDIT_PASTE in class CGDIDoc. We need to enable commands Edit | Copy and

Edit | Cut if there is selected text. So functions CGDIDoc::OnUpdateEditCopy(...) and CGDIDoc::OnUpdateEditCut(...) are implemented as follows:

(Code omitted)

Two functions are implemented exactly the same. For function CGDIDoc::OnUpdateEditPaste(...), we need to check if there is data available in the clipboard, if so, the command will be enabled. This checking can be implemented by calling function ::IsClipboardFormatAvailable(...) with appropriate data format passed to it. The function will return FALSE if there is no data present in the clipboard for the specified data format. The following is the implementation of funcition CGDIDoc::OnUpdateEditPaste(...):

(Code omitted)

Command Edit | Copy is implemented as follows:

(Code omitted)

First, we assign the smaller of the two selection indicies to variable nSel, and the number of selected characters to variable nNum. Then we copy the selected text to a CString type variable szStr. Next, we allocate a memory block, lock it, copy the string from szStr to the new buffers. Then we unlock the memory, open the clipboard, clear it, and copy the data to the clipboard. Finally we close the clipboard.

The implmentation of Edit | Cut command is almost the same except that we must delete the selected text after copying the data to the clipboard. So function CGDIDoc::OnEditCut() is implemented as follows:

(Code omitted)

For Edit | Paste command, everything is the reverse. We need to open the clipboard, obtain data from the clipboard, lock the global memory, copy the data to local buffers, unlock the global memory, and insert the new string to the text at the current caret position. The following is the implementation of this command:

(Code omitted)

Now the application can exchange data with another application that supports clipboard.

9.10 One Line Text Editor, Step 7: Getting Rid of Flickering

The editor is almost finished except for one annoying feature: every time the user inputs a character, makes selection or moves the caret, the text will flicker. This is because whenever the text is being updated, we call function CDocument::OnUpdateAllViews(...) to cause the whose view window to be updated. By default, before the client window is redrawn, it will be erased using white color. This is the cause of flickering. To get rid of it, we need to update only the area that has changed (instead of updating the whole window).

Function CDocument::UpdateAllViews(...)

Function CDocument::UpdateAllViews(...) has three parameters, two of which have default values:

```
void CDocument::UpdateAllViews(CView *pSender, LPARAM lHint=0L, CObject *pHint=NULL);
```

By default, the update message will be sent to view, this will cause function CView::OnUpdate(...) to be called:

```
void CView::OnUpdate(CView *pSender, LPARAM lHint, CObject *pHint);
```

All parameters passed to CDocument::UpdateAllViews(...) will be passed to this function. This provides us a way to know what part of the client window needs to be updated. The updating hint can be passed through either parameter lHint or pHint.

By default, CView::OnUpdate(...) will update the whole client area. If we want only a portion of the client window to be updated, we need to bypass the default implementation. Within the overridden funciton, we can use the hint to form a rectangle indicating the area needs to be updated, and use it to call function CWnd::InvalidateRect(...).

Function CWnd::InvalidateRect(...) will cause only the specified rectangular area to be updated.

Defining Hints

Our next task is to divide the updating events into different categories and calculate the rectangle for each situation. The following is a list of situations when only a portion of the client window needs to be updated:

(Table omitted)

The last two situations are a little complicated. When the user makes selections, the newly selected area may be smaller or larger than the old selected area. In either case, we only need to update the changed area to avoid flickering (Figure 9-5).

Because of this, we need to add new variables to remember the old selection indices. In the sample, two new variables and some functions are declared in class CGDIDoc as follows:

(Code omitted)

Variables m_nSelIndexBgnOld and m_nSelIndexEndOld are used to remember the old selection indices, functions GetSelIndexBgnOld() and GetSelIndexEndOld() are used to obtain their values outside class CGDIDoc. Because we also need to know the value of m_nCaretIndex when updating the client window, another function GetCaretIndex() is also added for retrieving its value.

The value of m_nSelIndexEndOld is initialized in the constructor:

```
CGDIDoc::CGDIDoc()

{

......

m_nSelIndexEndOld=-1;

}
```

In the sample, some macros are defined as follows to indicate different updating situations when function CDocument::UpdateAllViews(...) is called:

```
#define HINT_DELCHAR_AFTER 100

#define HINT_DELCHAR_BEFORE 101

#define HINT_DELETE_SELECTION 102

#define HINT_PASTE 103

#define HINT_SELECTION 104

#define HINT_UNSELECTION 105

#define HINT_INPUT 106
```

Calling Function CDocument::UpdateAllViews(...)

We must modify all the function calls to CDocument::UpdateAllViews(...). The following shows the modifications made to function CGDIDoc::SetCaret(...):

Old Version:

(Code omitted)

New Version:

(Code omitted)

In this function, the value of m_nSelIndexEnd is first assigned to m_nSelIndexEndOld before it is updated. Flag HINT_SELECTION will cause the difference between the newly selected area and the old one to be updated. The area can be calculated from the four selection indices.

The following shows the modifications made to function CGDIDoc::AddChar(...):

Old Version:

(Code omitted)

New Version:

(Code omitted)

Flag HINT_INPUT will cause all the characters after the caret to be updated.

The following shows the modifications made to funciton CGDIDoc::DeleteChar(...):

Old Version:

(Code omitted)

New Version:

(Code omitted)

Flag HINT_DELCHAR_AFTER will cause all the characters after the caret to be updated, and HINT_DELCHAR_BEFORE will cause the character before the caret along with all the characters after the caret to be updated.

The following shows the modifications made to function CGDIDoc::DeleteSelection():

Old Version:

(Code omitted)

New Version:

(Code omitted)

Flag HINT_DELETE_SELECTON will cause the selected text and the characters after the selection to be updated.

The following shows the modifications made to function CGDIDoc::OnEditPaste():

Old Version:

(Code omitted)

New Version:

(Code omitted)

Flag HINT_PASTE will cause all the characters after the caret to be updated.

The following shows the modifications made to inline function CGDIDoc::ResetSelection():

Old Version:

(Code omitted)

New Version:

(Code omitted)

Flag HINT_UNSELECTION will cause only the selected area to be updated. Because both m_nSelIndexBgn and m_nSelIndexEnd should be set to -1 to indicate that there is no selected text anymore, we need to use two other variables (m_nSelIndexBgnOld and m_nSelIndexEndOld) to store the old selection indices.

Overriding CView::OnUpdate(...)

On the view side, function OnUpdate(...) can be added through using Class Wizard. In this function, we need to know the current values of CGDIDoc::m_nSelIndexBgn, CGDIDoc::m_nSelIndexEnd, CGDIDoc:: m_nSelIndexBgnOld, CGDIDoc::m_nSelIndexEndOld and CGDIDoc::m_nCaretIndex in order to decide which part of the text should be updated. We also need to obtain the current font and text string in order to calculate the actual rectangle for implementing update.

If parameter lHint is NULL, it means that all client area needs to be updated. In this case, we call the default implementation of this function and return. The following is a portion of funciton CGDIView:: OnUpdate(...) which implements this:

(Code omitted)

If parameter lHint is non-null, we need to obtain the current font, text string and selection indices from the document, and calculate the area that should be updated. Here variable rect stores a rectangle that covers all of the text (within the window).

In the case when hint is one of HINT_DELCHAR_AFTER, HINT_PASTE and HINT_INPUT, we need to update all the characters after the caret:

(Code omitted)

The caret index is retrieved from the document and stored to variable nIndex. Then the sub-string before the caret is extracted and stored to variable szText. Its dimension is calculated by calling function CDC::GetTextExtent(...), and the left border of rect is changed so that it covers only the characters after the caret. Finally, function CWnd::InvalidateRect(...) is called and rect is passed to one of its parameters.

If the hint is HINT_DELCHAR_BEFORE, we need to update the character before the caret and all the characters after the caret:

(Code omitted)

If the hint is HINT_DELETE_SELECTION, we need to update the selected text as well as the characters after the selection:

(Code omitted)

If the hint is HINT_UNSELECTION, we need to update only the selected text.

Since both CGDIDoc:: m_nSelIndexBgn and CGDIDoc::m_nSelIndexEnd are -1 now, we must use CGDIDoc::m_nSelIndexBgnOld and CGDIDoc::m_nSelIndexEndOld to calculate the rectangle:

(Code omitted)

If the hint is HINT_SELECTION, we must check if CGDIDoc::nSelIndexEndOld is -1 or not. If it is -1, it means that the area needed to be updated is between CGDIDoc::nSelIndexBgn and CGDIDoc::nSelIndexEnd; if not, the area needed to be updated is between CGDIDoc::nSelIndexEnd and CGDIDoc::nSelIndexEndOld (See Figure 9-6):

(Code omitted)

If we pass FALSE to the second parameter of CWnd::InvalidateRect(...), the client area will be updated without being erased. This can further reduce flickering.

Summary:

1) Font can be created from structure LOGFONT. We need to provide the following information when creating a special font: face name, font size (height and width). To add special effects to the text, we need to know if the font is bolded, italic, underlined or strikeout. Also, we can change character's orientation by setting font's escapement.

2) All the fonts contained in the system can be enumerated by calling function ::EnumFontFamilies(...). We need to provide a callback function to receive information for each type of font.

3) Function CDC::ExtTextOut(...) can output a text string within a specified rectangle, the area outside the rectangle will not be affected no matter what the text size is. When we call this function, all the area not covered by the text within the rectangle is treated as the background of the text.

4) To implement caret within a window, first we need to create the caret by using one of the following functions: CWnd::CreateSolidCaret(...), CWnd::CreateGrayCaret(...), CWnd::CreateCaret(...). Then we can show or hide the caret by calling either function CWnd::ShowCaret() or CWnd::HideCaret().

5) Keyboard input events can be trapped by handling WM_KEYDOWN or WM_CHAR message.

6) Mouse cursor can be changed by handling message WM_SETCURSOR. We can load a user designed cursor resource by calling function

CWinApp::LoadCursor(...). We can also load a standard cursor by calling function CWinApp::LoadStandardCursor(...).

7) If we call function CDC::SetTextAlign(...) using TA_UPDATECP flag, the window origin will be updated to the end of the text each time funciton CDC::TextOut(...) is called.

8) To use global memory, we need to call ::GlobalAlloc(...) to allocate the buffers, call ::GlobalLock(...) to lock the memory before accessing it, call ::GlobalUnlock(...) to stop accessing the memory, and call ::GlobalFree(...) to release the memory.

9) To access the clipboard, we need to call CWnd::OpenClipboard(...) to open the clipboard, call ::EmptyClipboard() to clear any existing data, call ::SetClipboardData() to put data to the clipboard, and call ::CloseClipboard() to close the clipboard. To get data from the clipboard, after opening it, we need to call function ::GetClipboardData() to obtain a global memory handle, which can be used for accessing the data contained in the clipboard.

10) We can pass hints to function CDocument::UpdateAllViews(...) to indicate different updating situations. The hint can be received in function CView::OnUpdate(...). If we want only a portion of the client window to be updated, we can specify the area with a CRect type variable and use it to call function CWnd::InvalidateRect(...) instead of default function CWnd::Invalidate(...).

BACK TO INDEX

Chapter 10 Bitmap

From this chapter we are going to deal with another GDI object ¾ bitmap, which is a very complex issue in Windows(programming. There are many topics on how to use bitmaps, how to avoid color distortion, how obtain palette from bitmap, how convert from one bitmap format to another, and how to manipulate image pixels.

Samples in this chapter are specially designed to work on 256-color palette device. To customize them for non-palette devices, we can just eleminate logical palette creation and realization procedure.

10.1 BitBlt and StretchBlt

DIB & DDB

There are two type of bitmaps under Windows(: device independent bitmap (DIB) and device dependent bitmap (DDB). As we know, each computer may be equipped with a different type of device, therefore may use different format to store bitmap images in the hardware. Before displaying any image on the screen, we need to convert the image data to a format that is understandable by the hardware device, otherwise the image could not be displayed correctly. This format is called device dependent bitmap format (DDB), because it is hardware dependent. However, using DDB format will cause incompatibility between different systems, because one DDB format may not be understandable by another device. To solve this problem, under Windows(, a device independent bitmap format (DIB format) is supported by all device drivers. When we call the standard function to load a device independent bitmap, the device driver will convert it to DDB that is understandable by the device.

Drawing DDB

There are several ways to include bitmap image in an application. The simplest one is to treat it as bitmap resource. To load the image, we can call function CBitmap::LoadBitmap(...) and pass the bitmap resource ID to it.

After the bitmap is loaded, we need to output it to the target device (such as screen). The procedure of outputting a bitmap to a target device is different from using a pen or brush to draw a line or fill a rectangle: we cannot select bitmap into the target DC and draw the bitmap directly. Instead, we must create a compatible memory DC and select the bitmap into it, then copy the bitmap from memory DC to the target DC.

The functions that can be used to copy a bitmap between two DCs are CDC::BitBlt(...) and CDC::StretchBlt(...). The former function allows us to copy the bitmap in 1:1 ratio, and the latter one allows us to enlarge or reduce the dimension of the original image. Lets first take a look at the first member function:

```
BOOL CDC::BitBlt

(

int x, int y, int nWidth, int nHeight,

CDC *pSrcDC,

int xSrc, int ySrc,

DWORD dwRop

);
```

There are eight parameters, first four of them specify the origin and size of the target bitmap that will be drawn. Here x and y can be any position in the target device, also, nWidth and nHeight can be less than the dimension of source image (In this case, only a portion of the source image will be drawn). The fifth parameter is a pointer to the source DC. The seventh and eighth parameters specify the origin of the source bitmap. Here, we can select any position in the source bitmap as origin. The last parameter specifes the bitmap drawing mode. We can draw a bitmap using many modes, for example, we can copy the original bitmap to the target, turn the output black or white, do bit-wise OR, AND or XOR operation between source bitmap and target bitmap.

Creating Memory DC

A memory DC used for copying bitmap image must be compatible with the target DC. We can call function CDC::CreateCompatibleDC(...) to create this type of DC. The following is the format of this function:

```
BOOL CDC::CreateCompatibleDC(CDC* pDC);
```

The only parameter to this function (pDC) must be a pointer to the target DC.

Retrieving the Dimension of Bitmap Image

We see that in order to copy the bitmap from one DC to another, we need to know the dimension of the bitmap. The bitmap size, along with other information, can be retrieved by calling function CBitmap::GetBitmap(...). If we pass a BITMAP type pointer to this function, the object will be filled with the bitmap information. The bitmap dimension is stored in members bmWidth and bmHeight of structure BITMAP.

Sample 10.1\GDI

Sample 10.1-1\GDI demonstrates how to use function CDC::BitBlt(...). It is a standard SDI application generated by Application Wizard, and its view is based on class CScrollView. In the sample, first a bitmap resource is added to the application, whose ID is IDB_BITMAP.

A CBitmap type variable is declared in class CGDIDoc, it will be used to load this bitmap:

(Code omitted)

Variable m_bmpDraw will be used to load the bitmap, and function GetBitmap() will be used to access it outside class CGDIDoc. Bitmap IDB_BITMAP is loaded in the constructor of class CGDIDoc:

(Code omitted)

In function CGDIView::OnInitialUpdate(), we need to use bitmap dimension to set the total window scroll sizes so that if the window is not big enough, we can scroll the image to see the covered portion:

(Code omitted)

The bitmap pointer is obtained from the document. By calling function CBitmap::GetBitmap(...), all the bitmap information (including its dimension) is obtained and stored in variable bm. Then the scroll sizes are set using bitmap dimension. By doing this, the scroll bars will pop up automatically if the dimension of the client window becomes smaller than the dimension of the bitmap.

Function CGDIView::OnDraw(...) is implemented as follows:

(Code omitted)

First we call CDC::CreateCompatibleDC(...) to create a compatible memory DC, then use it to select the bitmap obtained from the document. Like other GDI objects, after using a bitmap, we need to select it out of the DC. For this purpose, a local variable pBmpOld is used to store the returned address when we call CDC::SelectObject(...) to select the bitmap (pBmp) into memory DC. After the bitmap is drawn, we call this function again to select pBmpOld, this will select bitmap stored by pBmp out of the DC.

In the next step function CBitmap::GetBitmap(...) is called to retrieve all the bitmap information into variable bm, whose bmHeight and bmWidth members (represent the dimension of bitmap) will be used for copying the bitmap. Then we call CDC::BitBlt(...) to copy the bitmap from the memory DC to the target DC. The origin of the target bitmap is specified at (0, 0), also, the source bitmap and target bitmap have the same size.

Sample 10.1-2\GDI

Sample 10.1-2\GDI demonstrates how to use function CDC::StretchBlt(...) to output the bitmap image. It is based on sample 10.1-1\GDI. In this sample, the image is enlarged to twice of its original size and output to the client window.

Because the target image has a bigger dimension now, we need to adjust the scroll sizes. First function CGDIView::OnInitialUpdate() is modified as follows for this purpose:

(Code omitted)

The scroll sizes are set to twice of the bitmap size.

Function CDC::StretchBlt(...) has 10 parameters:

(Code omitted)

There are two extra parameters nSrcWidth and nSrcHeight here (compared to function CDC::BitBlt()), which specify the extent of original bitmap that will be output to the target. Obviously, nWidth and nSrcWidth determine the horizontal ratio of the output bitmap (relative to source bitmap). Likewise, nHeight and nSrcHeight determine the vertical ratio.

In the sample, both horizontal and vertical ratios are set to 200%, and function CGDIView::OnDraw(...) is modified as follows:

(Code omitted)

With the above modifications, we will have an enlarged bitmap image in the client window.

10.2 Extracting Palette from DIB

If we execute the previous two samples on a palette device, we may experience color distortion (Please open file 10.1-1\bitmap.bmp using a standard graphic editor and compare the results). This is because we didn't implement logic palette for displaying the bitmap, so the the colors of the bitmap pixels are mapped to the nearest colors available in the system. To avoid color distortion, we must implement logical palette before drawing the bitmap.

Another problem is that when we call function CBitmap::LoadBitmap(...) to load the bitmap resource, it will create a device dependent bitmap from the data stored in the resource. So after the the bitmap is loaded, it becomes DDB, which could only be understood by the device. To extract palette information from the bitmap data, we must use DIB format.

To avoid color distortion, we must implement logical palette for device dependent bitmap before it is drawn to the target device. Because function CBitmap::LoadBitmap(...) does not extract palette and the bitmap data stored in the resource is in the format of DIB, we must convert the bitmap to DDB and extract palette information from it by ourselves.

DIB Format

A DIB comprises three parts: bitmap information header, color table, and bitmap bit values. The bitmap information header stores the information about the bitmap such as its width, height, bit count per pixel, etc. The color table contains an array of RGB colors, it can be referenced by the color indices. The bitmap bit values represent bitmap pattern by specifying an index into the color table for every pixel. The color table can also be empty, in which case the bitmap bit vlues must be actual R, G, B combinations.

There are several type of DIBs: monocrome, 16 colors, 256 colors and 24 bit. The first three formats use color table and color indices to form a bitmap. The last format does not have a color table and all the pixels are represented by R, G, B combinations.

The following is the format of bitmap information header:

```
typedef struct tagBITMAPINFOHEADER{
```

```
DWORD biSize;

LONG biWidth;

LONG biHeight;

WORD biPlanes;

WORD biBitCount

DWORD biCompression;

DWORD biSizeImage;

LONG biXPelsPerMeter;

LONG biYPelsPerMeter;

DWORD biClrUsed;

DWORD biClrImportant;

} BITMAPINFOHEADER;
```

It has 11 members, the most important ones are biSize, biWidth, biHeight, biBitCount and biSizeImage.

Member biSize specifies the length of this structure, which can be specified by sizeof(BITMAPINFORHEADER). Members biWidth and biHeight specify the dimension of the bitmap image. Member biBitCount specifies how many bits are used to represent one pixel (Bit count per pixel). This factor determines the total number of colors that can be used by the bitmap and also, the size of the color table. For example, if this member is 1, the bitmap can use only two colors. In this case, the size of color table is 2. If it is 4, the bitmap can use up to 16 colors and the size of the color table is 16. The possible values of this member are 1, 4, 8, 16, 24, and 32. Member biSizeImage specifies the total number of bytes that must be allocated for storing image data. This is a very important member, because we must know its value before allocating memory.

An image is composed of multiple raster lines, each raster line is made up of an array of pixels. To speed up image loading, each raster line must use a multiple of four-byte buffers (This means if we have a 2-color (monochrom) 1(1 bitmap, we need four bytes instead of one byte to store only one pixel). Because of this, the value of biSizeImage can not be simply calculated by the following fomulae:

biHeight*biWidth*biBitCount/8

We will discuss how to calculate this value later.

For bitmaps under Windows(, member biPlanes is always set to 1. If the bitmap is not compressed, member biCompress should be set to BI_RGB.

The next part of the DIB data is color table, which comprises an array of RGBQUAD objects. The bitmap information header and color table together form a bitmap header, which can be described by a BITMAPINFO structure:

```
typedef struct tagBITMAPINFO {

BITMAPINFOHEADER bmiHeader;

RGBQUAD bmiColors[1];

} BITMAPINFO;
```

DIB Example

Following the bitmap header are bitmap bit values. Image data is stored in the memory one pixel after another from left to right, and vertically from the bottom to the top. The following is an example of 3(4 image:

(Table omitted)

This image has only two colors: black and white. If we store it using monochrome DIB format (2 colors), we need only 3 bits to store one raster line. For 16-color DIB format, we need 12 bits to store one line. Since each raster line must use multiple of four-byte buffers (32 bits), if one raster line can not use up all the bits, the rest will simply be left unused.

The following table compares four different types of DIB formats by listing the following information: the necessary bits needed, the actual bits used, and number of bits wasted by one raster line:

(Table omitted)

We can define a macro that allows us to calculate the number of bytes needed for each raster line for different bitmap formats:

```
#define WIDTHBYTES(bits) ((((bits)+31)/32)*4)
```

Here bits represents the number of bits that are needed for one raster line, it can be obtained from BITMAPINFORHEADER by doing the following calculation:

biWidth(biBitCount

Now we know how to calculate the value of biSizeImage from other members of structure BITMAPINFOHEADER:

WIDTHBYTES(biWidth*biBitCount)*biHeight

The following is the image data for the above DIB example, assume all unused bits are set to 0:

2 color bitmap (assuming in the color table, index to white color = 0, index to black color = 1):

```
C0 00 00 00

40 00 00 00

C0 00 00 00

40 00 00 00
```

4 color bitmap (assuming in the color table, index to white color = 0, index to black color = 15):

```
F0 00 00 00

0F 00 00 00

F0 F0 00 00

0F 00 00 00
```

8 color bitmap (assuming in the color table, index to white color = 0, index to black color = 255):

```
FF 00 FF 00

00 FF 00 00

FF 00 FF 00
```

```
00 FF 00 00
```

24 bit color bitmap:

```
00 00 00 FF FF FF 00 00 00 00 00

FF FF FF 00 00 00 FF FF FF 00 00

00 00 00 FF FF FF 00 00 00 00 00

FF FF FF 00 00 00 FF FF FF 00 00
```

The bitmap resource is stored exactly in the format mentioned above. To avoid color distortion, we need to extract the color table contained in the DIB to create logic palette, and convert DIB to DDB before drawing it.

Creating DDB from DIB

To create a DDB from DIB data, we need to call function ::CreateDIBitmap(...), which has the following format:

```
HBITMAP ::CreateDIBitmap

(

HDC hdc,

CONST BITMAPINFOHEADER *lpbmih, DWORD fdwInit,

CONST VOID *lpbInit, CONST BITMAPINFO *lpbmi, UINT fuUsage

);
```

The first parameter of this function is a handle to target DC. Because DDB is device dependent, we must know the DC information in order to create the bitmap. The second parameter is a pointer to BITMAPINFORHEADER type object, it contains bitmap information. The third parameter is a flag, if we set it to CBM_INIT, the bitmap will be initialized with the data pointed by lpbInit and lpbmi; if this flag is 0, a blank bitmap will be created. The final parameter specifies how to use color table. If the color table is contained in the bitmap header, we can set its value to DIB_RGB_COLORS.

Loading Resource

To access data stored in the resource, we need to call the following three funcitons:

```
HRSRC ::FindResource(HMODULE hModule, LPCTSTR lpName, LPCTSTR lpType);

HGLOBAL ::LoadResource(HMODULE hModule, HRSRC hResInfo);

LPVOID ::LockResource(HGLOBAL hResData);
```

The first function will find the specified resource and return a resource handle. When calling this function, we need to provide module handle (which can be obtained by calling function AfxGetResourceHandle()), the resource ID, and the resource type. The second function loads the resource found by the first function. When calling this function, we need to provide the module handle, along with the resource handle returned from the first function. The third function locks the resource. By doing this, we can access the data contained in it. The input parameter to this function must be the global handle obtained from the second function.

Sample

Sample 10.2\GDI demonstrates how to extract color table from DIB and convert it to DDB. It is based on sample 10.1-2\GDI.

Because we need to implement logical palette, first a variable and a function are added to class CGDIDoc:

(Code omitted)

Variable m_palDraw is added for creating logical palette and function GetPalette() is used to access it outside class CGDIDoc.

In class CGDIView, some new variables are added for bitmap drawing:

(Code omitted)

Variable m_dcMem will be used to implement memory DC at the initialization stage of the client window. It will be used later for drawing bitmap. By implementing memory DC this way, we don't have to create it every time. Also, we will select the bitmap and palette into the memory DC after they are avialabe, and selet them out of the DC when the window is being destroyed. For this purpose, two pointers m_pPalMemOld and m_pBmpMemOld are declared, they will be used to select the palette and bitmap out of DC. Variable m_bmInfo is used to store the information of bitmap.

The best place to create bitmap and palette is in CGDIView::OnInitialUpdate(). First, we must locate and load the bitmap resource:

(Code omitted)

We call funcitons ::FindResource(...) and ::LoadResource(...) to load the bitmap resource. Function ::FindResource(...) will return a handle to the specified resource block, which can be passed to ::LoadResource(...) for loading the resource. This function returns a global memory handle. We can call ::LockResource(...) to lock the resource. This function will return an address that can be used to access the bitmap data. In the sample, we use pointer lpBi to store this address.

Next, we must calculate the size of color table and allocate enough memory for creating logical palette. The color table size can be calculated from "bit count per pixel" information of the bitmap as follows:

(Code omitted)

The color table size is stored in variable nSizeCT. Next, the logical palette is created from the color table stored in the DIB data:

(Code omitted)

The color table is obtained from member bmiColors of structure BITMAPINFO (pointed by lpBi). Since the palette is stored in the document, we first call CGDIDoc::GetPalette() to obtain the address of the palette (CGDIDoc::m_palDraw), then call CPalette::CreatePalette(...) to create the palette. After this we select the palette into the client DC, and call CDC::RealizePalette() to let the logical palette be mapped to the system palette.

Then, we create the DDB from DIB data:

(Code omitted)

Function ::CreateDIBitmap(...) returns an HBITMAP type handle, which must be associated with a CBitmap type varible by calling function CBitmap::Attach(...).

The rest part of this funciton fills variable CGDIView::m_bmInfo with bitmap information, sets the scroll sizes, create the memory DC, select bitmap and palette into it, then free the bitmap resource loaded before:

(Code omitted)

Because the bitmap and the palette are selected into the memory DC here, we must select them out before application exits. The best place to do this is in WM_DESTROY message handler. In the sample, a WM_DESTROY message handler is added to class CGDIView through using Class Wizard, and the corresponding function CGDIView::OnDestroy() is implemented as follows:

(Code omitted)

We must modify function CGDIView::OnDraw(...). Since everything is prepared at the initialization stage (the memory DC, the palette, the bitmap), the only thing we need to do in this function is calling CDC:: StrethBlt(...) or CDC::BitBlt(...) to copy the bitmap from memory DC to target DC:

(Code omitted)

With the above implementations, the bitmap will become more vivid.

10.3 Loading DIB from File

Now that we understand how to convert a DIB to DDB, it is fairly easy for us to load a device independent bitmap into memory and convert it to DDB. Sample 10.3\GDI demonstrates how to read DIB file and display the image in the clinet window. It is based on sample 10.2\GDI.

File Format

All bitmap images stored on the hard disk are in the format of DIB, and therefore can be any of the following formats: monochrome, 16 color, 256 color and 24-bit. The difference between DIB stored in a file and DIB stored as a resource is that there is an extra bitmap file header for DIB stored in file. This header has the following format:

```
typedef struct tagBITMAPFILEHEADER {

WORD bfType;

DWORD bfSize;

WORD bfReserved1;

WORD bfReserved2;

DWORD bfOffBits;
```

} BITMAPFILEHEADER;

This header specifies three factors: the file type, the bitmap file size, and the offset specifying where the DIB data really starts. So a real DIB file has the following format:

BITMAPFILEHEADER

BITMAPINFOHEADER

Color Table

Bitmap Bits

Member bfType must be set to 'BM', which indicates that the file is a bitmap file. Member bfSize specifies the whole length of the bitmap file, which is counted from the beginning of the file to its end. Member bfOffBits specifies the offset from BITMAPFILEHEADER to bitmap bits, which should be the size of BITMAPINFOHEADER plus the size of color table.

Supporting File Type

Using SDI or MDI model, it is easy to support certain type of files in "Open" or "Save As" common dialog box (activated when the user executes File | Open or File | Save As command). This feature can be added by changing IDR_MAINFRAME string resource. By default, this string resource contains the following sub-strings (Sub-strings are separated by '\n'):

GDI\n\nGDI\n\n\nGDI.Document\nGDI Document

The meaning of each sub string is listed in the following table:

(Table omitted)

By default, no special file types are supported. If we want the file dialog box to contain certain filters, we need to change the fourth and fifth items of the above string. In the sample, string resource IDR_MAFRAME is changed to the following:

GDI\n\nGDI\nBitmap Files(*.bmp)\n.bmp\nGDI.Document\nGDI Document

The fourth item (Bitmap Files(*.bmp)) is a descriptive name for bitmap files, and the fifth item (.bmp) is the filter.

Loading DIB through Serialization

Since we are going to store data read from the file in globally allocated memory, first we must add an HGLOBAL type varible (along with a member function) to class CGDIDoc. Because DIB data will be changed to DDB data before drawing, a CBitmap type variable and its associated funciton are also declared in class CGDIDoc:

(Code omitted)

Variable m_hDIB is initialized in the constructor:

```
CGDIDoc::CGDIDoc()

{

m_hDIB=NULL;

}
```

We will allocate global memory each time a new bitmap file is opened. So when the application exits, we need to check variable m_hDIB to see if the memory has been allocated. If so, we need to release it:

(Code omitted)

We need to modify function CGDIDoc::Serialize(...) to load data from DIB file. First, when a file is being opened, variable m_hDIB may be currently in use. In this case, we need to release the global memory:

(Code omitted)

Reading data from a DIB file needs the following steps:

1) Read bitmap file header.

2) Verify that the the file format is correct.

3) From the bitmap file header, calculate the total memory needed, then allocate enough buffers.

4) Read the DIB data into the allocated buffers.

We can call CArchive::Read(...) to read bytes from the file into the memory. In the sample, first bitmap file header (data contained in structure BITMAPFILEHEADER) is read:

(Code omitted)

Then the file type is checked. If it is DIB format, global memory is allocated, which will be used to store DIB data:

(Code omitted)

Next, the DIB data is read into the global memory:

(Code omitted)

This completes reading the bitmap file. The DIB data is stored in m_hDIB now.

Creating DDB

On the view side, we need to display the bitmap stored in memory (whose handle is m_hDIB) instead of the bitmap stored as resource. So we need to modify function CGDIView::OnInitialUpdate(), which will be called whenever a new file is opened successfully. First, instead of obtaining data from the resource, function CGDIDoc::GetHDib() is called to get the handle of DIB data:

(Code omitted)

Remember in the previous sample, after the DDB and palette are created, we will select them into the memory DC so that the image can be drawn directly later. In this sample, when a new DIB file is opened, there may be a bitmap (also a palette) that is being currently selected by the memory DC. If so, we need to select it out of the memory DC, then lock the global memory and obtain its address that can be used to access the DIB (In order to create DDB, we need the information contained in the DIB):

(Code omitted)

The rest portion of this function calculates the size of the color table, creates the palette (If there is an existing palette, delete it first), and creates DDB from DIB data. Then the newly created bitmap and palette are selected into the memory DC, and m_bmInfo is updated with the new bitmap information. Finally, the global memory is unlocked.

Other functions remain unchanged. With the above implementation, we can open any DIB file with our application and display the image in the client window.

10.4 Saving DDB to File

We cannot save DDB data directly to a file. Saving bitmap to a file is a reverse procedure of loading it from the disk: we must first convert the DDB back to DIB then write the data to file.

Converting DDB to DIB

To convert a DDB to DIB, we need to call API function ::GetDIBits(...) to receive the bitmap data in DIB format. When calling this function, we must allocate enough buffers for receiving data. As we know, DIB data contains three parts: header BITMAPINFOHEADER, whose size is fixed; color table, whose size depends on the data format; bitmap bit values, whose size depends upon both the bitmap format and bitmap dimension.

We need to calculate the size of color table and bitmap bit values before allocating memory for storing DIB. The attributes of a DDB can be obtained from function CBitmap::GetBitmap(...). The information will be stuffed into a BITMAP type object, from which we know the properties of a bitmap such as its height and width, bit count per pixel. The size of the color table can be calculated from bit count per pixel information.

Lets further take a look at function ::GetDIBits(...):

(Code omitted)

Its parameters are similar to that of funciton ::CreateDIBitmap(). Since DDB is device dependent, we must know the attribute of device context in order to convert DDB to DIB. So we must pass the handle of the target DC to the first parameter of this function. Parameter uStartScan and uScanLines specify the starting raster line and total number of raster lines whose data is to be retrieved. Parameter lpBits specifies the buffer address that can be used to recieve bitmap data. Pointer lpbi provides a BITMAPINFO structure specifying the desired format of DIB data.

When calling this function, we can pass NULL to pointer lpvBits. This will cause the function to fill the bitmap information into a BITMAPINFO object. By doing this, we can get the color table that is being used by the DDB.

So the conversion takes three steps: 1) Call function CBitmap::GetBitmap(...) to obtain the information of the bitmap, calculate the color table size, allocate enough buffers for storing bitmap information header and color table. 2) Call function ::GetDIBits(...) and pass NULL to parameter lpvBits to receive bitmap information header and color table. 3) Reallocate buffers for storing bitmap data and call ::GetDIBits(...) again to get the DIB data.

New Functions

Sample 10.4\GDI demonstrates how to convert DDB to DIB and save the data to hard disk.

First some functions are added to class CGDIDoc, they will be used for converting DDB to DIB:

(Code omitted)

Function CGDIDoc::ConvertDDBtoDIB(...) converts DDB to DIB, its input parameter is a CBitmap type pointer and its return value is a global memory handle. Function CGDIDoc::GetColorTableSize(...) is used to calculate the size of color table from bit count per pixel information (In the previouse samples, color table size calculation is implemented within function CGDIView::OnInitialUpdate(). Since we need color table size information more frequently now, this calculation is implemented as a single member function):

(Code omitted)

In function CGDIDoc::ConvertDDBtoDIB(...), first we must obtain a handle to the client window that can be used to create a DC:

(Code omitted)

Then function CBitmap::GetBitmap(...) is called to retrieve the information of bitmap and allocate enough buffers for storing structure BITMAPINFOHEADER and color table:

(Code omitted)

We first fill the information obtained previously into a BITMAPINFOHEADER object. This is necessary because when calling function ::GetDIBits(...), we need to provide a BITMAPINFOHEADER type pointer which contains useful information. Here, some unimportant members of BITMAPINFOHEADER are assigned 0s (biSizeImage, biXPelsPerMeter...). Then the size of the color table is calculated and a global memory that is big enough for holding bitmap information header and color table is allocated, and the bitmap information header is stored into the buffers. We will use these buffers to receive color table.

Although the memory size for storing bitmap data can be calculated from the information already known, usually it is not done at this point. Generally the color table and the bitmap data are retrieved separately, in the first step, only the memory that is big enough for storing structure BITMAPINFOHEADER and the color table is prepared. When color table is being retrieved, the bitmap information header will also be updated at the same time. Since it is more desirable to calculate the bitmap data size using the updated information, in the sample, the memory size is updated after the color table is obtained successfully, and the global memory is reallocated for retrieving the bitmap data.

We also need to select logical palette into the DC and realize it so that the bitmap pixels will be intepreted by its own color table.

Function ::GetDIBits(...) is called in the next step to recieve BITMAPINFOHEADER data and the color table. Because some device drivers do not fill member biImageSize (This member carries redunant information with members biWidth, biHeight, and biBitCount), we need to calculate it if necessary:

(Code omitted)

Now the size of DIB data is already known, we can reallocate the buffers, and call function ::GetDIBits(...) again to receive bitmap data. Finally we need to select the logical palette out of the DC, and return the handle of the global memory before function exits:

(Code omitted)

Using New Functions

Using function CGDIDoc::ConvertDDBtoDIB(...), it is farily easy to save the bitmap into a file. All we need is to call this function to convert DDB to DIB, add a BITMAPFILEHEADER structure to it, and write the whole data into a file. In the sample, file saving is implement in function CGDIDoc::Serialize(...):

(Code omitted)

In the sample, command File | Save is disabled so that the user can only save the image through File | Save As command by specifying a new file name. To implement this, UPDATE_COMMAND_UI message handlers are added for both ID_FILE_SAVE and ID_FILE_SAVE_AS commands, and the corresponding member functions are implemented as follows:

(Code omitted)

It seems unnecessary to conver the DDB to DIB before saving the image to a disk file because its original format is DIB. However, if the DDB is changed after being loaded (This is possible for a graphic editor application), the new DDB is inconsistent with the original DIB data.

The DDB to DIB converting procedure is a little complex. If we are programming for Windows 95 or Windows NT 4.0, we can create DIB section (will be introduced in later sections) to let the format be converted automatically. If we are writing Win32 programs that will be run on Windows 3.1, we must use the method discussed in this section to implement the conversion.

10.5 Drawing DIB Directly

The counterpart function of ::GetDIBits(...) is ::SetDIBits(...), it can be used to convert DIB data to device dependent bitmap. The two functions provide us a way of implementing image editing: whenever we want to make change to the image, we can first retrieve DIB data from the DDB, edit the DIB data, and set it back to DDB.

New Functions

Sometimes it is easier to edit DDB directly instead of using DIB data. For example, if we want to reverse every pixel of the image, we can just call one API funciton to let this be handled by lower level driver instead of editting every single pixel by ourselves. This is why we need to handle both DIB and DDB in the applications.

If our application is restricted on edittng only DIB data, we can call an API function directly to draw DIB in the client window. By doing so, we eleminte the complexity of converting DIB to DDB back and forth. This function is ::SetDIBitsToDevice(...), which has the following format:

(Code omitted)

There are altogether 12 parameters, whose meanings are listed in the following table:

(Table omitted)

This function can output image at 1:1 ratio with respect to the source image. Similar to CDC::BitBlt(...) and CDC::StretchBlt(...), there is another function ::StretchDIBits(...), which allows us to enlarge or reduce the original image and output it to the target device:

(Code omitted)

The ratio between source and target image can be set through the following four parameters: nDestWidth, nDestHeight, nSrcWidth, nSrcHeight.

Modifications Made to Document

Sample 10.5\GDI is based on sample 10.4\GDI, it demonstrates how to use the above two functions.

With the new functions, many implementations in the previouse sample can be eleminated. First, on the document side, we no longer need CBitmap type variable (CGDDoc::m_bmpDraw) any more. We will use CGDIDoc::m_hDIB to store DIB data, which can be used directly to draw the image. Also we do not need function CGDIDoc::ConvertDDBtoDIB(...). Function CGDIDoc::GetColorTableSize(...) is still needed because we can use it to calculate the size of BITMAPINFO structure (BITMAPINFOHEADER and color table). In the sample, variable CGDIDoc::m_bmpDraw and function CGDIDoc::ConvertDDBtoDIB() are removed. The following is the modified class:

(Code omitted)

When saving image to file, we do not need to convert DDB to DIB any more. Instead, CGDIDoc::m_hDIB can be used directly for storing data:

(Code omitted)

Note since biSizeImage member of BITMAPINFOHEADER structure may be zero, we need to calculate its value before saving the image to file. Also, the original statement for releasing global memory is deleted because CGDIDoc::m_hDIB is the only variable that is used for storing image in the application.

Function CGDIDoc::OnUpdateFileSaveAs(...) is changed to the following:

```
void CGDIDoc::OnUpdateFileSaveAs(CCmdUI* pCmdUI)

{

pCmdUI->Enable(m_hDIB != NULL);

}
```

Modifications Made to View

On the view side, we do not need memory DC any more, so three variables CGDIView::m_dcMem, CGDIView::m_pBmpMemOld and CGDIView::m_pPalMemOld are deleted. Since the image size can be obtained from DIB data, variable CGDIView::m_bmInfo can also be eleminated. The following code fragment shows the modified class CGDIView:

(Code omitted)

In function CGDIView::OnInitialUpdate(), there is no need to create DDB any more. So in the updated function, only the logical palette is created:

(Code omitted)

In this functon, DIB handle is obtained from the document and locked. From the global memory buffers, the color table contained in the DIB is obtained and is used for creating the logical palette. The a flag is set to indicate that the bitmap is loaded successfully.

In function CGDIView::OnDraw(...), the DIB is painted to the client window:

(Code omitted)

The procedure of selecting and realizing the logical palette is the same with the previous sample. The difference between them is that function CDC::BitBlt(...) is replaced by function ::SetDIBitsToDevice(...) here.

Message handler CGDI::OnDestroy() is removed through using Class Wizard in the sample. The reason for this is that we no longer need to select objects (palette, bitmap) out of memory DC any more. Also, the constructor of CGDIView is changed as follows:

(Code omitted)

With the above modification, the application is able to display any DIB image without doing DIB to DDB conversion.

10.6 Bitmap Format Conversion: 256-color to 24-bit

Now that we understand different DIB formats, we can easily implement conversion from one format to another. Sample 10.6\GDI demonstrates how to convert 256-color DIB format to 24-bit format, it is based on sample 10.5\GDI.

Conversion

We need to delete the color table and expand the indices to explicit RGB combinations in order to implement this conversoin. Also in the bitmap information header, we need to change the value of member biBitCount to 24, and recalculate member biImageSize. There is also another difference in the bitmap header bwteen 256-color and 24-bit formats: for DIB that does not contain the color table, member biClrUsed is 0; for DIB that contains the color table, this member specifies the number of color indices in the color table that are actually used by the bitmap.

Current Format

In the sample, a new command Convert | 256 to RGB is added to the mainframe menu IDR_MAINFRAME, whose command ID is ID_CONVERT_256TORGB. Also, WM_COMMAND and UPDATE_COMMAND_UI message handlers are added for this command through using Class Wizard. The corresponding functions are CGDIDoc::OnConvert256toRGB() and CGDIDoc::OnUpdateConvert256toRGB(...) respectively. This command will be used to convert the image from 256-color format to 24-bit format. We want to disable this menu item if the current DIB is not 256 color format.

Before doing the conversion, we must know the current format of the image. So in the sample, a new variable is declared in class CGDIDoc for this purpose:

```
class CGDIDoc : public CDocument

{

protected:

……

int m_nBmpFormat;

……

}
```

The following macros are defined in the header file of class CGDIDoc:

```
#define BMP_FORMAT_NONE 0

#define BMP_FORMAT_MONO 1

#define BMP_FORMAT_16COLOR 2

#define BMP_FORMAT_256COLOR 3

#define BMP_FORMAT_24BIT 4
```

Variable CGDIDoc::m_nBmpFormat is initialized in the constructor:

```
CGDIDoc::CGDIDoc()

{

……

m_nBmpFormat=BMP_FORMAT_NONE;

}
```

Whenever a DIB is loaded, function CGDIDoc::GetColorTableSize(...) will be called by CGDIView:: OnInitialUpdate(), so it is convenient to set CGDIDoc::m_nBmpFormat to an appropriate value to indicate the image format in this function. The following code fragment shows the modified function CGDIDoc::GetColorTableSize():

(Code omitted)

Function Implementation

Function CGDIDoc::OnUpdateConvert256toRGB(...) is implemented as follows so that the menu command will be enabled only when the current DIB format is 256-color:

```
void CGDIDoc::OnUpdateConvert256toRGB(CCmdUI* pCmdUI)

{

pCmdUI->Enable(m_nBmpFormat == BMP_FORMAT_256COLOR);

}
```

In function CGDIDoc::OnConvert256toRGB(), first we need to lock the current DIB data, calculate the size of new DIB data (after format conversion) and allocate enough buffers:

(Code omitted)

The new DIB size is stored in local variable dwSize. Here macro WIDTHBYTES is used to calculate the actual bytes needed for one raster line (We use 24 instead of member biBitCount when using this macro to implement calculation for the new format). The size of new DIB data is the size of BITMAPINFOHEADER structure plus the size of bitmap data (Equal to bytes needed for one raster line multiplied by the height of bitmap, there is no color table any more). Then we allocate buffers from global memory and lock them, whose address is stored in pointer lpBi24.

Then we need to fill structure BITMAPINFOHEADER. Most of the members are the same for two different formats, such as biHeight, biWidth. There are three members we need to change: biBitCount must be set to 24, biImageSize should be recalculated, and biClrUsed needs to be 0:

(Code omitted)

Then we need to fill the DIB bit values. The image is converted pixel by pixel using two loops (one is embedded within another): the outer loop converts one raster line, and the inner loop converts one single pixel. As we move to a new raster line, we need to calculate the starting buffer address so that it can be used as the origin of the pixels for the whole raster line (For each single pixel, we can obtain its address by adding an offset to the origin address). The starting address of each raster line can be calculated through multiplying the current line index (0 based) by total number of bytes needed for one raster line. As we move from one pixel to the next of the same raster line, we can just move to the neighboring buffer (for RGB format, next three buffers). However, the final pixel of one raster line and the first pixel of next raster line may not use neighboring buffers, this is because there may exist some unused bits between them (Since each raster line must use a multiple of 4-byte buffers). The following portion of function CGDIDoc::OnConvert256toRGB() shows how to convert bitmap pixels from one format to another:

(Code omitted)

Finally, we must unlock the global memory, release the previous DIB data and assign the new memory handle to CGDIDoc::m_hDIB. We also need to inform the view to reload the image because the bitmap format has changed. For this purpose, a new function CGDIView::LoadBitmap(...) is implemented, it will be called from CGDIDoc::OnConvert256toRGB() and CGDIView::OnInitialUpdate() (The original portion of this funciton that loads the bitmap is replaced by calling the new function). The following is the portion of funciton CGDIDoc::OnConvert256toRGB() which shows what should be done after the format is converted:

(Code omitted)

Function CGDIView::LoadBitmap(...) should implement the following: delete the old palette, check if the DIB contains color table. If so, create a new palette. The function is declared in class CGDIView as follows:

(Code omitted)

The function is implemented as follows (Most part of this function is copied from function CGDIView::OnInitialUpdate()):

(Code omitted)

With this function, CGDIView::OnInitialUpdate() can be simplified to the following:

(Code omitted)

With the above implementation, the application is able to convert a bitmap from 256-color format format to 24-bit format.

10.7 Converting 24-bit Format to 256-color Format

Sample 5.7\GDI is based on sample 5.6\GDI, it demonstrates how to convert 24-bit bitmap format to 256-color format.

Two Cases

To convert a 24-bit format bitmap to 256-color format bitmap, we must extract a color table from the explicit RGB values. There are two cases that must be handled differently: the bitmap uses less than 256 colors, and the bitmap uses more than 256 colors.

If the bitmap uses less than 256 colors, the conversion is relatively simple: we just examine every pixel of the bitmap, and extract a color table from all the colors contained in the bitmap.

The following is the conversion procedure for this situation: At the beginning, the color table contains no color. Then for each pixel in the bitmap, we examine if the color is contained in the color table. If so, we move to the next pixel. If not, we add the color used by this pixel to the color table. After we go over all the pixels contained in the bitmap, the color table should contain all the colors that are used by the bitmap image.

If the bitmap uses more than 256 colors, we must find 256 colors that best represent all the colors used by the image. There are many algorithms for doing this, a relatively simple one is to omit some lower bits of RGB values so that maximum number of colors used by a bitmap does not exceed 256. For example, 24-bit bitmap format uses 8 bit to represent a basic color, it can result in 256(256(256 different colors. If we use only 3 bits to represent red and green color, and use 2 bits to represent blue color, the total number of possible combinations are 8(8(4=256.

In this situation, when we examine a pixel, we use the 3 most significant bits of red and green colors, along with 2 most significant bits of blue color to form a new color that will be used to create color table (Other bits will be filled with 0s). By doing this, the colors contained in the color table will not exceed 256. Although this algorithm may result in color distortion, it is relatively fast and less image dependent.

Sample

In the sample, a new command Convert | RGB to 256 is added to mainframe menu IDR_MAINFRAME, whose command ID is ID_CONVERT_RGBTO256. Also, WM_COMMAND and UPDATE_COMMAND_UI message handlers are added through using Class Wizard. The new corresponding functions are CGDIDoc:: OnConvertRGBto256() and CGDIDoc::OnUpdateConvertRGBto256(...) respectively.

Function CGDIDoc::OnUpdateConvertRGBto256(...) is implemented as follows:

```
void CGDIDoc::OnUpdateConvertRGBto256(CCmdUI* pCmdUI)

{

pCmdUI->Enable(m_nBmpFormat == BMP_FORMAT_24BIT);

}
```

If the current bitmap format is 24-bit, command Convert | RGB to 256 will be enabled.

The implementation of function CGDIDoc::OnConvertRGBto256() is somehow similar to that of CGDIDoc::OnConvert256toRGB(): we must first lock the global memory where the current 24-bit bitmap image is stored, then calculate the size of new bitmap image (256-color format), allocate enough buffers from global memory, and fill the new bitmap bit values one by one.

The first thing we need to do is creating the color table for the new bitmap image. The following portion of function CGDIDoc::OnConvertRGBto256() shows how to extract the color table from explicit RGB colors contained in a 24-bit bitmap image:

(Code omitted)

We examine from the first pixel. The color table will be stored in array arRgbQuad, which is empty at the beginning. For each pixel, we compare the color with every color contained in the color table, if there is a hit, we move on to next pixel, otherwise, we add this color to the color table.

The size of color table obtained this way may be less or greater than 256. In the first case, the conversion is done after the above operation. If the color table size is greater than 256, we must create a new color table using the alogrithsm discussed above:

(Code omitted)

If the size of color table is greater than 256, we first delete the color table, then create a new color table that contains only 256 colors. This color table comprises 256 colors that are evenly distributed in a 8(8(4 3-D space, which has the following contents:

(Table omitted)

For a 24-bit color, if we use only 3 most significant bits of red and green colors, and 2 most significant bits of blue color, and set rest bits to 0. Every possible RGB combination (8 bits for each color) has a corresponding entry in this table.

We use a flag bStandardPal to indicate which algorithm was used to generate the color table. This is important because for the two situations the procedure of converting explicit RGB values to indices of color table is different. If the color table is generated directly from the colors contained in the bitmap (first case), each pixel can be mapped to an index in the color table by comparing it with every color in the color table (there must be a hit). Otherwise, we must omit some bits before looking up the color table (second case).

Following is a portion of function CGDIDoc::OnConvertRGBto256() that allocates buffers from global memory, fill the buffers with bitmap information header and color table:

(Code omitted)

The differences between the new and old bitmap information headers are member bitBitCount (8 for 256-color format), biSizeImage, and biClrUsed (Member biClrUsed can be used to indicate the color usage. For simplicity, it is set to zero).

Next we need to convert explicit RGB values to color table indices. As mentioned before, there are two situations. If the color table is extracted directly from the bitmap, we must compare each pixel with every entry of the color table, find the index, and use it as the bitmap bit value. Otherwise the index can be formed by omitting the lower 5 bits of red and green colors, the lower 6 bits of blue color then combining them together. This eleminates the procedure of looking up the color table. It is possible for us to do so because the color table is created in a way that if we implement the above operation on any color contained in the table, the result will become the index of the corresponding entry.

For example, entry 1 contains color (32, 0, 0), which is (0x20, 0x00, 0x00). After bit omission, it becomes (0x01, 0x00, 0x00). The followng calculation will result in the index of this entry:

red | (green << 3) | (blue << 6)

By using this method, we do not need to look up the color table.

The following portion of function CGDIDoc::OnConvertRGBto256() shows how to convert explicit RGB values to indices of the color table:

(Code omitted)

Finally, we must unlock the global memory, destroy the original 24 bit bitmap, and assign the new handle to variable CGDIDoc::m_hDIB. We also need to delete the array that holds the temprory color table and update the view to redraw the bitmap image:

(Code omitted)

With this application, we can convert between 256-color and 24-bit bitmap formats back and forth. If the application is executed on a palette device with 256-color configuration, we may experience color distortion after converting a 256-color format bitmap to 24-bit format bitmap. This is because for this kind of bitmap, no logical palette is implemented in the application, so the color approximation method is applied by the OS.

10.8 Pixel Manipulation

With the above knowledge, it is easy for us to implement pixel manipulation on a DIB image. For different types of DIB formats, the procedure of manipulating pixel is different. If the format is color table based, we need to retrieve the color of a pixel through the color table. If the format is not color table based, we can directly edit the color of a pixel.

Sample 10.8\GDI demonstrates how to edit DIB image pixel by pixel. It is based on sample 10.7\GDI. The application will convert any color image to a black-and-white image. To make the conversion, we need to examine every pixel and find its brightness, average it, and assign the averaged value to each of the R, G, B factors. The brightness of a pixel can be calculated by adding up its R, G and B values. For 256-color format, since all the colors are stored in the color table, we can just convert the color table to a black-and-white one in order to make this change. No modification needs to be made to the pixels. For 24 bit format, we need to edit every pixel.

In the sample, a new command Convert | Black White is added to the mainframe menu IDR_MAINFRAME. Also, WM_COMMAND and UPDATE_COMMAND_UI message handlers are added to class CGDIDoc through using Class Wizard. The corresponding two new functions are CGDIDoc::OnConvertBlackwhite() and CGDIDoc::OnUpdateConvertBlackwhite(...) respectively.

Function CGDIDOC::OnUpdateConvertBlackwhite(...) is implemented as follows for supporting both 256-color and 24-bit formats:

(Code omitted)

For function CGDIDoc::OnConvertBlackwhite(), first we need to lock the global memory that is used for storing the bitmap, and judge its format by examining biBitCount member of structure BITMAPINFOHEADER:

(Code omitted)

If its value is 8, the format of the bitmap is 256-color. We need to change each color contained in the color table to either black or white color:

(Code omitted)

For every color, we add up its R, G, B values and average the result. Then we assign this result to each of the R, G, B factors.

The 24 bit format is slightly different. We need to examine each pixel one by one and implement the same conversion:

(Code omitted)

Finally, we need to unlock the global memory and update the view to reload the bitmap image.

Based on the knowledge we already have, it is not so difficult for us to enhance the quality of the images using other image processing methods, such as contrast and brightness adjustment, color manipulation, etc.

10.9 DIB Section: Using Both DIB and DDB

Sample 10.9\GDI demonstrates how to draw an image with transparent background on the client window. It is based on sample 10.8\GDI.

Importance of DDB

By now everything seems fine. We use DIB format to load, store image data. We also use it to draw images. Withoug converting from DIB to DDB and vice versa, we can manage the image successfully.

However, sometimes it is very inconvenient without DDB. For example, it is almost impossible to draw an image with transparency by solely using DIB (We will call the transparent part of an image "background" and the rest part "foreground"). Although we can edit every pixel and change its color, there is no way for us to prevent a pixel from being drawn, because functon ::SetDIBitsToDevice(...) will simply copy every pixel contained in an image to the target device (It does not provide different drawing mode such as bit-wise AND, OR, or XOR).

To draw image with transparency, we need to prepare two images, one is normal image and the other is mask image. The mask image has the same dimension with the normal image and contains only two colors: black and white, which indicate if a corresponding pixel contained in the normal image should be drawn or not. If a pixel in the normal image has a corresponding black pixel in the mask image, it should be drawn. If the corresponding pixel is white, it should not be drawn. By doing this, any image can be drawn with transparency.

A DDB image can be painted with various drawing modes: bit-wise AND, OR, XOR, etc. Different drawing modes will combine the pixels in the source image and the pixels in the target device differently. Special effects can be made by applying different drawing modes consequently.

When drawing a DDB image, we can use bit-wise XOR along with AND operation to achieve transparency. First, the normal image can be output to the target device by using bit-wise XOR mode. After this operaton, the output pattern on the target device is the XORing result of its original pattern and the normal image. Then the mask bitmap is output to the same position using bit-wise AND mode, so the background part (corresponding to white pixels in the mask image) of the device still remains unchanged (it is still the XORing result of the original pattern and the normal image), however, the foreground part (corresponding to black pixels in the mask image) becomes black. Now lets ouput the normal image to the target device using bit-wise XOR mode again. For the background part, this is equivalent to XORing the normal image twice with the original pattern on the target device, which will resume its original pattern (A^B^A = B).

For the foreground part, this operation is equivalent to XORing the normal image with 0s, which will put the normal image to the device (0^B = B).

Although the result is an image with a transparent background, when we implement the above-mentioned drawings, the target device will experience pattern changes (Between two XOR operations, the pattern on the target device is neigher the original pattern nor the normal image, this will cause flickering). So if we do all these things directly to the device, we will see a very short flickering every time the image is drawn. To make everything perfect, we can prepare a bitmap in the memory and copy the pattern on the target device to it, then perform XOR and AND drawing on the memory bitmap. After the the drawing is complete, we can copy the memory bitmap back to the device. For the memory bitmap, since its background portion has the same pattern with that of the target device, we will not see any flickering.

To paint a DDB image, we need to prepare a memory DC, select the bitmap into it, then call function CDC::BitBlt(...) or CDC::StretchBlt(...) to copy the image from one device to another.

In order to draw an image with transparent background, we need to prepare three DDBs: the normal image, the mask image, and the memory bitmap. For each DDB, we must prepare a memory DC to select it. Also, because the DDB must be selected out of DC after drawing, we need to prepare a CBitmap type pointer for each image.

Since the usr can load a new image when there is an image being displayed, we need to check the states of variables that are used to implement DCs and bitmaps. Generally, before creating a new memory DC, it would be safer to check if the DC has already been initialized. If so, we need to delete the current DC and create a new one. Before deleting a DC, we further need to check if there are objects (such as bitmap, palette) currently being selected. All the objects created by the user must be selected out before a DC is delected. The DC can be deleted by calling function CDC::DeleteDC(). Also, before creating a bitmap, we need to check if the bitmap has been initialized. If so, before creating a new bitmap, we need to call function CGDIObject::DeleteObject() to destroy the current bitmap first.

Functions CBitmap::CreateBitmap(...) and CDC::CreateCompatibleDC(...) will fail if CBitmap and CDC type variables have already been initialized.

If a logical palette is implemented, we must select it into every DC before performing drawing operations. Before the application exits, all the objects selected by the DCs must be selected out, otherwise it may cause the system to crash.

Although we can prepare normal image and mask image separately, it is not the most convenient way to implement transparent background. The mask image can also be generated from the normal image so long as all the background pixels of the normal image are set to the same color (For example, white). In this situation, pixel in the mask image can be set by examing the corresponding pixel in the normal image: if it is the background color, the pixel in the mask image should be set to white, otherwise it should be set to black.

DIB Section

Both DIB and DDB are needed in order to implement transparent background drawing: we need DIB format to generate mask image, and need DDB to draw the image. Of course we can call ::GetDIBits(...) and ::SetDIBits(...) to convert between DIB and DDB format, however, there exists an easier way to let us handle DIB and DDB simultaneously.

A DIB section can be created to manage the image so that we can have both the DIB and DDB features without doing the conversion. A DIB section is a memory section that can be shared between the process and the system. When a change is made within the process, it is automatically updated to the system. By doing this, there is no need to update the data using functions ::GetDIBits(...) and ::SetDIBits(...).

We can call function ::CreateDIBSection(...) to create a DIB section. This function will return an HBITMAP handle, which can be attached to a CBitmap variable by calling function CBitmap::Attach(...).

Function ::CreateDIBSection(...) has six parameters:

```
HBITMAP ::CreateDIBSection

(

HDC hdc,

CONST BITMAPINFO *pbmi, UINT iUsage, VOID *ppvBits, HANDLE hSection,

DWORD dwOffset

);
```

(Table omitted)

After calling this function, we can access the buffers pointed by ppvBits and make change to the DIB bits directly, there is no need for us to do any DIB to DDB conversion or vice versa. After the change is made, we can draw the bitmap immediately by calling funciton CDC::BitBlt(...), this will draw the updated image to the window.

New Variables

The following new variables are declared in class CGDIView for drawing bitmap with transparancy:

(Code omitted)

Altogether there are four CBitmap type variables, three CBitmap type pointers, three CDC type variables, and three CPalette type pointers. Their meanings are explained in the following table:

(Table omitted)

To make the application more interesting, we will also draw the background of client window using bitmap. This bitmap will be loaded into m_bmpBkd variable.

The six pointers are initialized to NULL in the constructor:

(Code omitted)

Cleaning Up

A new function CGDIView::CleanUp() is added to the application for doing the clean up job. Within this function, all the objects selected by the DCs are selected out, then DCs and bitmaps are deleted:

(Code omitted)

If a pointer is not NULL, it means that there is an object being currently selected by the DC, so funciton CGDIObject::SelectObject(...) is called to select the object (palette or bitmap) out of the DC before destroying it.

We need to call this function just before the application exits. In the sample, a WM_DESTROY message handler is added to class CGDIView through using Class Wizard. The corresponding member function is implemented as follows:

(Code omitted)

Loading Bitmap & Creating Mask Bitmap

In the sample, funciton CGDIView::LoadBitmap(...) is changed. In the new function, a DIB section is created from the DIB data, and the mask bitmap image is generated from the normal image. The palette creation precedure is still the same. The handle retruned from the DIB section is attached to variable CGDIView::m_bmpDraw:

(Code omitted)

Please note that function CGDIView::CleanUp() is called before the bitmap is created. After the DIB section is created, we use the DIB data passed through hData parameter to initialize the image. The buffers that store DIB bit values are pointed by pointer pBits. We can use it to edit the image pixels directly, there is no need to convert between DDB and DIB foramts any more.

After the bitmap is loaded, we need to create the mask bitmap, memory bitmap, and theree memory DCs. We also need to select the bitmaps and the logical palette into the DCs if necessary:

(Code omitted)

The mask and memory bitmaps must be created by calling function CBitmap:: CreateCompatibleBitmap(...), this will allow the created bitmaps to be compatible with the device context.

Next, the mask bitmap is generated from the normal bitmap image:

(Code omitted)

Every pixel of the normal image is examined to generate the mask image. Here functions CDC::GetPixel(...) and CDC::SetPixel(...) are called for manipulating single pixels. Although the two functions hide the details of device context and bitmap format, they are very slow, and should not be used for fast bitmap drawing or image processing.

Drawing Bitmap with Transparancy

Function CGDIView::OnDraw(...) is modified as follows for drawing bitmap with transparency:

(Code omitted)

In the above function, the pattern on the target device is first copied to the memory bitmap. Then function CDC::BitBlt(...) is called three times to draw the normal image and mask image on the memory bitmap, with two XOR drawings of the normal image (first and thrid operations) and one AND mode drawing of the mask image (second operation). Finally, the new pattern in the memory bitmap is copied back to the targert device.

Adding Background

If the window's background is also white, it is difficult for us to see the transparency effect. To show this effect, in the sample, the background of the client window is also painted with a bitmap image.

The image that is used to paint the background is prepared as a resource, whose ID is IDB_BITMAPBKD. The bitmap is loaded to variable CGDIView::m_bmpBkd in function CGDIView:: OnInitialUpdate():

(Code omitted)

The bitmap can also be loaded in the constructor of CGDIView.

To paint the background of a window, we need to handle message WM_ERASEBKGND and implement background drawing by ourseleves in the message hanlder. In the sample, this message handler is added through using Class Wizard, and the corresponding function is implemented as follows:

(Code omitted)

This function simply draw the bitmap image repeatedly so that the whole client area is covered by the image.

To test this sample, we may use it to load any bitmap images with white background.

Figure 10-1 shows the result after 10.9\Rose.bmp is loaded into the application.

(Figure 10-1 omitted)

10.10 Creating Chiseled Effect

We will continue to show the power of DDB. Sample 10.10\GDI demonstrates how to convert a normal bitmap to a grayed image with chiseled effect. It is based on sample 10.9\GDI. We can see that with DDB, this effect can be easily implemented with just a few CDC::BitBlt(...) calls. If we use DIB, we will have to make very complex mathematical calculation, and the program will become very slow.

The chiseled effect can be implemented by drawing the outline of an object with two different colors: highlighted color and shadowed color. Usually white is used as the the highlighted color and dark gray is used as the shadowed color. For example: the rectangle in Figure 10-2 uses white and dark gray as the highlighted and shadowed colors respectivly, it creates a chiseled effect:

Algorithm

The chiseled effect can be implemented with the following algorithm: find out the object's outline, imagine some parallel lines with 135(angle are drawn from the upper left side to bottom right side (Figure 10-3). Think these lines as rays of light. If we draw the portion of the outline that first encounters these parallel lines (the portion facing the light) with shadowed color, and draw the rest part of the outline (the portion facing away fromt he light) with highlighted color, the object will have a chiseled effect. If we swap the shadowed and highlighted colors, it will result in an embossed effect.

If we have a 2-D binary image (an image that contains only two colors: white (255, 255, 255) and black (0, 0, 0)), the outline can be generated by combining the inverse image with the original image at an offset origin. For example, the outline that should be drawn using the shadowed color can be generated with the following steps:

1) Draw the original bitmap at position (0, 0).

2) Invert the bitmap image.

3) Combine the inverted image with the image drawn in step 1) with bit-wise OR operation, with the inverted image be put at position (1, 1) (Figure 10-4).

The highlighted outline can be obtained in the same way, but we need to combine the original bitmap and the inverted bitmap differently here:

1) Draw the original bitmap at position (1, 1).

2) Invert the bitmap image.

3) Combine the inverted image with the original image with bit-wise OR operation, with the inverted image be put at position (0, 0) (Figure 10-5).

If we combine the two outlines and paint them with highlighted and shadowed colors respectively, then fill the rest part with a normal color (A color between the highlighted and shadowed color), we will have a 3D effect. For example, we can use white as the highlighted color, dark gray as the shadowed color, and light gray as the normal color.

Creating Binary Bitmap Image

So we need to generate monochrome black/white binary image from the original color bitmap. Of course we can examine every pixel one by one and compare its brightness with a threshold (Usually the threshold value is set to the middle between the brightest and darkest color, which are white and black respectively). If the brightness of a pixel is greater than the threshold, its color is set to white, otherwise it will be set to black.

However, we can implement this conversion in an easier way. Remember, when creating the bitmap by calling function CBitmap::CreateBitmap(...), we are allowed to specify the bitmap's attributes, which includes the dimension of the image, number of bytes in each raster line, and bit count per pixel. Here, the bit count per pixel indicates how many bits will be used for representing a single pixel. If we set it to 1, the image can have only 2 colors, and will be automatically converted to mono chrome fomat no matter what kind of data we use to initialize the bitmap.

Raster Operation Mode

Another issue needs to be discussed here is how to draw the outline portion using specified color without affecting rest part of the image. In order to implement this, we can treat the two outline bitmaps as masks, and draw only the unmasked pixels with specified colors. This procedure is similar to that of drawing bitmaps with transparency.

When calling function CDC::BitBlt(...) or CDC::StretchBlt(...) to paint the bitmap, we always need to specify the drawing mode, which specifies how to combine the pixels in the source bitmap with the corresponding destination pixels. This drawing mode is also called raster operation mode, because it is applicable only to raster devices (Contary to the raster devices are vector devices, for example, a plotter is a vector device). We have many choices such as bit-wise AND, OR, XOR etc. Actually, we can specify more complex combinations among the following three different objects when calling the above two functions: the pixel in the source bitmap, the corresponding pixel in the target device, and the brush currently being selected by the DC. We can specify up to three Boolean bit-wise operations among them.

For example, the following operation will draw the outline on the destination bitmap with the brush color:

(Brush Color) XOR (Destinaton Color) AND (Source Color) XOR (Brush Color)

The reason is simple: After the first operation (between the brush and destination pixels), the pixels in the target device will become the XOR combination between the original pixel colors and the brush color. Next, this XORed result will be ANDed with the source bitmap (Only the outlined part is black, rest part is white), the outlined part on the target device will become black and the rest part remains unchanged (still XORed result from the first operation). Then we do XOR again between the target device and the brush, for the outlined part, this operation will fill it with the brush color (A ^ 0 = A); for the rest part, this will resume the original color for every pixel (A ^ B ^ A = B).

Chiselled Effect

With the following steps, we can create chiselled effect:

1) Create a brush with shadowed or highlighted color.

2) Paint the destination with the brush using bit-wise XOR operation mode.

3) Draw the mask bitmap on the target device using bit-wise AND operation mode.

4) Paint the destination with the brush using bit-wise XOR operation mode again.

This will draw one of the outlines. Creating new brush and repeating the above steps using the other mask bitmap can result in the chiselled effect.

The first and fourth steps can be implemented by calling function CDC::PatBlt(...), which allows us to fill a bitmap with a brush using specified operation mode:

```
BOOL CDC::PatBlt(int x, int y, int nWidth, int nHeight, DWORDdwRop);
```

The last parameter allows us to specify how to combine the color of the brush with the destination pixels. To do bitwise XOR, we need to specify PATINVERT mode.

So we can call CDC::PatBlt(...), CDC::BitBlt(...) and CDC::PatBlt(...) again to draw the outline using the brush created by our own. However, there is a simpler way. When calling function CDC::BitBlt(...), we can pass it a custom operation code and let it do the above-mentioned operations in one stroke.

To find out the custom operation code, we need to enumerate all the possible results from the combinations among brush, destination and source bits for our raster operation:

(Brush Color) XOR (Destinaton Color) AND (Source Color) XOR (Brush Color)

The following table lists all the possible results from the above fomulae:

(Table omitted)

The sequence in the table must be arragned so that brush is in the first column, source pixel in the second column and destination pixel in the third column. The code resulted from the combination of the three bits must increment by one for adjacent rows (In the above sample, for the first row it is 000, second row it is 001, third row it is 010...). If we read the output from the bit contained in the last row to the bit in the first row (10111000), we will have 0xB8.

With this index, we can find a raster code that will implement this typical operation for either CDC::BitBlt(...) or CDC::StretchBlt(...) calling. The table is documented in Win32 programming, we can also find it in Appendix B.

By looking up the table, we know that the raster code needs to be use is 0x00B8074A.

Highlighted and Shadowed Colors

The standard highlighted and shadowed colors in the system can be retrieved by calling function ::GetSysColor(...), which has the following format (The standard system colors can be set by calling the counterpart function ::SetSysColor(...)):

```
DWORD ::GetSysColor(int nIndex);
```

The following is a list of some values of nIndex parameter that could be used to retrieve some standard colors in the system:

(Table omitted)

In the sample, we choose COLOR_BTNHIGHLIGHT as the highlighted color, and COLOR_BTNSHADOW as the shadowed color.

New Function

In the sample, a new function CreateGrayedBitmap(...) is declared in class CGDIView to create grayed image from a nomal bimap:

(Code omitted)

The only parameter to this function is a CBitmap type pointer. The function will return an HBITMAP handle, which is the grayed bitmap. Within the function, we must prepare three bitmaps: the bitmap that will be used to store the final grayed image, the mask image that stores the shadowed outline, and the mask image that stores the highlighted outline. The function starts with creating these bitmaps:

(Code omitted)

The final grayed image will be stored in variable bmpGray. We will refer image created by this variable as "grayed image", although the image may not be grayed in the interim.

The other two CBitmap type variables, bmpHilight and bmpShadow will be used to store the outline mask images. We need two memory DCs, one used to select the color bitmap (normal image passed through pointer pBmp, and grayed image bmpGray) and one for binary bitmaps (the outline mask bitmaps). Note that binary bitmaps are created with bit count per pixel set to 1 and the grayed bitmap (actually it is a color bitmap, but we use only monochrom colors) is created by calling function CBitmap:: CreateCompatibleBitmap(...). Since the DC supports color bitmap (If the program is being run on a system with a color monitor) , this will create a color bitmap compatible with the window DC.

Then we select the color bitmap and the binary bitmaps into the memory DCs and create the outline mask images:

(Code omitted)

First we fill the mask bitmap with white color. Then we copy the patterns from the original image to the mask bitmap image. When doing this copy, the first horizontal line (the upper-most line) and the first vertical line (the left-most vertical line) are eleminated from the original image (Pixels with coordinates (0, y) and (x, 0) are elemented, where x can be 0, 1, ... , up to width of image -1; y can be 0, 1, ..., up to height of image -1). The colors contained in the souce bitmap will be automatically converted to black and white colors when we call function CDC::BitBlt(...) because the target image is a binary bitmap. The souce image is copied to the mask bitmap at the position of (0, 0). Then the original bitmap is inverted, and merged with the mask image with bit-wise OR operation. Here flag MERGEPAINT allows the pixels in the souce image and pixels in the target image to be combined in this way. After these operations, the binary bitmap image will contain the outline that should be drawn with the shadowed color.

The following portion of the function generates the highlighted outline:

(Code omitted)

Next we create a brush with standard button face color and used it to fill the grayed image (By default, the standard button face color is light gray. It can also be customized to other colors):

(Code omitted)

The button face color is retrieved by calling ::GetSystColor(...) API function. Actually, all the standard colors defined in the system can be retrieved by calling this function. Next, we draw the highlighted outline of the grayed bitmap using the standard highlighted color:

(Code omitted)

Also, the shadowed outline is drawn on the grayed image in the same way:

(Code omitted)

Finally, some clean up routines. Before this function exits, we call function CBitmap::Detach() to detach HBITMAP type handle fromCBitmap type variable. This is because we want to leave the HBITMAP handle for further use. If we do not detach it, when CBitmap type variable goes out of scope, the destructor will destroy the bitmap automatically, and therefore, the bitmap handle will no longer be valid from then on.

Function CGDIView::LoadBitmap(...) & CGDIView::OnDraw(...)

In the sample, funciton CGDIView::LoadBitmap(...) is changed so that when the user opens a normal color bitmap, the application will automatically convert it to a grayed bitmap image. The original variables that are used to implement image transparency along with the relavant functions are deleted. The only variable remained in class CGDIView are m_bmpDraw, m_dcMem, m_pBmpOld and m_pPalOld:

(Code omitted)

Function CGDIView::OnInitialUpdate(...) and CGDIView::OnCleanUp() are modified in order to conform this change. Also, WM_ERASEBKGND message handler is removed through using Class Wizard.

In function CGDIView::OnLoadBitmap(...), after the palette is created, we call function CGDIView:: CreateGrayedBitamp(...) to create the grayed bitmap and attach the returned handle to variable m_bmpDraw:

(Code omitted)

The bitmap that was originally created by function ::CreateDIBSection(...) is destroyed. Finally, function CGIDView::OnDraw(...) is changed to draw the grayed bitmap to the client window:

(Code omitted)

With the above implementations, the application is able to create grayed images with chiselled effect.

Summary

1) Usually an image is stored to disk using DIB format. To display it on a specific type of device, we must first convert it to DDB format, which may be different from device to device.

2) To draw bitmap, we must prepare a memory DC, select the bitmap into it, and copy the image between the memory DC and target DC.

3) Function CDC::BitBlt(...) can be used to draw the image with 1:1 ratio. To draw an image with an enlarged or shrunk size, we need to use function CDC::StretchBlt(...).

4) A DIB contains three parts: 1) Bitmap information header. 2) Color Table. 3) DIB bit values. For the DIB files stored on the disk, there is an extra bitmap file header ahead of DIB data.

5) We can call function ::GetDIBits(...) to get DIB bit values from a DDB selected by a DC, and call function ::SetDIBits(...) to set DIB bit values to the DDB.

6) There are several DIB formats: monochrom format (2 colors), 16-color format, 256-color format, 24-bit format. For each format, the total number of colors contained in the color table is different. The pixels of 24-bit DIB contain explict RGB values, for the rest formats, they contain indices to a color table which resides in the bitmap information header.

7) In order to accelarate bitmap image loading and saving, each raster line of the imge must use multiple of 4 bytes for storing the image data. The extra buffers will simply be wasted if there are not enough image data.

8) The following information contained in the bitmap information header is very important: 1) The dimension of the image (width and height). 2) Bit count per pixel. 3) Image size, which can be calculated from the image dimension and bit cout per pixel.

9) The size of a color table (in number of bytes) can be calculated from the following formula:

for 24-bit format: 0

for other formats that contain color table: (size of structure RGBQUAD) (2Bit count per pixel

10) The total image size can be calculated from the following formula:

for 24-bit format:

(size of bitmap information header) + (number of bytes for one raster line) (image height

for other formats that contain color table:

(size of bitmap information header) + size of color table + (number of bytes for one raster line) (image height

where number of bytes for on raster line can be calculated as:

(((bit count per pixel) ((image width) + 31)/32) (4

Before preparing DIB, we need the above information to allocate enough buffers for storing DIB data.

11) Function ::SetDIBitsToDevice(...) can be used to draw DIB directly to a device. We don't need to implement any DIB-to-DDB conversion. However, using this function, we also lose the control over DDB.

12) DIB section can be created for managing the image in both DIB and DDB format.

13) Bitmap with transparency can be implemented by using a mask image, and drawing the normal and mask images using bit-wise XOR and AND operation modes.

14) We can convert a color bitmap to a grayed image with chiselled or embossed effect by finding out the outline of the object, then drawing the portion facing the light with highlighted (shadowed) color and the portion facing away from the light with shadowed (highlighted) color.

BACK TO INDEX

Chapter 11 Sample: Simple Paint

This chapter introduces a series of samples imitating standard "Paint" application, using the knowledge from previous chapters. Also, some very useful concepts such as region, path are discussed. By the end of this chapter, we will be able to build simple graphic editor applications.

Samples in this chapter are specially designed to work on 256-color palette device. To customize them for non-palette devices, we can just eleminate logical palette creation and realization procedure.

11.0 Preparation

With the knowledge we already have, it is possible for us to built a simple graphic editor now. So lets start to build an application similar to "Paint". Sample 11.0\GDI is a starting application whose structure is similar to what we have implemented in previous chapters. The application has the following functionalities: 1) Device independent bitmap loading and saving. 2) DIB to DDB conversion (implemented through DIB section). 3) Displaying DDB using function CDC::BitBlt(...). Lets first take a look at class CGDIDoc:

(Code omitted)

We will support only 256 color device, so in the constructor, a logical palette with size of 256 is created, the first 20 entries are filled with predefined colors. Later when we implement the application, colors contained in the first 20 entries of this logical palette will be displayed on a color bar that could be used by the user for interactive drawing. In the sample, variable CGDIDoc::m_palDraw implements a logical palette, which will be used throughout the application's lifetime. In the constructor, a default palette is created and the current colors in the system palette are used to initialize the logical palette (Of course, we can also initialize the logical palette with user-defined colors). When a new bitmap is loaded, colors contained in the color table of the bitmap will be used to fill the logical palette.

After the DIB is loaded, its handle will be stored in variable CGDIDoc::m_hDIB, which is initialized to NULL in the constructor. In function CGDIDoc::Serialize(...), the bitmap is loaded into memory and stored to disk.

Function CGDIDoc::GetHDib() and CGDIDoc::GetPalette() let us access the DIB and logical palette outside class CGDIDoc.

The following is a portion of class CGDIView:

(Code omitted)

Here variable m_bmpDraw is used to store the device dependent bitmap, variable m_dcMem is the memory DC that will be used to select this bitmap. Other two pointers m_pBmpOld and m_pPalOld will be used to resume m_dcMem's original state.

Function CGDIView::LoadBitmap(...) will be called from function CGDIView::OnInitialUpdate(), when a new bitmap is loaded by the application. In this function a DIB section will be created, and the returned HBITMAP handle will be attached to variable CGDIView::m_bmpDraw. So any operation on the DDB will be reflected to DIB bit values. Also if we modify DIB bits, the DDB will be affected automatically. In the function, the color table contained in the DIB is extracted, and the entries of the logical palette (implemented in the document) are updated with the colors contained in the bitmap file by calling function CPalette::SetPaletteEntries(...).

After a bitmap is loaded, it will be painted to the client window by calling CDC::BitBlt(...) in function CGDIView::OnDraw(...). Every time before the bitmap is painted, the logical palette contained in the document is selected into the target DC and realized. By doing this, we can avoid color distortion.

Function CGDIView::CleanUp() selects the palette and bitmap out of the DC, then deletes the memory DC and DIB. It is called from the following two functions: 1) In CGDIView::OnDestroy() when the application is about to exit. 2) In CGDIView::LoadBitamp() before new DDB is created.

That's all the features included in sample 11.0\GDI. The application can load a DIB file from the disk and display it.

11.1 Ratio and Grid

Sample 11.1\GDI is based on sample 11.0\GDI, it implements zoom in and zoom out commands. Also, when the image is displayed with an enlarged size, the grid can be turned on.

For a graphic editor, it is desirable that the image can be displayed with different ratios. Also, When the image is zoomed in, we need to add grid to let the user have a better view of pixels. These two features are included in almost all the graphic editors.

We know it is easy to display an image in different ratios. In order to do this, we need to call function CDC::StretchBlt(...) instead of CDC::BitBlt(...). The only concern here is that we should let the user select different ratios with mouse clicking, and whenever the ratio changes, the effect should be shown in the client window at once.

Zoom In & Zoom Out

So we need to add a new variable that can be used to store the current image ratio. The bitmap image should be drawn in the client window according to the value of this variable, and the user can change it through mouse clicking. Although this variable can be included in any of the four classes (Any of CGDIApp, CFrameWnd, CGDIDoc or CGDIView derived classes), generally we'd like to put the data in document because this will make the application document centered. For the application that has only one document and view it doesn't make much difference where we put this variable. But if an application has more than one document or view, we should consider this more carefully. For example, suppose we have two documents opened at the same time, if both documents need to share a same feature (for example, the ratio change will affect both documents), the variable needs to be put in the frame window class. If we don't want to affect the other document when changing the feature of one document (for example, the ratio change for one document should not affect the ratio of another document), we should let each document have its own variable.

In the sample application, we use an integer type variable m_nRatio to record the current ratio. This variable is initialized to 1 in the constructor. A member function CGDIDoc::GetRatio() is added to let this value be accessible from other classes. To let the user be able to change the ratio of the image, two buttons are added to toolbar IDR_MAINFRAME. The IDs of the two buttons are ID_ZOOM_IN and ID_ZOOM_OUT, one of them lets the user zoom in and the other let the user zoom out the image.

Both of the two commands have WM_COMMAND and UPDATE_COMMAND_UI message handlers. The message handlers allow the user to change the current ratio, which are relatively easy to implement. Within the function, we need to judge if the current value of ratio will reach the upper or lower limit, if not, we should increment or decrement the value, and update the client window. For example, function CGDIDoc::OnZoomIn() is implemented as follows:

(Code omitted)

The lower limit of the ratio value is 1 and the upper limit is 16. Another concern is that we must also change the scroll sizes of the client window whenever the ratio has changed (In sample 11.0\GDI, since the image does not change after it is displayed in the client window, it is enough to just set the scroll sizes according to the image size after the bitmap is loaded).

To let the scroll sizes be set dynamically, a new member function CGDIView::UpdateScrollSizes() is added to the application. In this function, the current ratio value is retrieved and the scroll sizes are set to the zoomed bitmap size. In the sample, this newly added function is also called in function CGDIView::OnInitialUpdate() to set the scroll sizes according to the image size whenever a new bitmap is loaded (The old implementation is elemenated).

Functions CGDIDoc::OnUpdateZoomIn(...) and CGDIDoc::OnUpdateZoomOut(...) are used to set the state of zoom in and zoom out buttons. We should disable both commands when there is no bitmap loaded. Besides this, upper and lower limits are also factors to judge if we should disable either zoom in or zoom out command. For example, CGDIDoc::OnUpdateZoomIn(...) is implemented as follows:

```
void CGDIDoc::OnUpdateZoomIn(CCmdUI* pCmdUI)

{

pCmdUI->Enable(m_nRatio < 16 && m_hDIB != NULL);

}
```

Grid

Grid implementation is similar. Since grid has only two states (it is either on or off), a Boolean type variable is enough for representing its current state. In the sample, a Boolean type variable m_bGridOn is added to class CGDIDoc, which is initialized to FALSE in the constructor. Besides this, an associate function CGDIDoc::GetGridOn() is added to allow its value be retrieved outside the document. Also, a new button (whose command ID is ID_GRID) is added to tool bar IDR_MAINFRAME, whose message handlers are also added through using Class Wizard. The value of m_bGridOn is toggled between TRUE and FALSE in function CGDIDoc::OnGrid(). Within function CGDIDoc::OnUpdateGrid(...), the button's state (checked or unchecked) is set to represent the current state of grid:

```
void CGDIDoc::OnUpdateGrid(CCmdUI* pCmdUI)

{

pCmdUI->SetCheck(m_bGridOn == TRUE);

}
```

We must modify function CGDIView::OnDraw(...) to implement grid. First we need to check the current value of CGDIDoc::m_bGridOn. If it is TRUE, we should draw both the image and the grid; if it is FALSE, we need to draw only the image.

We can draw various types of grids, for example, the simplest way to implement grid would be just drawing parallel horizontal and vertical lines. However, there is a disadvantage of implementing grid with solid lines. If the image happens to have the same color with grid lines, the grid will become unable to be seen. An alternate solution is to draw grid lines using image's complement colors, this can be easily implemented by calling function CDC::SetROP2(...) and passing R2_NOT to its parameter before the grid is drawn. However, this type of grid does not have a uniform color, this makes the image looks a little awkward.

Pattern Brush and Its Origin

The best grid is implemented with alternate colors (for example, black and white), thus at any time one of the two contiguous grid pixels will always have a different color with the image pixels under them (When an image is enlarged, one pixel of the image will become several pixels). We might want to use dashed or dotted lines to implement this type of grid. However, the alternating frequency of dotted or dashed lines is not one pixel.

In the sample a pattern brush is used to implement grid. Remember we can use pattern brush to fill a rectangle with a bitmap pattern. If we limit width and height of the pattern brush to 1 unit, the filling result will become a straight line (vertical or horizontal). We can prepare a bitmap with the pattern that any two adjacent pixels (not diagnal) have different colors, and use it to create the pattern brush (Figure 11-1).

If we write program for Windows 95, the size of the bitmap for making pattern brush must be 8(8. In the sample, this image is included in the application as a bitmap resource, whose ID is IDB_BITMAP_GRID. The variable used for creating pattern brush is CGDIView::m_brGrid, and the pattern brush will be created in the constructor of class CGDIView.

In function CGDIView::OnDraw(...), after drawing the bitmap, we must obtain the value of CGDIDoc::m_bGridOn. If it is true, we will use the pattern brush to draw the grid. When using pattern brush, we must pay special attention to its origin. By default, the brush's origin will always be set to (0, 0). This will not cause problem so long as the client window is not scrolled. However, if scrolled position (either horizontal or vertical, but not both) happens not to be an even number, we need to adjust the origin of the pattern brush to let the pattern be drawn started from 1 (horizontal or vertical coordinate). This is because our pattern repeats every other pixel.

Figure 11-2 demonstrates the two situations. In the left picture, the logical coordinates of the upper-left pixel of the visible client window are (2, 2). If we draw the grid starting from the pixel located at the logical coordinates (0, 0), the grid pixel at (2, 2) should be drawn using dark color (See Figure 11-1). If the client window is further scrolled one pixel leftward (the right picture of Figure 11-2), the logical coordinates of the upper-left pixel of the visible client window become (3, 2). In this situation, it should be drawn using the light color. However, if we do not adjust the origin of the pattern brush, the system will treat the upper-left visible pixel in the client window as the origin and draw it using the dark color.

To set pattern brush's origins, we need to call the following two functions before selecting brush into the DC:

BOOL CGDIObject::UnrealizeObject();

CPoint CDC::SetBrushOrg(int x, int y);

In the second function, x and y specify the new origin of the pattern brush.

In the sample, the brush origin is set according to the current scrolled positions. The following code fragment shows how the origin is adjusted in function CGDIView::OnDraw(...):

(Code omitted)

Here m_brGrid is a CBrush type variable that is used to implement the pattern brush, pt is a POINT type variable whose value is retrieved by calling function CScrollView::GetScrollPosition().

Drawing horizontal grid lines and vertical grid lines are implemented separately. We use two loops to draw different types of lines. Within each loop, function CDC::PatBlt(...) is called to draw one grid line. The following code fragment shows how the horizontal grid lines are drawn in the sample application:

(Code omitted)

The height of line is set to 1, so the actual result will be a pattern line.

11.2 Color Selection

Sample 11.2\GDI is based on sample 11.1\GDI. In this sample, a "Color Bar" is implemented, it allows the user to select current color from a series of colors (Figure 11-3).

When the user is editing the image, both foreground and background color need to be set. The foreground color will be used to draw line, curve, arc, or the border of rectangle, ellipse, polygon, etc. The background color will be used to fill the interior of rectangle, ellipse and polygon. In the sample, the user can left click on any color contained in the color bar to select a foreground color, and right click on any color to select a background color.

We know that this feature is similar to that of standard graphic editor "Paint". In "Paint" application, color bar is docked to the top or bottom border of the mainframe window. There are two rows of colors that can be selected for drawing. The user can use left and right mouse buttons to select foreground and background colors, double click on any color to customize it.

We need to recollect some old knowledge from chapter 1 through chapter 4 in order to implement the color bar.

Color Selection Control

First, the color bar should be implemented by dialog bar. This will allow it to be docked or floated, and we can include any type of common controls very easily. Second, we need to decide what type of control is needed for implementing the color selection controls.

The color selection control should have the following features: 1) It can respond to mouse clicking events (Left and right clicking, also, the double clicking). 2) The surface of the control should be painted with the color it represents.

There are many ways to implement this control. One solution is to use owner-draw button. Remember if we set a button's style to "Owner draw", when the button needs to be updated, its parent window will call function CBitmapButton::DrawItem(...). We can override this member function and paint the surface of the button with the color it is representing. The advantage of this method is that by doing this, all the buttons will be instances of the same class, and the same member function will be called to draw every button. In this case, it is relatively easy to add or delete such type of buttons without having to rewrite the code for drawing every single button. Imagine if we handle button drawing in the parent window, we have to calculate the position and size of each button whenever it needs to be redrawn.

In the sample, a dialog box template with ID of IDD_DIALOG_COLORBAR is added to the application. There are altogether twenty-one owner-draw buttons. Among them, one button will be used to display currently selected foreground and background colors, the rest buttons will be used for displaying colors contained in the logical palette.

We must create new classes for the controls contained in the dialog bar. In the sample, class CColorButton and CFBButton are added for this purpose. Both of them are derived from the class CBitmapButton, also, both of them override function CBitmapButton::DrawItem(...). Please note that instead of handling message WM_DRAWITEM in the derived classes, we must override CBitmapButton::DrawItem(...) to customize the appearance of button. This is because message WM_DRAWITEM will not be routed to derived classes (Only the base class will receive this message, in which case default function CBitmapButton::DrawItem(...) will be called). If we do not override this function, we will not be notified when buttons need to be updated.

Since this sample is supposed to be used for palette device (Of course, it can be run on a non-palette device), we will let each button display a color contained in a different entry of the logical palette. In order to do this, we should let different button have a different index that represents a different entry of the logical palette. For this purpose, a variable m_nPalIndex and two functions (GetPaletteIndex() and SetPaletteIndex(...)) are added to class CColorButton. In function CColorButton::DrawItem(...), this value is used as the index to the application's logical palette for button drawing:

(Code omitted)

We use macro PALETTEINDEX to retrieve the actual color contained in the palette entry. As usual, before doing any drawing, we have to select the logical palette into the DC and realize it.

Class CFBButton is similar. Two variables m_BgdIndex and m_FgdIndex are added to class CGDIDoc representing the currently selected foreground and background colors. Their values can be retrieved and set through calling functions CGDIDoc::GetBgdIndex(), CGDIDoc::GetFgdIndex(), CGDIDoc:: SetBgdIndex(...), CGDIDoc::SetFgdIndex(...). Two variables are declared in the document class instead of color bar class because their values may need to be accessed from the view. Since the document is the center of the application, we should put the variables in the document so that they can be easily accessed from other classes.

Function CFBButton::Drawitem(...) implements drawing a rectangle filled with current background color overlapped by another rectangle filled with current foreground color. Like class CColorButton, the color is retrieved from the logical palette contained in the document. The following code fragment shows how the background rectangle is drawn:

(Code omitted)

Variable nBgdIndex is an index to the logical palette, the whole area that needs to be painted is specified by lpDrawItemStruct->rcItem (lpDrawItemStruct is the pointer passed to function DrawItem(...)). When drawing the rectangle, we see that a margin of 2 is left first (This is done through calling function CRect:: InflateRect(...)), then the width and height of the rectangle are set to 3/4 of their original values. The foreground rectangle has the same dimension, but overlaps the background rectangle. To add more fluff to the application, the border of both rectangles has a 3D effect, which is implemented by calling function CDC::DrawEdge(...).

Color Bar

To implement color bar, a new class derived from CDialogBar is added to the application. This class is named CColorBar. To let the buttons act as color selection controls, we need to implement subclass for all the owner-draw buttons. In the sample, function CColorBar::InitButtons() is added to initialize the indices of all the buttons and implement subclass. Also, function CDialogBar::Create(...) is overridden, within which CColorBar::InitButtons() is called to change the default properties of the buttons. The following is the implementation of function CColorBar::InitButtons():

(Code omitted)

Please note that in the sample, the first color button's ID is IDC_BUTTON_COLOR1, and the IDs of all the color buttons are consecutive. This may simplify message mapping.

Color Selection

Another feature implemented in the sample application is that the user may set foreground and background colors by left/right mouse clicking on a color selection control. Also, the color of the color selection control may be customized by double clicking on the button. Since the messages related to mouse events will not be routed to the child window of dialog box (We can treat dialog bar as a dialog box), they are handled in base class CColorBar. In the sample, functions CColorBar::OnLButtonDown(...), CColorBar::OnRButtonDown(...) and CColorBar::OnLButtonDblClk(...) are implemented to handle mouse clicking messages. In the first two functions, first the foreground or background palette index contained in the document is set to a new value according to which button is clicked, then the button being clicked is updated. In the third function, a color dialog box is implemented, if the user selects a new color, we will use it to fill the corresponding entry of the logical palette, then update all the color buttons.

Integrate the Color Bar into the Program

Finally, the color bar is created in function CMainFrame::OnCreate(...). The variable that is used to implement the color bar is CMainFrame::m_wndColorBar. Within this function, CColorBar::Create(...) is called to create the color bar and CControlBar::EnableDocking(...) is called to dock the color bar. The color bar can be either floated or docked to any border of the mainframe window.

11.3 Simple Drawing

Sample 11.3\GDI is based on sample 11.2\GDI.

The new application implemented in this section supports two most basic editing functions: dot drawing and line drawing. Remember we have implemented some basic interactive drawings in chapter 9, but the implementation here is slightly different. As the user draw a dot or line, besides updating the client window, we also need to update the new drawings to the bitmap, so when it is saved to the disk, the data will always be up-to-date. Also, since the bitmap image can be displayed in different ratios in a scrolled window, we must map the mouse position from the coordinate system of the client window to the coordinate system of the bitmap image (1:1 ratio) in order to update the new pixel. Still, we also need to support cursor shape changing: when the mouse is within the bitmap area, it is more desirable to change the shape of the cursor to represent the current drawing tool.

New Tool Bar

A new tool bar is implemented in the sample that allows the user to select drawing tool. The ID of the tool bar is IDR_DRAWTOOLBAR, and there are two buttons included in the tool bar: ID_BUTTON_PEN and ID_BUTTON_LINE. The two tools can be used for drawing dot and line respectively.

A new variable m_nCurrentTool is declared in class CGDIDoc. It will be used to indicate the current drawing tool. In the sample, WM_COMMAND and UPDATE_COMMAND_UI messages for both ID_BUTTON_PEN and ID_BUTTON_LINE are handled in the following two functions respectively (New drawing tools added in the following sections will also be handled here):

```
void CGDIDoc::OnDrawTool(UINT nID)

{

m_nCurrentTool=nID-ID_BUTTON_PEN;

}

void CGDIDoc::OnUpdateDrawTool(CCmdUI *pCmdUI)

{

pCmdUI->SetCheck((UINT)m_nCurrentTool+ID_BUTTON_PEN == pCmdUI->m_nID);

}
```

With this implementation, at any time, only one tool can be selected. The currently selected tool is indicated by variable CGDIDoc::m_nCurrentTool.

New Functions

The implementation of drawing is complex. We must handle different mouse events, do the coordinates conversion, update the bitmap image according to mouse activity and current drawing tool, and update the client window. It is important to break this whole procedure down into small modules, so the entire drawing task can be implemented by calling just several module functions.

It is obvious that both dot drawing and line drawing should be implemented by handling three mouse related messages: WM_LBUTTONDOWN, WM_RBUTTONDOWN and WM_MOUSEMOVE. There is one thing that must be done before doing any dot or line drawing: converting the current mouse position from the coordinate system of the client window to the coordinate system of the bitmap image (The current ratio and scrolled position must also be taken into consideration). For dot drawing, we need a function that can draw a dot on the bitmap using current foreground color. This function will also be called for line drawing because after the left button is pressed and the mouse has not been moved, we need to draw a dot first. Also, we need a function that can draw a straight line on the bitmap image using the current foreground color if the starting and ending points are known.

There are some concerns with the line drawing. When the user clicks the left button, we need to draw a dot at this position and set the beginning point of the line to it. As the user moves the mouse (with left button held down), we should draw temporary lines until the left button is released. Before the button is released, every time the mouse is moved, we need to erase the previous line and draw a new one. Although this can be easily implemented by using XOR drawing mode, it is not the only solution. An alternate way is to back up the current bitmap image before drawing any temporary line. If we want to erase the temporary drawings, we can just restore the bitmap image backed up before.

In the sample application of this section, several new functions are added to implement dot and line drawings. These functions are listed as follows:

```
CPoint CGDIView::NormalizePtPosition(CPoint pt);
```

The parameter of this function is the mouse cursor position that is measured in the coordinate system of the client window. It will be normalized to the coordinate system of the bitmap image. If the current ratio is greater than 1, the position will be divided by the current ratio. If any of the scroll bars is scrolled, the scrolled position will also be deducted.

void CGDIView::DrawPoint(CPoint pt);

This function draws a dot on the bitmap with current foreground color, which is stored in the document. The input parameter must be a normalized point.

void CGDIView::DrawLine(CPoint ptStart, CPoint ptEnd);

This function draws a line on the bitmap from point ptStart to ptend using the current foreground color, which is stored in the document. The input parameters must be normalized points.

void CGDIView::BackupCurrentBmp();

For the purpose of backing up the current bitmap, a new CBitmap type variable m_bmpBackup is declared in class CGDIView. When function CGDIView::BackupCurrentBmp() is called, we create a new bitmap and attaches it to m_bmpBackup then initialize the bitmap with the current bitmap image (CGDIView:: m_bmpDraw).

void CGDIView::ResumeBackupBmp();

This function does the opposite of the previous function, it copies the bitmap stored in CGDIView:: m_bmpBackup to CGDIView::m_bmpDraw.

With the above new functions, we are able to implement dot and line drawing. To implement interactive line drawing, another new variable m_ptMouseDown is declared in class CGDIView. This variable is used to record the position of mouse cursor when its left button is being pressed down. As the mouse moves or the left button is released, we can use it along with the new mouse position to draw a straight line. The following is the implementation of WM_LBUTTONDOWN message handler:

(Code omitted)

After left button of the mouse is pressed down, we must set window capture in order to receive mouse messages even when the cursor is not within the client window. The window capture is released when the left button is released. If the current drawing object is dot, we need to call function CGDIView:: DrawPoint(...) to draw the dot at the current mouse position; if the current drawing object is line, we need to first backup the current bitmap then draw a dot at the current mouse position.

For WM_MOUSEMOVE message, first we must check if the left button is being held down. If so, we can further proceed to implement drawing. For dot drawing, we need to draw a new dot at the current mouse position by calling function CGDIView::DrawPoint(...); for line drawing, we need to first erase the old drawings by copying the backup bitmap to CGDIView::m_bmpDraw, then draw a new line:

(Code omitted)

The implementation of WM_LBUTTONUP message handler is almost the same with that of WM_MOUSEMOVE message handler: for dot drawing, a new dot is drawn at the current mouse position by calling function CGDIView::DrawPoint(...). For line drawing, the backup bitmap is first resumed to CGDIView::m_bmpdraw. Then a new line is drawn between points represented by CGDIView::m_ptMouseDown and current mouse position.

Mouse Cursor

It is desirable to change the shape of mouse cursor when it is within the bitmap image. We can either choose a standard mouse cursor or design our own cursor. A standard cursor can be loaded by calling function CWinApp::LoadStandardCursor(...). There are many standard cursors that can be used in the application, which include beam cursor (IDC_IBEAM), cross cursor (IDC_CROSS), etc. The mouse cursor can be changed by handling WM_SETCURSOR message. In this message handler, we can call ::SetCursor(...) to change the current cursor shape if we do not want the default arrow cursor.

We need another function to judge if the current mouse cursor is within the bitmap image contained in the client window. In the sample, function CGDIView::MouseWithinBitmap() is added for this purpose. The current image ratio and scrolled positions are all taken into consideration when doing the calculation. The following is the implementation of this function:

(Code omitted)

First we retrieve the current image ratio, horizontal and vertical scrolled positions of the client window. Then function ::GetCursorPos(...) is called to obtain the current position of mouse cursor. Because the returned value of this function (a POINT type value) is measured in the coordinate system of the desktop window (whole screen), we need to convert it to the coordinate system of the client window before judging if the cursor is within the bitmap image. Next, the image rectangle is stored in variable rectBmp, and function CRect::PtInRect(...) is called to make the judgment.

This function is called in WM_SETCURSOR message handler. The following is the implementation of the corresponding function:

(Code omitted)

If the cursor is within the bitmap image and is over neither the horizontal scroll bar nor the vertical scroll bar, we set the cursor to IDC_CROSS (a standard cursor).

Otherwise by calling the default implementation of function OnSetCursor(...), the cursor will be set to the default arrow cursor.

11.4 Tracker

Sample 11.4\GDI is based on sample 11.3\GDI.

Tracker can be implemented to let the user select a rectangular area very easily, and is widely used in applications supporting OLE to provide a graphical interface that lets the user interact with OLE client items. When implementing a tracker, we can select different styles. This can let the tracker be displayed with a variety of visual effects such as hatched borders, resize handles, etc.

Tracker can also be applied to any normal application. In a graphic editor, tracker can be used to select a rectangular region, move and drop it anywhere within the image. It can also be used to indicate the selected rectangular area when we implement cut, copy and paste commands (Figure 11-4).

Implementing Tracker

Tracker is supported by MFC class CRectTracker. To enable a rectangular tracker, we need to first use this class to declare a variable, then set its style. When the window owns the tracker is being painted, we need to call a member function of CRectTracker to draw the tracker.

We can set the tracker to different styles. The style of tracker is specified by variable CRectTracker:: m_nStyle. The following values are defined in class CRectTracker and can be used to specify the border styles of a tracker: CRectTracker::solidLine, CRectTracker::dottedLine, CRectTracker:: hatchedBorder. The following values can also be assigned to CRectTracker::m_nStyle to specify how the tracker can be resized: CRectTracker::resizeInside, CRectTracker::resizeOutside. Finally, CRectTracker::hatchInside can be assigned to CRectTracker::m_nStyle to specify if the hatched border should be drawn outside or inside the rectangle. All the above styles can be combined together using bit-wise OR operation.

Moving and Resizing Tracker

The tracker's position and dimension are stored in variable CRectTracker::m_rect. We can modify it at any time to move the tracker or resize it. If this variable specifies a valid rectangle, we can draw the tracker by calling function CRectTracker::Draw(...) and display it in a window.

To let user resize the tracker through clicking and dragging tracker's resizing buttons (See Figure 11-4), we need to handle WM_LBUTTONDOWN message and call function CRectTracker::HitTest(...) to find out if the current mouse cursor hits any portion of the tracker. The following is the format of this function:

```
int CRectTracker::HitTest(CPoint point);
```

The following is a list of values that can be returned from this function along their meanings:

(Table omitted)

If mouse cursor hits any of the resize buttons, we can call CRectTracker::Track() to track the mouse moving activities from now on until the left button is released. With this function, there is no need for us to handle other two messages WM_MOUSEMOVE and WM_LBUTTONUP, because once it is called, the function will not return until the left button is released. Of course, we can also write code to implement right button tracking. When we call function CRectTrack::Track(), the tracker's owner window should not set window capture, otherwise the mouse message will not be routed to the tracker.

Customizing Cursor Shape

To let the mouse cursor shape change automatically when it is over tracker's region, we need to call function CRectTracker::SetCursor(...) inside window's CWnd::OnSetCursor(...) function (in the window that contains the tracker). If the function returns TRUE, it means that the cursor shape has already been customized (The cursor is over tracker's region). In this case we can exit and return a TRUE value. Otherwise we must call function CWnd::OnSetCursor(...) to let the cursor's shape be set to the default one.

New Tool

In the new sample application, a new tool "Rectangular Selection" is implemented in tool bar IDR_DRAWTOOLBAR (Figure 11-5). If it is selected as the current tool, the user can drag the mouse to create a tracker over the image, resize or move it to change the selection. The cursor will be automatically changed if the mouse cursor is within the tracker's region.

First, a new command ID_BUTTON_RECSEL is added to IDR_DRAWTOOL tool bar. Each time a new tool command is added to the tool bar, we must make sure that the IDs of all the commands contained in the tool bar are consecutive. Otherwise the macros ON_COMMAND_RANGE and ON_UPDATE_COMMAND_UI_RANGE will not work correctly. In the sample, two macros TOOL_HEAD_ID and TOOL_TAIL_ID are defined, and they represent the first and last IDs of the commands contained in the drawing tool bar. We use the above two macros to do the message mapping. By doing this, if we add a new tool next time, all we need to do is redefining the macros.

In class CGDIView, a CRectTracker type variable m_trackerSel is declared to implement the tracker. The tracker's styles are initialized in the constructor as follows:

(Code omitted)

The tracker's border is formed by dotted line and the resize buttons are located outside the rectangle.

The tracker is drawn in function CGDIView::OnDraw(...) if the tracker rectangle is not empty:

(Code omitted)

As we will see, the tracker's size and position will be recorded in the zoomed bitmap image's coordinate system. This is for the convenience of coordinate conversion. Since the DC will draw the tracker in client window's coordinate system, we must add some offset to the tracker rectangle before it is drawn. Also we need to resume rectangle's original state after the drawing is completed.

Function CGDIView::OnSetCursor(...) is modified as follows so that the cursor will be automatically changed if it is over the tracker region:

(Code omitted)

Here we first check if the cursor can be set automatically. If CRectTracker::SetCursor(...) returns TRUE, we can exit and return a TRUE value (If this function returns TRUE, it means currently the mouse cursor is within the tracker region, and the cursor is customized by the tracker). If not, we check if the mouse is over the bitmap image, if so, the cursor's shape is set to IDC_CROSS and the function exits (We need to return TRUE every time the cursor has been customized). If all these fail, we need to call function CWnd::SetCursor(...) to set the cursor to the default shape.

If Mouse Clicking Doesn't Hit the Tracker

We must implement a way of letting the user create tracker interactively. Like line drawing implementations, we need to handle WM_LBUTTONDOWN, WM_RBUTTONDOWN and WM_MOUSEMOVE messages in order to let the user create tracker by mouse clicking. This procedure is similar to rectangle drawing: when left button is pressed down, we need to record the current mouse position as the starting point; as the mouse is dragged around, we draw a series of temporary rectangles; when the left button is released, we use the current mouse position as the ending point and use it along with the starting point to draw the tracker.

For the tracker, there are two possibilities when the mouse button is pressed down. If currently there is an existing tracker and the mouse hits the tracker, we should let the tracker be moved or resized instead of creating a new tracker. If there is no tracker currently implemented or the mouse did not hit the existing tracker, we should start creating a new tracker.

This situation can be judged by calling function CTracker::HitTest(...), whose input parameter should be set to the current position of mouse cursor that is measured in the client window's coordinate system. If the function returns CRectTracker::hitNothing, either there is no existing tracker or the mouse didn't hit any portion of the tracker. In the sample, this situation is handled as follows:

(Code omitted)

We record the starting position in the upper-left point of the tracker rectangle. As the mouse moves, the rectangle's bottom-right point is updated with the current mouse position, and temporary rectangles are drawn and erased before the tracker is finally fixed.

Temporary rectangles are drawn by calling function CDC::DrawFocusRect(...). Because this function uses XOR drawing mode, it is easy to erase the previous rectangle by simply calling the function twice.

When the left button is released, we erase the previous temporary rectangle if necessary, update the tracker rectangle, and call function CWnd::InValidate() to let the tracker be updated (along with the client window).

Because we must keep track of mouse cursor position after its left button is pressed down, the window capture must be set when a new tracker is being created. Since we share the code implemented for dot and line drawing here, there is no need to add extra code to set window capture for the client window here.

If Mouse Clicking Hits the Tracker

If mouse clicking hits the tracker (any of the resize buttons, or the middle of the tracker), we must implement tracking to let the user resize or move the existing tracker. This can be easily implemented by calling function CRectTracker::Track():

(Code omitted)

Because the window capture is set when a new tracker is being created (also, when a dot or a line is being drawn), we must first release the capture before trackering mouse movement (Otherwise the tracker will not be able to receive messages related to mouse moving events). There is no need for us to handle WM_MOUSEMOVE and WM_LBUTTONUP messages here because after function CRectTracker::Track() is called, mouse moving events will all be routed to the tracker. After this function exits, variable CRectTracker::m_rect will be automatically updated to represent the new size and position of the tracker. So after calling this function, we can update the client window directly to redraw the tracker.

11.5 Moving the Selected Image

Sample 11.5\GDI is based on sample 11.4\GDI. It is implemented with a new feature: when a portion of the bitmap is selected by the tracker, the user can move the selected image by dragging the tracker to another place; if the user resizes the tracker, the selected image will also be stretched.

Normalizing Tracker

Since the tracker rectangle is recorded in the zoomed image's coordinate system, we must first convert it back to the original bitmap's own coordinate system (the image with 1:1 ratio) in order to find out which part of the image is being selected. In the sample, function CGDIView::NormalizeTrackerRect(...) is added for this purpose. In this function, the current image ratio is retrieved from the document, and the four points of the tracker rectangle is divided by this ratio. The tracker can be created in two different ways. For example, the user may click and hold the left mouse button and drag it right-and-downward; also, the mouse may be dragged up-and-leftward. For the first situation, a normal rectangle will be formed, in which case member CRectTracker.m_rect.left is always less than member CRectTracker.m_rect.right, and CRectTracker.m_rect.top is less than CRectTracker::m_rect.bottom. However, in the second situation, CRectTracker.m_rect.left and CRectTracker.m_rect.top are all grater than their corresponding variables. So before using variable CRectTracker::m_rect, we must normalize the rectangle.

We can call function CRect::NormalizeRect() to normalize a rectangle implemented by class CRect. In function CGDIView::NormailizeTrackerRect(...), before the four points of the tracker rectangle are divided by the ratio, this function is called to first normalize the rectangle.

Moving and Resizing the Selected Image

The easiest way to move or resize the selected image is to back up the image that is under the current tracker, as the user resizes the tracker, copy the backup image back and let it fit within the new tracker. When doing this copy, we can call function CDC::StretchBlt(...) to resize the copied image.

The procedure of backing up the selected image is similar to backing up the whole image as we did in function CGDIView::BackupCurrentBmp(). The size of the backup image must be the same with the size of tracker rectangle. One thing we need to pay attention to is that since we allow the user to resize and move the tracker freely, it is possible that after moving or resizing, some part of the tracker is outside the image. In this case, we must recalculate the backup area so that only the intersection of the tracker and the image will be copied. The non-intersection part should be left blank. For this purpose, before copying the selected image, we need to fill all the backup bitmap with the current background color.

Variable CGDIView::m_bmpSelBackup is declared in the application to backup the selected area.

Besides backing up the selected area, we need another function to copy the backup image back to the original bitmap. In the sample, function CGDIView::StretchCopySelection() is added for this purpose. Within it, function CDC::StretchBlt(...) is called to copy the selection back to the bitmap, the position and dimension of the current tracker rectangle are used to specify the target image. Whenever we want to move or resize the selected image, we can first move or resize the tracker rectangle then call this function.

When Left Button is Up

We need to back up the selected image after the tracker is created. This corresponds to the left button release event (Also, the current drawing tool must be "Rectangular Selection"). Besides the selected region, we also need to backup the whole bitmap. This is because when the user moves the tracker, we must copy the backup image back to the original bitmap (before the selection is moved) instead of the current one. Otherwise as the selection is moved around, it will leave a trail on the image. So within WM_LBUTTONUP message handler, both CGDIView::BackupCurrentBmp() and CGDIView::BackupSelection() are called to implement the backup:

(Code omitted)

The selection should be copied back within WM_LBUTTONDOWN message handler after function CRectTracker::Track() is called. By using this function, the tracker rectangle can be automatically updated when the mouse button is released. In the sample, functions CGDIView::ResumeBackupBmp() and CGDIView::StretchCopySelection() are called to copy the selected image back to the original bitmap:

(Code omitted)

With the above implementations, we are able to select the image using "Rectangular Selection" tool, then move or resize it.

11.6 Region

Before implementing new drawing tools, we need to introduce some new concepts.

In this and following sections, we will discuss region and path, both of which are GDI objects. The implementation of simple "Paint" will be resumed in section 11.8.

Basics

Region is another very useful GDI object. We can use region to confine DC drawings within a specified area no matter where the DC actually outputs. After a specified region is specified, all DC's outputs (dot and line drawing, brush fill, bitmap copy) will be confined within the region area. By using the region, it is very easy for us to draw objects with irregular shapes.

A region can have any type of shapes. It can be rectangular, elliptical, polygonal or any irregular closed shape. Moreover, a region can be created by combining two existing regions, the result can be the union, the intersection or difference of the two regions. With these operations, we can create regions with a wide variety of shapes.

Region Creation

In MFC, region is implemented by class CRgn. Like other GID objects, this class is derived from CGDIObject, which means a valid region must be associated with a valid handle. Standard regions can be created by calling one of the following functions:

```
BOOL CRgn::CreateRectRgn(int x1, int y1, int x2, int y2);

BOOL CRgn::CreateRectRgnIndirect(LPCRECT lpRect);

BOOL CRgn::CreateEllipticRgn(int x1, int y1, int x2, int y2);

BOOL CRgn::CreateEllipticRgnIndirect(LPCRECT lpRect);

BOOL CRgn::CreatePolygonRgn(LPPOINT lpPoints, int nCount, int nMode);

BOOL CRgn::CreatePolyPolygonRgn

(

LPPOINT lpPoints, LPINT lpPolyCounts, int nCount, int nPolyFillMode

);

BOOL CRgn::CreateRoundRectRgn(int x1, int y1, int x2, int y2, int x3, int y3);
```

As we can see, a region may have different shapes: rectangular, elliptical, polygonal. We can even create a region that is composed of a series of polygons by calling function CRgn:: CreatePolyPolygonRgn(...). As we will see later, a region can also have an irregular shape.

Existing regions can be combined together to form a new region. The combining operation mode can be logical AND, OR, XOR, the union or the difference of the two regions. The function that can be used to combine two existing regions is:

```
int CRgn::CombineRgn(CRgn* pRgn1, CRgn* pRgn2, int nCombineMode);
```

Please note that CRgn type pointers passed to this function must point to region objects that have been initialized by one of the functions mentioned above, or by other indirect region creating functions. Parameter nCombineMode can be set to any of RGN_AND, RGN_COPY, RGN_DIFF, RGN_OR and RGN_XOR, which specify how to combine the two regions.

Using Region

Like any other GDI object, before using the region, we must first select it into the DC. The difference between using region and other GDI objects is that we need to call function CDC::SelectClipRgn(...) to select the region instead of calling function CDC::SelectObject(...).

Function CDC::SelectClipRgn(...) has two versions:

```
virtual int CRgn::SelectClipRgn(CRgn* pRgn);

int CRgn::SelectClipRgn(CRgn* pRgn, int nMode);
```

For the second version of this function, parameter nMode specifies how to combine the new region with the region being currently selected by the DC. Again, it can be set to any of the following flags: RGN_AND, RGN_COPY, RGN_DIFF, RGN_OR and RGN_XOR.

After using the region, we must select it out of the DC before deleting it. To select a region out of the DC, we can call function CDC::SelectClipRgn(...) and pass a NULL pointer to it. For example, the following statement selects the region out of DC (pointed by pDC):

```
pDC->SelectClipRgn(NULL);
```

A region can be deleted by calling function CGDIObject::DeleteObject(). Also, the destructor of CRgn will call this function automatically, so usually there is no need to delete the region unless we want to reinitialize it.

Sample

Sample 11.6\GDI is a standard SDI application generated by Application Wizard. In the sample, two variables are declared in class CGDIView: CGDIView::m_rgnRect and CGDIView::m_rgnEllipse. They will be used to create two regions, one is rectangular and one is elliptical. The two regions will be combined together to create a new region that is the difference of the two. We will select this region into the client window's DC, and output text to the whole window. As we will see, only the area that is within the region will have the text output.

The regions are created in the constructor of class CGDIView:

(Code omitted)

The final region will look like the shaded area shown in Figure 11-6.

In function CGDIView::OnDraw(...), string "Clip Region" is output repeatedly until all the client window is covered by this text:

(Code omitted)

The output result is shown in Figure 11-7.

11.7 Path

Basics

Path is another type of powerful GDI object that can be used together with device context. A path can be seen as a closed figure that is formed by drawing trails. For example, Figure 11-8 shows a path that is made up of alternative lines and curves.

A path can record almost all types of outputs to the device context. Like other GDI objects, it must be first selected into a DC before being used. However, there is no class such as CPath that lets us declare a path type variable. Therefore, we can not select a path into DC by calling function CDC:: SelectObject(...). To use path, we must call function CDC::BeginPath() to start path recording and call CDC::EndPath() to end it.

Between the above two functions, we can call any of the drawing functions such as CDC::LineTo(...), CDC::Rectangle(...), and CDC::TextOut(...). The trace of the output will be recorded in the path and can be rendered later. When rendering the recorded path, we can either draw only the outline of the path using the selected pen or fill the interior with the selected brush, or we can do both.

The following functions can be used to implement these path drawing:

```
BOOL CDC::StrokePath();

BOOL CDC::FillPath();

BOOL CDC::StrokeAndFillPath();
```

Function CDC::StrokePath() will render a specific path using the currently selected pen. This will draw outline of the closed figure. Function CDC::FillPath() will close any open figures in the path and fill its interior using the currently selected brush. After the interior is filled, the path will be discarded from the device context. Function CDC::StrokeAndFillPath() implements both: it will stroke the outline of the path and fill the interior.

Please note that the last function can not be replaced by calling the first two functions consecutively. After function CDC::StrokePath() is called, the path will be discarded, so further calling CDC::FillPath() will not have any effect.

Path & Region

Region can also be created from path. One straightforward method is to call function CDC:: SelectClipPath(...) to create a region from the current path then select it into the DC. If we create region this way, there is no need for us to use CRgn type variable. Also, we can explicitly create a region from the path by calling function CRgn::CreateFromPath(...). The parameter we need to pass to this function is a pointer to the DC that contains the path. This is a very powerful method: by creating an irregular-shaped path, we can use it to create a region that can be used to confine the output of DC.

Sample 11.7-1\GDI

Sample 11.7-1\GDI demonstrates path implementation. It is a standard SDI application generated by Application Wizard. No new variable is declared. In function CGDIView::OnDraw(...), we begin path recording and output four characters 'P', 'a', 't', 'h' to the client window. Then we stroke the outlines of the four characters and fill the path with a hatched brush.

In the sample, the font used to output the text is "Times New Roman", and its height is 400. The brush used to fill the interior of the path is a hatched brush whose pattern is cross hatch at 45 degrees. Between function CDC::BeginPath() and CDC::EndPath(), there is only one statement that calls function CDC::TextOut(...) to output the four characters. Please note that while path recording is undergoing, no output will be generated to the target device. So this will not output anything to the client window. Finally function CDC::StrokeAndFillPath() is called to stroke the text outline and fill the path's interior using hatched brush. The following is the implementation of function CGDIView::OnDraw(...):

(Code omitted)

Figure 11-9 shows the output result.

(Figure 11-9 omitted)

Obtaining Path

A path can be retrieved by calling function CDC::GetPath(...). This function has three parameters, first two of which are pointers that will be used to receive path data, and the final parameter specifies how many points are included in the path:

```
int CDC::GetPath(LPPOINT lpPoints, LPBYTE lpTypes, int nCount);
```

A path is formed by a series of points and different type of curves. The points are stored in the buffers pointed by lpPoints, and curve types are stored in the buffers pointed by lpTypes, which can be any of the following: PT_MOVETO, PT_LINETO, PT_BEZIERTO or PT_CLOSEFIGURE.

To receive path information, we must first allocate enough buffers for storing point and type information. Since the buffer size depends on the number of points included in the path, when calling function CDC::GetPath(...), we can first pass NULL pointer to lpPoints and lpTypes parameters and 0 to nCount. This will cause the function to return the number of points included in the path. After enough buffers are allocated for storing both point and type information, we can call CDC::GetPath(...) again to get the path.

Since path stores drawing trace in the form of vectors, we can change the shape of a path by moving the control points without losing quality of the image. We can change the positions of the points using certain algorithm. For example, if we multiply the vertical coordinate of all points with a constant factor, the result will be an enlarged image scaled from the original path.

Sample 11.7-2\GDI

Sample 11.7-2\GDI demonstrates how to obtain and modify a path. It is based on sample 11.7-1\GDI. In the sample, after text "Path" is output to the device context (which is recorded into path), function CDC::GetPath(...) is called to retrieve the points and curve types into the allocated buffers. Then, we change the y coordinates of all points by linearly moving them upward. The following code fragment of function CGDIView::OnDraw(...) shows how the buffers are allocated and path is obtained:

(Code omitted)

The total number of points is stored in variable nNumPts. Because we need to use a POINT type array to receive the points, the buffer size for storing points is calculated as follows:

(number of points) * (size of structure POINT)

Since a curve type uses only one byte, the buffer size for storing curve types is nNumPts.

The following portion of function CGDIView::OnDraw(...) shows how the points are moved:

(Code omitted)

The following formulae is used to move a point upward:

new vertical coordinate =

(original vertical coordinate) (

Then a new path is created from new points:

(Code omitted)

For a different type of curves, we call the corresponding CDC member function. Please note that since Bezier curve uses three points, after drawing the Bezier curve, we need to advance the loop index by 2.

After this, function CDC::StrokeAndFillPath(...) is called to stroke the path's outline and fill the interior. Figure 11-10 shows the effect.

11.8 Freeform Selection

Now its time to go back to our simple graphic editor. If we are familiar with the actual "Paint" application, we know that it allows the user to select an irregular part of the image, move it, and resize it. This freeform selection tool can be implemented by using path and region together.

Implementation

The freeform selection tool has a lot in common with the rectangular selection tool:

both need to respond to mouse events for specifying a selected region; both need to implement a tracker to allow the user to move and resize the selected image; both need to back up the selected portion of the bitmap. The only difference is that for the freeform selection, the selected region may not be rectangular.

However, we can still make use of functions CGDIView::BackupSelection() and CGDIView:: StretchCopySelection(). Since we can store the irregular selection in a CRgn type variable, it doesn't matter if the backup area is rectangular or not. If we select the region into the DC before copying the rectangular backup image, only the pixels within the region will be copied.

The region can be created from path. As the user presses the left button, if the current drawing tool is freeform selection, we will start path recording. After the left button is released, the path recording will be stopped and the irregular region will be created from the path. To allow the user to resize or move this region, we must enable tracker and backup the selected area at this point.

Although the selection can be an irregular area, the tracker must be rectangular. We can set the tracker rectangle to the bounding rectangle of the region, which can be obtained by calling function CRgn:: GetRgnBox(...). If the user changes the tracker, we must copy this region to a new position and resize it to let the region fit within the rectangle which is indicated by the new tracker.

Scaling Region

There is a problem here: the recorded region was created when the user first made the selection. After the tracker is moved or resized, we must offset and scale the original region to let it fit within the new tracker rectangle before selecting it into the target DC. Please note that in order to confine DC drawings within a region, we must use the target DC to select the region. Thus the problem is how to offset and scale the region to let it fit within another rectangle without losing its original shape.

Unlike path, region is not recorded using vectors. Instead, it is made up of a number of rectangles. We need to resize every rectangle in order to resize the whole region (Figure 11-11).

Just like we can retrieve path data by calling function CDC::GetPath(...), for region, we can also retrieve its data by calling function CRgn::GetRegionData(...). This function has two parameters:

```
int CRgn::GetRegionData(LPRGNDATA lpRgnData, int nCount);
```

Here lpRgnData is a pointer that will be used to receive the region data, and nCount specifies the size of buffers that are pointed by lpRgnData. Of course it is not possible to know the size of region data before we know its detail. To find out the necessary buffer size, we can first pass NULL to lpRgnData parameter and 0 to nCount parameter, which will cause the function to return the size needed for storing all the region data. Using this size, we can allocate enough buffers and call the function again to actually retrieve the region data.

Region data is stored in a RGNDATA type structure:

```
typedef struct _RGNDATA {

RGNDATAHEADER rdh;

char Buffer[1];

} RGNDATA;
```

It contains two members, rdh is of RGNDATAHEADER type, which is the region data header. Member Buffer is the first element of a char type array that contains a series of rectangles. The information of the region is stored in its header structure:

```
typedef struct _RGNDATAHEADER {

DWORD dwSize;

DWORD iType;

DWORD nCount;

DWORD nRgnSize;

RECT rcBound;

} RGNDATAHEADER;
```

Member dwSize specifies the size of this header, and the iType specifies the region type, whose value must be RDH_RECTANGLES, which means that the region is made up of a number of rectangles. Member nCount specifies the number of rectangles contained in this region, and rcBound specifies the bounding rectangle. In order to scale the region, we need to implement a loop, and scale the corresponding rectangle whose data is contained in the data buffer within each loop.

If we simply want to offset the region, there is a member function of CRgn that can be used to offset the whole region just like we can offset a rectangle:

```
int CDC::OffsetClipRgn(int x, int y);
int CDC::OffsetClipRgn(SIZE size);
```

The region must be selected into a DC in order to call the above member functions.

New Tool

Sample 11.8\GDI is based on sample 11.6\GDI that implements freeform selection. In the sample, first a new command is added to toolbar IDR_DRAWTOOLBAR, whose ID is ID_BUTTON_FREESEL (Figure 11-12). Macro TOOL_HEAD_ID is redefined (it represents ID_BUTTON_FREESEL now) to allow this new tool be automatically considered for message mapping. In order to use old message handlers, the ID of the freeform selection tool is set to have the following relationship with other IDs:

```
ID_BUTTON_FREESEL = ID_BUTTON_RECTSEL+1
```

In class CGDIView, like rectangular selection command, we need to handle WM_LBUTTONDOWN, WM_LBUTTONUP and WM_MOUSEMOVE messages to implement freeform selection tool. In the sample, a new case is added to the switch statement within each message handler.

Drawing rectangular outline and freeform outline is different. For rectangular selection, whenever mouse is moving, we erase the old rectangular outline and draw a new one using old mouse position (recorded when mouse left button was first pressed down) and current mouse position. For freeform selection, we do not need to erase the old outline: each time the mouse is moving, we just draw a line between the previous mouse position (recorded when WM_MOUSEMOVE was received last time) and the current mouse position. For this purpose, a new variable CGDIView::m_ptPrevious is declared, which will be used to record the previous mouse position. The current mouse position will be passed to the mouse message handler as a parameter. Since the outline is drawn in the client window, it needs to have a same ratio with the current displayed bitmap. However, to make it easy for copying and moving the selection, the path must be recorded with a ratio of 1:1 no matter what the current image ratio is. So for each mouse move, we must do the following two things: draw a line between the old mouse position and the current mouse position in the client window, then draw the same thing on the memory bitmap (CGDIView:: m_bmpDraw) for path recording.

To simplify outline drawing, a CPen type variable is declared in class CGDIView and is initialized in the constructor as follows:

```
......

m_penDot.CreatePen(PS_DOT, 1, RGB(0, 0, 0));
```

......

We will use dotted line to draw the outline of irregular selection while the mouse is moving.

To record the selected region, a CRgn type variable is declared in class CGDIView. The following portion shows how freeform selection is handled after WM_LBUTTONDOWN message is received:

(Code omitted)

For WM_MOUSEMOVE message, we need to draw the freeform outline in the client window and do the same thing to CGDIView::m_bmpDraw. Note since the current image ratio may not be 1:1, we must use normalized points when recording path:

(Code omitted)

For WM_LBUTTONUP message, we need to draw the last segment of outline, close the figure, and end the path. Then we need to create the region from path, and set the dimension of the tracker to the dimension of the bounding box of the selected region. Again we must consider ratio conversion here: since the path is recorded with a ratio of 1:1, we must scale its bounding box to the current image ratio. In the sample, function CGDIView::UnnormalizeTrackerRect(...) is added to scale a normalized rectangle to the current image ratio. The rest thing needs to be done is the same with that of rectangular selection: we need to back up the whole image as well as the selected area:

(Code omitted)

Resizing and Moving the Freeform Selection

Resizing and moving the freeform selection is almost the same with that of rectangular selection, except that before copying the selected area, we must offset or resize the region and select it into the target DC. The following portion of function CGDIView::OnLButtonDown(...) shows how the region is resized and selected into the target memory DC, and how the selected image is copied back:

(Code omitted)

With the above implementation, the application will support freeform selection.

11.9 Cut, Copy and Paste

Sample 11.9\GDI is based on sample 11.8\GDI.

Clipboard DIB Format

Like what we did for one line editor in chapter 9, here we will also implement copy, cut and paste commands for our simple graphic editor. As we know, if we want to put data to the clipboard, first we must open and empty it, then we must put data with standard format to the clipboard so that it can be shared by other applications. There are a lot of standard clipboard formats. Also, we can define and register our own clipboard formats.

The format for device independent bitmaps is CF_DIB, we need to prepare image data with standard DIB format. Because there exists many different DIB formats, it is desirable if the application supports format conversion for cut, copy and paste commands. For example, if the image being edited by our application is 16-color format, and the DIB contained in the clipboard is 256-color format, it is not convenient if we can not paste the data just because of the difference on data format.

Just for the demonstration purpose, in our sample we will implement DIB copy, cut and paste for only 256-color format. With the knowledge of previous chapters, it is easy to extend the application to let it support multiple image formats.

Preparing DIB Data

We already know how to prepare DIB data: allocate enough buffers, stuff them with bitmap information header, color table, and DIB bit values. Because the image being edited (the whole bitmap) and the image that will be put to the clipboard (bitmap portion under the tracker) has the same bitmap information header (except for members biWidth, biHeight and biImageSize) and color table, but different dimension and bit values, we can first copy the current bitmap information header and color table into the buffers, then we need to obtain the image bit values from the area that is selected by the tracker.

Both cut and copy commands can share one function to prepare DIB data and put it to the clipboard. For cut command, besides image copying, we also need to fill all the selected area with the current background color.

We need to calculate the values of the following members for the new bitmap information header: biWidth, biHeight and biImageSize. Here the dimension of the new image (biWidth, biHeight) can be decided from the size of the tracker, and the image size can be calculated from member biBitCout along with the new image dimension. In the sample, function CGDIDoc::CreateCopyCutDIB()creates a DIB that is exactly the same with the image contained in the selected area. The following portion of this function shows how the buffers are allocated and how the bitmap information header is created:

(Code omitted)

The buffer size of new DIB data is calculated and stored in variable dwDIBSize. Here rect stores the normalized dimension of the current tracker. The current image's bitmap information header is stored in variable bi. After copying it into the new buffers, we change members biWidth, biHeight and biImageSize to new values.

Then, we need to copy the current color table to the newly allocated buffers. Although we could retrieve the palette from the original DIB data, it may not be up-to-date because the user may change any entry of the palette by double clicking on the color bar. In the sample, function CPalette::GetPaletteEntries(...) is called to retrieve the current palette, and is used to create the new image:

(Code omitted)

Because there is no restriction on the dimension of the tracker, the user can actually select an area that some portion of it is outside the current image. In order to copy only the valid image, we need to adjust the rectangle before doing the copy. In the sample, only the intersection between the tracker and the image is copied, and the rest part of the target image will be filled with the current background color. In function CGDIDoc::CreateCopyCutDIB(...), the actual rectangle that will be used to obtain the image bit values from the source bitmap is stored in variable rectSrc, and sizeTgtOffset is used to specify the position where the image will be copied to the target image. This variable is necessary because the tracker's upper-left corner may resides outside the image. The whole image is copied using a loop. Within each loop, one raster line is copied to the target bitmap. In the function, two pointers are used to implement this copy: before copying the actual pixels, lpRowSrc is pointed to the source image buffers and lpRowTgt is pointed to the target image buffers. Their addresses are calculated by adding the bitmap information header size and the color table size to starting addresses of the DIB buffer. Since one raster line must use multiple of 4 bytes, we need to use WIDTHBYTES macro to calculate the actual bytes that are used by one raster line. The following shows how the selected rectangular area of the bitmap is copied to the target bitmap:

(Code omitted)

Cut & Copy

With function CGDIDoc::CreateCopyCutDIB(...), it is much easier to implement cut and copy commands. For copy command, we need to open the clipboard, empty it, set clipboard data, and close the clipboard. For the cut command, we also need to fill the selected area with the current background color. The following shows how the cut command is implemented:

(Code omitted)

Paste

Paste command is the reverse of cut or copy command: we need to obtain DIB data from the clipboard and copy it back to the bitmap image that is being edited. To let the user place pasted image everywhere, we need to implement tracker again to select the pasted image. With this implementation, the user can move or resize the image surrounded by the tracker just like using the rectangular selection tool.

So instead of copying DIB data from the clipboard directly to the bitmap being edited, we can first create and copy it to the backup bitmap image (CGDIView::m_bmpSelBackup), then change the current drawing tool to rectangular selection (if it is not). By doing this, everything will go on as if we just selected a portion of image using the rectangular selection tool.

When the user executes Edit | Paste command, function CGDIDoc::OnEditPaste()will be called. The DIB data in the clipboard will be replicated and passed to function CGDIView::PasteDIB(...). Within the function, a new bitmap will be created using variable m_bmpSelBackup, and the DIB data contained in the clipboard will be copied to it. Please note we must replicate the clipboard data instead of using it directly because the data may be used by other applications later. The following is the implementation of function CGDIDoc::OnEditPaste():

(Code omitted)

The following portion of function CGDIView::PasteDIB(...) shows how to copy the clipboard DIB data to the backup DIB image by calling function ::SetDIBits(...) (the clipboard data is passed through parameter hData):

(Code omitted)

When calling function ::SetDIBits(...), we need to provide the handle of client window (to its first parameter). This is because the image will finally be put to the client window. Also we need to provide the handle of target bitmap image (to the second parameter). The third and fourth parameters of this function specify the first raster line and total number of raster lines in the target bitmap respectively. The fifth parameter is a pointer to the source DIB bit values (stored as an array of bytes). The sixth parameter is a pointer to bitmap header, and final parameter specifis how to use the palette. Since we want the DIB bits to indicate the color table contained in the DIB data, we should choose DIB_RGB_COLORS flag.

After copying the image, we need to enable the tracker, backup the current bitmap, copy the new bitmap to the area specified by tracker rectangle, and set the current drawing tool to rectangular selection. Then, we need to update the client window. The following portion of function CGDIView::PasteDIB(...) shows how these are implemented:

(Code omitted)

Because the rectangular selection button is not actually pressed by the user, we need to generate a WM_COMMAND message in the program and send it to the mainframe window. When doing this, WPARAM parameter of the message is set to the ID of rectangle selection command. This will have the same effect with clicking the rectangular selection button using the mouse.

With the above implementation, we can execute cut, copy and paste commands now. Please note that these commands work correctly only if the currently loaded image is 256-color format. For the other formats, error message may be generated.

11.10 Palette Change & Flickering

Sample 11.10\GDI is based on sample 11.9\GDI.

Problems

Now we have a very basic graphic editor. Before the editor becomes perfect, we still have a lot of things to do: we need to add new tools for drawing curves, rectangles, ellipses, and so on; also we need to support more bitmap formats, for example: 16-color DIB format, 24-bit DIB format. We can even add image processing commands to adjust color balance, brightness and contrast to make it like a commercial graphic editor.

Besides these, there are still two problems remained to be solved. One is that after loading an image with this simple editor, if we switch to another graphic editor and load a colorful image then switch back, the color of the image contained in our editor may change. Another problem is the unpleasant flickering effect when we draw lines with the grid on.

Message WM_PALETTECHANGED

The first problem is caused by the change on the system palette. Because each application has its own logical palette, and the system has only one physical palette, obviously the system palette can not be occupied by only one application all the time. Since the operating system always tries to first satisfy the needs of the application that has the current focus, if we switch to another graphic editor and leave our editor working in the background, most entries of the system palette will be occupied by that application and very few entries are left for our application. In this case, the system palette represents the logical palette of another application rather than ours, so if we still keep the original logical-to-system palette mapping, most colors will not be implemented correctly. This situation remains unchanged until we realize the logical palette again, which will cause the logical palette to be mapped to the system palette.

Under Windows(, there is a message associated with system palette changing. When the system palette is mapped to certain logical palette and this causes its contents to change, a WM_PALETTECHANGED message will be sent out. All the applications that implement logical palette should handle this message and re-map the logical palette to the system palette whenever necessary to avoid color distortion.

In the sample, whenever we draw the image in CGDIView::OnDraw(...), function CDC:: RealizePalette() is always called to update the logical palette. So when receiving message WM_PALETTECHANGED, we can just update the client window to cause the palette to be mapped again.

Usually message WM_PALETTECHANGED is handled in the mainframe window. This is because an application may have several views attached to a single document. By handing this message in the mainframe window, it is relatively easy to update all views. The following is the message handler CMainFarme::OnPaletteChanged(...) that is implemented in the sample for handling the above message:

```
void CMainFrame::OnPaletteChanged(CWnd* pFocusWnd)

{

CFrameWnd::OnPaletteChanged(pFocusWnd);

GetActiveDocument()->UpdateAllViews(NULL);

}
```

This function is quite simple. With the above implementation, the first problem is solved.

Flickering

The second problem will be present only if the grid is on. This is because the grid is drawn directly to the client window: whenever the image needs to be updated, we must first draw the bitmap, and this operation will erase the grid. The user will see a quick flickering effect when the grid appears again. If we keep on updating the client window, this flickering will become very frequent and the user will experience very unpleasant effect.

We already know that one way to get rid of flickering is to prepare everything in the memory and output the final result in one stroke. This is somehow similar to that of drawing bitmap image with transparency (See Chapter 10). To solve the flickering problem, we can prepare a memory bitmap whose size is the same with the zoomed source image, before updating the client window, we can output everything (image + grid) to the memory bitmap, then copy the image from the memory bitmap to the client window.

However, the problem is not that simple. Because the image size can be different from time to time, also the image could be displayed in various zoomed ratio, it is difficult to decide the dimension of the memory bitmap. To avoid creating and destroying memory bitmaps whenever the size (or ratio) of the output image changes, we can create a bitmap with fixed size for painting the client window. If the actual output needs a larger area, we can use a loop to update the whole client area bit by bit: within any loop we can copy a portion of the source image to the memory bitmap, add the grid, then output it to the client window. Because CDC::BitBlt(...) is a relatively fast function, and the bitmap drawing will be implemented directly by the hardware, calling this function several times will not cause obvious delay.

In the sample application, some new variables are added for this purpose. We know that in order to prepare a memory bitmap, we also need to prepare a memory DC that will be used to select the bitmap. In class CGDIView, a CBitmap type variable m_bmpBKStore and a CDC type variable m_dcBKMem are declared. Also, since the DC will select the bitmap and logical palette, two other pointers CGDIView::m_pBmpBKOld and CGDIView::m_pPalBKOld are also declared. The two pointers are initialized to NULL in the constructor, and the memory bitmap CGDIView::m_bmpBKStore is created in function CGDIView::OnInitialUpdate()when an image is first loaded. The reverse procedure is done in function CDC::OnDestroy(), where the memory bitmap and the palette are selected out of the memory DC.

In function CGDIView::OnDraw(...), the drawing procedure is modified. First the zoomed image is divided horizontally and vertically into small portions, all of which can fit into the memory bitmap. Then each portion of the image is copied to the memory bitmap, and the grid is added if necessary. Next the image contained in the memory bitmap is copied to the client window at the corresponding position. Here we need to calculate the origin and dimension of the output image within each loop. The following portion of function CGDIView::OnDraw(...) shows how the zoomed source image is divided into several portions and copied to the memory bitmap:

(Code omitted)

The values of macros BMP_BKSTORE_SIZE_X and BMP_BKSTORE_SIZE_Y must be multiple of the maximum ratio (In the sample, the maximum ratio is 16 and the values of BMP_BKSTORE_SIZE_X and BMP_BKSTORE_SIZE_Y are both 256). When doing the copy, we must provide the dimension and the origin for both source and target images, whose values are explained in the following table:

(Table omitted)

The actual dimension of the source image that we can display in one loop is stored in variable size (with 1:1 ratio). For the memory bitmap, for each loop the image is copied to its upper-left origin (0, 0); for the source bitmap, the origin depends on the values of i and j.

When we copy the image from the memory bitmap to the client window, we must calculate if the whole bitmap contains valid image. If not, we should draw only the valid part:

(Code omitted)

The grid drawing becomes very easy now. The brush origin can always be set to (0, 0) because both the horizontal and vertical dimensions of the memory bitmap are even. Before the image contained in the memory bitmap is copied to the client window, the grid is added if the current ratio is greater than 2 and the value of CGDIDoc::m_bGridOn is TRUE:

(Code omitted)

With the above implementation, there will be no more flickering when we draw lines with grid on.

Summary

1) Tracker can be implemented by using class CRectTracker. To add tracker to any window, first we need to use CRectTracker to declare a variable, then set its style. To display the tracker, we need to override the function derived from CWnd::OnDraw(...)and call CRectTracker::Draw(...) within it.

2) The style of a tracker can be specified by enabling or disabling flags for member CRectTracker::m_nStyle. The following flags are predefined values that can be used: CRectTracker::solidLine, CRectTracker::dottedLine, CRectTracker::hatchedBorder, CRectTracker::resizeInside, CRectTracker::resizeOutside, CRectTracker::hatchInside.

3) When the mouse left button is pressed, we can call function CRectTracker::HitTest(...) to check if the mouse cursor is over the tracker object. If so, we can call function CRectTracker::Track(...) to track the activities of the mouse. By doing this, there is no need to handle WM_MOUSEMOVE and WM_LBUTTONUP messages.

4) Region can be used to confine the DC output within a specified area. The shape of a region can be rectangular, elliptical, polygonal, or irregular. A new region can be created from existing regions by combining them using logical AND, OR, and other operations.

5) A region must be selected into DC before being used. The function that can be used to select a region is CDC::SelectClipRgn(...). To select the region out of the DC, we can simply call this function again and pass NULL to its parameter.

6) Path can be used to record outputs to the device context. To start path recording, we need to call function CDC::BeginPath(). To end path recording, we can call function CDC::EndPath(). All the drawings between the two function calls will be recorded in the path. The output will not appear on the DC when the recording is undergoing.

7) Function CDC::StrokePath() can be called to stroke the outline of a path. Function CDC::FillPath(...) can be used to fill the interior of the path. Function CDC::StrokeAndFillPath() can be used to implement both.

8) Region can be created from an existing path by calling function CRgn::CreateFromPath(...).

9) A region is made up of a series of rectangles. By resizing all the rectangles, we can resize the region.

10) A path is made up of a series of vectors. To resize a path, we can scale all the control points. We can also change the position of some control points to generate special effects.

11) The standard DIB format that can be used in the clipboard is CF_DIB. To put DIB data to the clipboard, we need to prepare DIB data with standard DIB format, open the clipboard, empty the clipboard, call function ::SetClipboardData(...) and pass CF_DIB flag along with the data handle to it. After all these operations, we must close the clipboard.

12) To obtain DIB data from the clipboard, we can use CF_DIB flag to call function ::GetClipboardData(...).

13) Message WM_PALETTECHANGED is used to notify the applications that the system palette has changed. Applications that implement logical palettes should realize the logical palette again after receiving this message to achieve least color distortion.

14) Outputting directly to window may cause flickering sometimes. This is because

usually the old drawings must be erased before the window is updated. To avoid flickering, we can prepare everything in a memory bitmap, then output the patterns contained in the memory bitmap to the window in one stroke.

BACK TO INDEX

Chapter 12 Screen Capturing & Printing

By now we already have a lot of experience of using both DIB and DDB along with logical palette. But it is still not enough. All the samples we created in the previous chapters start from DIB, so we know the color table before the image is displayed in the window, and therefore can implement logical palette and realize it before the pixels are actually drawn. By doing this, we can always get the best color result.

However, sometimes we need to create image starting from the DDB. If we use DDB to record image data, it would be very simple because we don't have to worry about the actual data format and palette realization. However, there are two problems: 1) If we want to save the image data to a file, we still need to convert it to DIB format. 2) If the hardware is a palette device, the entries of the system palette may actually change (e.g., when another application realizes its own logical palette). In this case, the colors contained in our DDB may not be correctly mapped if we do not implement logical palette for it.

A very typical application of this kind is the screen capturing application. Screen capture can be implemented by copying images between the desktop window DC and our own memory DC. We can call function CDC::BitBlt(...) to copy the images. In this case, the memory DC must select a DDB rather than DIB. If the system uses palette device, this may cause potential problem. Think about the following situation: we use screen capturing application to make a snapshot of the desktop screen, and currently there is a graphic editor opened with a colorful image being displayed. As long as the system palette remains unchanged, the captured image can be displayed in its original colors. Suppose the graphic editor is closed and this causes the system palette to change, the captured image will also change accordingly.

To prevent this from happening, when capturing the screen, we need to copy not only the image bits, but also the colors contained in the current system palette. Then we can use them to create a logical palette, and convert the DDB to DIB data. In the client window, we can display the image using DIB data instead of captured DDB data. By doing this, when the system palette changes, we can re-match the logical palette to achieve the best effect.

Samples in this chapter are specially designed to work on 256-color palette device. To customize them for non-palette devices, we can just eleminate logical palette creation and realization procedure.

12.1 Capturing the Whole Screen

Capture

Making capture is very simple. We already have a lot of experience of creating bitmap in memory, selecting it into a memory DC, and copying it to the client window. We can make a screen capture by reversing the above procedure. The following lists the necessary steps for making a screen capture: 1) Create a blank bitmap in the memory. 2) Create a memory DC that is compatible with the window DC. 3) Select the memory bitmap into the memory DC. 4) Obtain a valid window DC. 5) Copy the image from the window DC to the memory DC. Here the window DC can be either the desktop window DC or a client window DC. In the former case, the whole screen will be captured. In the later case, only the client window will be captured.

Sample 12.1\GDI demonstrates how to make screen capture. It is a standard SDI application generated by Application Wizard, whose view class is based on CScrollView. In the sample, function CGDIView:: Capture() is added to capture the whole desktop screen and store the image in a CBitmap type variable.

Under Windows(, desktop window is the parent of the all windows in the system, and its pointer can be obtained by calling function CWnd::GetDesktopWindw(). With this pointer, we can create a DC and use it to draw anything on the desktop (Be careful with this feature, generally an application should not draw outside its own window). Of course, we can also copy a bitmap to the desktop window.

The following is the implementation of function CGDIView::Capture() that shows how to copy the whole screen and store the image in a CBitmap type variable:

(Code omitted)

In the above function, first desktop window DC is created from the pointer of desktop window (By calling function CWnd::GetDesktopWindow()), then the dimension of the desktop window is retrieved and stored in variable rect; next the memory DC and blank memory bitmap are created and the bitmap is selected into the memory DC; finally function CDC::BitBlt(...) is called to capture the whole screen.

Of course we can use the bitmap (bmpCap) and memory DC (dcMem) created here to display the captured image. However, because there is no logical palette associated with the bitmap, if the colors contained in the system palette change, the captured image may not be displayed correctly. So before displaying the image, we need to first convert it to DIB and implement logical palette.

Converting DDB to DIB

Like other samples, after DDB is converted to DIB we'd like to store it in document so that the image can be easily saved to file through serialization. Also, a logical palette is created in the constructor of the document, it will be used to display captured image later. The entries of this palette is not initialized in the constructor. The variables used for storing DIB and logical palette are CGDIDoc::m_hDIB and CGDIDoc::m_palDraw respectively. Like other samples, here we also have corresponding functions in CGDIDoc that makes two variables accessible from outside the document.

In CGDIDoc, function CGDIDoc::ConvertDDBtoDIB(...) is implemented to convert captured DDB format image to DIB. The procedure is almost the same with what we implemented in sample 10.4\GDI (There is also a function CGDIDoc::ConvertDDBtoDIB(...) there), the only difference is where the logical palette comes from. In sample 10.4\GDI, whenever we open a DIB image, we can obtain its color table, which can be used to create the logical palette. Here, we do not have such kind of color table and must create a logical palette from the colors contained in the current system palette and use it to generate the color table for DIB image.

Obtaining colors contained in the system palette is not difficult. Remember in sample 8.9\GDI, we can actually monitor the system palette all the time. To create a palette whose entries are always synchronized to the entries of the system palette, we can set member peFlags of structure PALETTEENTRY to flag PC_EXPLICIT when creating the logical palette (Also, set lower two bytes of structure PALETTEENTRY to an index to the system palette). The logical palette (More accurately, the entries with flag PC_EXPLICT) created this way will not be mapped to the system palette by looking up the same (or nearest) colors or filling empty entries. Instead, each entry will always be mapped to a fixed entry contained in the system palette. If the colors in the system palette change, the corresponding colors in the logical palette will also change.

We need to create this kind of logical palette right after the image is captured (Before the system palette changes) so that the colors contained in the captured image will be represented by the system palette. Although we can use this palette to obtain the correct color table we will use, it can not be used for later DIB displaying. The reason is that the colors contained in these entries may change constantly. We need to create a logical palette using the color table obtained this way (Member peFlags of the palette entries should be set to NULL).

After the logical palette with PC_EXPLICIT flag entries is created, we can select it into DC and realize it. Then we can allocate enough buffers to store BITMAPINFOHEADER type object and color table. When calling function ::GetDIBits(...), we can pass NULL to its lpvBits parameter, this will cause the bitmap header and the correct color table to be filled into the buffers allocated before.

Now that we have the correct color table, we can use it to create a logical palette with member peFlags set to NULL. In the sample, since an uninitialized logical palette is created at the beginning, we can just fill the palette entries after each capturing.

The rest part of DDB-to-DIB conversion is the same with that of sample 10.4\GDI: we just need to calculate the image size, reallocate the buffers, and call ::GetDIBits(...) again to receive actual bitmap bit values.

The following is the implementation of function CGDIDoc::ConvertDDBtoDIB(...), the input parameter is a CBitmap type pointer, and the returned value is the handle of global memory that contains DIB data:

(Code omitted)

New Command

In sample 12.1\GDI, a new command Capture | Go! is added to mainframe menu IDR_MAINFRM. This command is handled by function CGDIDoc::OnCaptureGo(). Within the function, we first minimize the application by calling function CMainFrame::ShowWindow(...) using SW_SHOWMINIMIZED flag. Then we call function CGDIView::PrepareCapture() to set the timer (This function contains only one statement).

Because we will minimize our application window, the capture should be delayed for a few seconds after the user executes Capture | Go! command. Function CGDIView::PrepareCapture() does nothing but setting up a timer with time out period set to 2 seconds. Function CGDIView::Capture() is called when the timer times out. Within function CGDIView::Capture(), after the capture is made, function CGDIDoc:: GetCaptureBitmap(...) will be called to convert DDB to DIB and update the client window (Within this function, CGDIDoc::ConvertDDBtoDIB(...) will be called).

In CGDIView::OnDraw(...), we use function ::SetDIBitsToDevice(...) to display DIB data. This is the same with sample 10.4\GDI.

Figure 12-1 shows the time sequence of function calling.

(Figure 12-1 omitted)

12.2 Capturing a Specified Window

One problem of the sample implemented in the previous section is that it can capture only the whole desktop window. To improve it, we will add some new functions so that the user can specify any window for capturing.

Picking Up a Window

Often there are many windows contained in the desktop window. Each application may have one mainframe window and several child windows. To let the user pick up a window with mouse clicking, we must find a way to detect if there is a window under the current mouse cursor.

We can call function CWnd::WindowFromPoint(...) to retrieve the window under current mouse cursor. If there is such a window, its handle will be returned by this function. Otherwise the function will return NULL.

We also need to monitor mouse movement and respond to its activities when the user is selecting a window. We all know that this can be implemented by handling mouse related messages: WM_LBUTTONDOWN, WM_MOUSEMOVE, and WM_LBUTTONUP. Also we must be able to receive these messages even if the mouse cursor is outside our application window. To implement this, we need to set window capture. By doing so, the user can hold the left button and move it anywhere to pick up window. As long as the left button is held down, we can receive WM_MOUSEMOVE messages (The capture will be released by the system if the left button is released).

Another issue is how to indicate the selected window. To make the application easier to use, we need to put some indication on the window that is being selected. From the previous section we know that the DC of the desktop window can be obtained and used to draw anything anywhere on the screen. With the desktop window DC, we can reverse the selected window when the mouse cursor is over it, and resume it as the mouse cursor leaves the window.

The drawing mode can be set by calling function CDC::SetROP2(...) using R2_NOT flag. After this if we draw a rectangle (By calling function CDC::Rectangle(...)) using the window's position and size, the whole window will be reversed. Calling this function twice using the same parameter will resume the original window.

One thing we must pay attention to is that the application itself also has a mainframe window, and therefore will be selected for capturing if the mouse cursor is over it. This is not a desirable feature. To solve this problem, after function CWnd::WindowFromPoint(...) is called, we must check if the window returned by this function belongs to the application itself.

Because the windows can overlap one another, when the mouse cursor is over one window, we should not reverse the overlapped part (Figure 11-2). To solve this problem, we need to use region. We can create a region that contains only the non-overlapped part of a window, which can be selected by the desktop window DC. By doing this, the overlapped portion of the window will not be reversed when the function CDC::Rectangle(...) is called.

After the user has selected a specific window and executed Capture | Go! command, we can find out the size and position of this window, start timer, call function CDC::BitBlt(...) to make a snapshot of the window and store the data to the memory bitmap.

Dialog Box IDD_DIALOG_SELECT

Sample 12.2\GDI is based on sample 12.1\GDI. The new sample allows the user to select a specified window for making snapshot. First a dialog box is added to the application, it will be used to let the user select a window. The new dialog template is IDD_DIALOG_SELECT, and the class associated with it is CSelDlg. To make the interface user friendly, mouse cursor will be changed when the user is selecting a window (With left button held down). In the sample, a cursor resource IDC_CURSOR_SELECT is added for this purpose. The cursor is loaded in the constructor of CSelDlg and its handle is stored in variable CSelDlg::m_curSelect. In the dialog box template, an icon that contains the cursor is displayed. If the user click on this icon, it will be changed to a blank icon and at the same time, the cursor will be customized to IDC_CURSOR_SELECT. If the user releases the mouse button, everything will be resumed. This will give the user a feeling that the mouse clicking actually picks up the cursor (Figure 12-3).

The icons displayed in the dialog box are also stored as resources. Their IDs are IDI_ICON_CURSOR and IDI_ICON_BLANK. Also, they are loaded in the class constructor and are displayed in the dialog box by calling function CStatic::SetIcon(...). The control used to display the icon is a static control (Actually it is added as a "Picture" control). In the property sheet "Picture Properties", we can choose the type of images that will be displayed, such as bitmap or icon (Figure 12-4).

Messages WM_LBUTTONDOWN, WM_LBUTTONUP and WM_MOUSEMOVE are handled to let the user select a window. When the left button is pressed down, we check if it hits the icon contained in the dialog box. If so, we set window capture for the dialog box, change the mouse cursor, and change icon IDI_ICON_CURSOR to IDI_ICON_BLANK:

(Code omitted)

Variable CSelDlg::m_hWnd is used to store the handle of the selected window. Also, variable CSelDlg::m_rectSelect is used to indicate the rectangle of the previously selected window. If this rectangle is empty, no window is currently being reversed.

As the mouse moves, we will keep on receiving WM_MOUSEMOVE messages. In the sample, function CSelDlg::DrawSelection(...) is implemented to handle this message. Within this function, first we create a region that contains all of the desktop window excluding the area occupied by the application window. We select this region into the desktop window DC before reversing any window. By doing this, if the selected window is overlapped by the application window, the reversing effect does not apply to the application window. The following portion of function CSelDlg::DrawSelection(...) shows how the region is created:

(Code omitted)

In order to resume the reversed window, the drawing mode should be set to R2_NOT, which will reverse all the pixels contained in the rectangle when function CDC::Rectangle(...) is called. The drawing mode can be set by calling function CDC::SetROP2(...):

(Code omitted)

Then we call function CWnd::WindowFromPoint(...) to see if the current mouse cursor is over any window. We use the returned pointer to retrieve the handle of that window, and compare it with the handles of our application windows (both mainframe window and dialog box window). If there is a match, we should not draw the rectangle because the cursor is over the application window (In this case, if there exists a window that has been reversed, we should resume it). Otherwise, we further compare it with the handle stored in CSelDlg::m_hWnd, if they are the same, we don't do anything because the window under the cursor has been reversed. If not, this means a new window is being selected and we should resume the old reversed window (If there exists such a window) then reverse the newly selected one:

(Code omitted)

In function CGDIView::Capture(), we need to first obtain handle CGDIDoc::m_hWnd from the document, if it is not a valid window handle, we still capture the whole desktop screen. Otherwise we use this handle to find out the rectangular area of the window and make the snapshot.

12.3 Simple Printing

Although we didn't write a single line of code to implement printing feature, all our SID or MDI samples have the default printing functionality. This includes default printer set up, print preview, and printing the client window. Like display, printer is another type of graphic device that can be used to output drawings. Its interface to the software programmer is similar to that of display: instead of writing code to control the hardware directly, we can use DC to output drawings to the printers. Actually we can call member functions of class CDC to output dot, line, curve, rectangle and bitmaps to a printer.

Mapping Mode

However, there are some differences between printing devices and display devices. One main difference is that two devices may have different capabilities. Because all displays have similar sizes and resolutions, it is relatively convenient to measure everything on the screen using pixel. For example, it doesn't make much difference if we display a 256(256 bitmap on an 800(600 display or a 1024(768 display. Since every window is able to display an object that is larger than the dimension of its client area (using scroll bars), it is relatively easy to base every drawing on the minimum possible unit ¾ pixel.

For printers, this is completely different. There are many types of printers in the world, whose resolutions are remarkably different from one another. For example, there are line printers, one pixel on this kind of printers may be 0.1mm(0.1mm; also, there are many types of laser printers, whose resolution can be 600dpi, 800dpi or even denser. If we display a 256(256 image on the two types of printers, their sizes will vary dramatically.

Anther difference between printing devices and display devices is that when doing the printing, it is desirable to make sure that all the outputs fit within the device. For a window, since we can customize the total scroll size, it doesn't matter what the actual output dimension is (The scroll size can always be set to the dimension of output drawings). For the printer, we need to either scale the output to let it fit within the device or print the output on separate papers.

In order to handle this complicated situation, under Windows(, OS and devices have some common agreements. When we draw a dot, copy a bitmap to device, everything is actually based on logical units (pixels). By default, the size of one logical unit is mapped to one minimum physical pixel on the device, however, this can be changed. Actually class CDC has a function that let us customize it:

virtual int CDC::SetMapMode(int nMapMode);

Parameter nMapMode specify how to map one logical unit to physical units of the target device. By default it is set to MM_TEXT, which maps one logical unit to one physical unit. It can also be set to one of the following parameters:

(Table omitted)

Instead of mapping one logical unit to a fixed number of pixels, it is mapped to a fixed size on the target device. It is the device driver's task to figure out the actual number of pixels that should be used for drawing one logical pixel. By doing this type of mappings, the output will have the same dimension no matter what type of target device we use.

There is one difference between MM_TEXT mapping mode and other modes: for

MM_TEXT mode, the positive y axis points downward. For other mapping modes, the positive y axis points upward. So if we decide to use one of the mapping mode listed above, and the origin of the bitmap is still the same, we need to use negative values to reference a pixel's vertical position (Figure 12-5).

Converting between Logical and Device Units

Sometimes we need to implement the conversion between logical unit and actual device unit. Class CDC has a bunch of functions that allow us to do the conversion between two coordinate systems. For example, CDC::LPtoDP(...) allows us to convert a point (POINT type variable) or a size (CSize type variable) measured in logical unit coordinate system to device coordinate system. And CDC::DPtoLP(...) does the reverse.

Implementing Print

Actually it is easy to implement printing for applications generated from the Application Wizard. When the user executes File | Print or File | Print Preview command, a series of printing messages will be sent to the application. Upon receiving these messages, the frame window finds out the current active view, and call that view's CView::OnPrint(...) function to output drawings to the target device.

By default, CView::OnPrint(...) does nothing but calling function CView::OnDraw(...), so everything contained in the client window (view) will also be output to the printer. We can experiment this with the sample already implemented. For example, after executing sample 12.2\GDI, if we make a snapshot and execute File | Print command, the captured image will be sent to the printer. The actual size of the output image depends on the type of printer because in CGDIView::OnDraw(...), we didn't set the mapping mode so the default mode MM_TEXT is used.

We must know the resolution of the target device so that we can either scale the output to let it fit into the device or we can manage to print one image on separate papers. One way of obtaining the target device's resolution is to call function CDC::GetDeviceCap(...) using HORZRES and VERTRES parameters. The returned value of this function will be the horizontal or vertical resolution of the target device, measured in its device unit. Besides this, we can also use LOGPIXELSX and LOGPIXELSY to retrieve the number of pixels per logical inch in the target device for both horizontal and vertical directions. Since the width and height of a minimum pixel in the target device may not be the same, the above two parameters may be different.

Scaling the Image before Printing

Sample 12.3\GDI is based on sample 12.2\GDI. In this sample, when doing the printing, the output is scaled so that it can fit within one sheet of paper no matter what the resolution of target device is. In order to implement this, we need to calculate the proportion between the logical unit and the physical unit of the target device before the image is output to the device.

For example, suppose we have an image whose logical dimension is x(y, and want to output it to the target device whose physical resolution is px(py. We further assume that the aspect ratio of a minimum physical pixel on the target device is rx : ry. This is illustrated by Figure 12-6:

In Figure 12-6, x=4, y=2, px=12, py=9, rx : ry = 1:2. First we choose px as the actual width of the output image. In this case, we need to map 4 logical units to 12 physical units in the horizontal direction. So the horizontal mapping ratio is selected as 1:3. If we do the same thing in the vertical direction, we will have a 12(6 image on the target device. However, because a basic pixel on the target device is not square, the proportion of the output image will change if we do not take it (rx : ry ratio) into consideration. To compensate for this, we need to map one logical unit to (px/x) ((rx/ry) pixels in the vertical direction. In this sample, we will have a 12X3 image on the target device.

If we have a very tall image (For example, in Figure 12-6, if x=2, y=9), such mapping may cause some portion of the image unable to fit into the target device. So after calculating the mapping using the above method, we need to check the vertical physical size and see if it is greater than py (if y((px/x) ((rx/ry) > py). If so, we need to calculate the mapping again by first setting the vertical size to py and then calculating the horizontal size using the same method.

Displaying or Printing?

In function CView::OnDraw(...), this mapping is unnecessary if the target device is a display rather than a printer. To find out if this function is being called for printer, we can call function CDC::IsPrinting(). If the returned value is FALSE, the function is called to output drawings to display. Otherwise it is called to output drawings to printer.

Function CGDIView::OnDraw(...)

The following portion of function CGDIView::OnDraw(...) shows how to scale the image dimension so that it will fit within the target device before being output to the printer. Since the image must be scaled before being printed, we call function ::StretchDIBits(...) to implement printing and still use ::SetDIBitsToDevice(...) for painting the client window:

(Code omitted)

The physical resolution of the target device is retrieved and stored in variables cxPage and cyPage, and the number of pixels per logical inch are retrieved and stored in variables cxInch and cyInch, which can be used to calculate the aspect ratio of a basic pixel on the target device. The logical dimension of the image is stored in members lpBi->bmiHeader.biWidth and lpBi->bmiHeader.biHeight. With the above parameters, it is easy to figure out the actual physical size of the output image. The output dimension is stored in variable rcDest, and function ::StretchDIBts(...) is called to output the captured image to printer.

12.4 Fixed Scale Printing

The solution in the previous section scales image so that it will always fit within the target device. However, this is not he best solution. For example, if we need to print a small button image, the image will be enlarged to fit the paper; if we want to print a very big image, it will be shrunk and we will inevitably lose some details.

Printing Related Functions

To let the printing output always have the same fidelity, we need to call function CDC::SetMapMode(...) to map one logical unit to a fixed value. For example, if we map one logical unit to 0.1mm, a 256(256 image will always have a 25.6(25.6cm2 dimension, no matter what type of target device we use. When doing the printing, we can further call function CDC::StretchBlt(...) or ::StretchDIBits(...) to scale one logical unit to multiple of 0.1 millimeters. For example, if we set one logical unit to 0.1 mm and call ::StretchDIBits(...) to scale the image to three times of its original dimension, one logical unit will ultimately equal to 0.3 mm.

Usually printing is handled in function CView::OnPrint(...). If we do not override it, function CView::OnDraw(...) (Or its overridden function) will be called to implement the default printing. The advantage of using CView::OnPrint(...) instead of CView::OnDraw(...) is that function CView::OnPrint(...) has two parameters, one is a pointer to the target device context, the other is a CPrintInfo type pointer that brings us a lot of information of the printing device. We will see how to use this parameter in later sections.

Before printing begins, function CView::OnBeginPrinting(...) will be called. A CDC type pointer and a CPrintInfo type pointer will be passed to this function, from which we can obtain the information of current printing status. If we need to prepare something before the printing starts, we can override this function. This function is necessary because there exist some applications whose printing output is different from what is displayed in the client window. For example, if we are programming a video editing application, what can be displayed in the client window is usually one of a series of images. When the user does the printing, we actually want to print all the images. In this case, we must create either DIB or DDB data before the printing starts. Function CView::OnBeginPrinting(...) is a best place to implement such kind of preparation: we can create GDI objects, allocate memory, set device mapping modes, and so on. Since the GDI objects and memory prepared here are solely used for printing, after the printing is done, we must destroy them. Function CView::OnEndPrinting(...) is designed for this purpose, it will be called when the printing task is over.

When the printing is undergoing, function CView::OnPrint(...) will be called. We can use CDC type pointer passed to this function to output objects to the printing device, this is the same with outputting objects to display device.

Sample 12.4\GDI

Sample 12.4\GDI is based on sample 12.3\GDI. In this sample the printing is handled in function CGDIView::OnPrinting(...), and CGDIView::OnDraw(...) is only responsible for painting the client window. Since we can use the DIB stored in the document for printing, we do not need to do any preparation in function CGDIView::OnBeginPring(...). Within function CGDIView::OnPrinting(...), we first need to obtain the DIB and palette from the document, and set the mapping mode to MM_LOMETRIC, which will map one logical unit to 0.1 mm. Then we need to select the palette into the target DC, call function ::StretchDIBits(...) to copy the image to target device. Please note that after the mapping mode is set to MM_LOMETRIC, the direction of y axis is upward. When calling function ::StretchDIBits(...), we must set the output vertical dimension on the target device to a negative value if the origin is still located at (0, 0):

(Code omitted)

Now no matter what type of printer we use, the output dimension will be the same. The only difference between the output from two different types of printers may be the image quality: for printers with high DPIs, we will see a smooth image; for printers with low DPIs, we will see undesirable image.

Although the printing ratio is fixed for this sample, the user can still modify it through File | Print Setup... command. In the Print Setup dialog box, the user can also select printer, paper size and the printing ratio. The maximum printing ratio that can be set by the user is 400%. This may cause the output image unable to fit within the target device. In this case, we need to print one image on separate pages.

12.5 Printing on Separate Pages

If we want to output an image on separate papers, there are two situations: 1) Number of required pages is known before the printing starts. 2) Number of required pages has to be decided after the printing starts. For different situations, we need to use different approaches.

Number of required Pages is Known Beforehand

Sample 12.5-1\GDI is based on sample 12.4\GDI and demonstrates how to implement printing when the total number of pages is known beforehand.

For some applications, the number of required pages for printing is fixed. For this situation, we need to override function CView::OnPreparePrinting(...), and call function CPrintInfo::SetMaxPage(...) to set the page range. By doing this, when the printing is being processed, function CView:OnPrint(...) will be called repeatedly until all the pages are printed out. Within CView::OnPrint(...), the page information can be obtained from member CPrintInfo::m_nCurPage (The second parameter of this function is a pointer to class CPrintInfo), which stands for the current page number that is being printed. According to this number, we can output different contents to different pages. The printing will be stopped after all the pages are printed out.

For example, suppose whenever we want to print out the captured image, we'd like to make two copies, one with 100% ratio and one with 200% ratio. In this case the number of pages is determined beforehand. Before the printing begins, we can set the number of pages in function CGDIView::OnPreparePrinting(...) as follows:

(Code omitted)

By doing this, if the user executes printing command, in the popped up dialog box, the total number of pages will be set to 2. The user can choose to print any of the pages or both of them. In function CGDIView::OnPrint(...), we need to check which page is being printed and call function ::StretchDIBits(...) using the corresponding ratio:

(Code omitted)

Setting Number of Pages Just Before Printing Starts

However this is not a normal case. Because the user can actually change the printing ratio, the number of pages actually needed depends on print settings. For example, when the user set the printing ratio to 400%, the image that can originally fit into one page (when the ratio is 100%) often requires more than one page now. So the actual number of pages needed depends on the printing ratio, which can range from 25% to 400%.

One solution to this problem is to calculate the actual number of pages just before the printing starts. At this time, the print setting will not be changed any more, so we can calculate the number of required pages and call function CPrintInfo::SetMaxPage(...) to set the page range.

This can be done in either function CView::BeginPrinting(...) or in function CView:: OnPrepareDC(...). The first function will be called just before the print job begins, and the second function will be called before CView::BeginPrinting(...) is called when the printing DC needs to be prepared. Please note that CView::OnPrepareDC(...) will also be called for preparing display DC, to distinguish between the two situations, we can check if parameter pInfo is NULL, if not, it is called for the printing job.

The number of required pages can be calculated by retrieving the device resolution (need to be converted to logical unit) and comparing it with the image size. If the image size is greater than the device resolution, we can print one portion at a time until the whole image is output to the target device.

Sample 12.5-2\GDI is based on sample 12.4\GDI, it demonstrates how to print the captured image using this method. In the sample, first function CGDIView::OnPrepareDC(...) is overridden, within which the number of required pages is calculated as follows:

(Code omitted)

Here, the device resolution is retrieved by calling function CDC::GetDeviceCaps(...). Because the device mapping mode is MM_LOMETRIC, which may cause the values returned by this function to be negative for vertical dimensions, we need to use absolute value when doing the calculation.

Within function CGDIView::OnPrint(...), we print the corresponding portion of the image according to the current page number. This procedure can be illustrated in Figure 12-7.

The shaded area represents the image. It is divided into horizontal and vertical cells, each cell has a dimension that is the same with target device resolution. To draw the image, we need nine pages, each page print one cell that is labeled (v, u). First we need to calculate the cell label from the page number:

v = (page number - 1)/(number of horizontal cells)

u = (page number -1)%(number of horizontal cells)

Here the page number starts from 1. The next step is to calculate the position and the dimension of the cell. Obviously, the origin of a cell rectangle can be calculated as follows:

origin X = u((device horizontal resolution)

origin Y = v((device vertical resolution)

If the cell is not the one located right-most or bottom-most, the horizontal and vertical sizes of the cell can be determined from the resolution of target device. If it is located right-most (i.e., cells (0, 2), (1, 2) and (2, 2) in Figure 12-7), the horizontal size can be calculated using the following formulae:

size X = image horizontal size (u ((cell horizontal size)

Similarly, if the cell is located bottom-most, the cell's height can be calculated using the following formulae:

size Y = image vertical size (v ((cell vertical size)

The following shows the procedure of calculating the dimension of a cell in function CGDIView::OnPrint(...) (Sample 12.5-2\GDI):

(Code omitted)

Variables nRepX and nRepY are used to store the number of cells in horizontal and vertical directions, and variable rectDC stores the device resolution. When a cell is copied, its dimension is calculated and stored to variable rect. The following portion of function CGDIView::OnPrint(...) shows how a cell is output to the device:

(Code omitted)

Calculating the Number of Pages when the Printing Is Undergoing

Sample 12.5-3\GDI uses an alternate method to calculate the number of required pages for printing. It is based on sample 12.4\GDI. Instead of calculating the number of pages before printing begins, we can set the number of pages to the maximum value then start printing. Each time a page is printed, we check if all of the image has been output to the printer. If so, we can stop printing.

The advantage of this method is that the actual number of required pages need not to be decided beforehand. This is especially useful for the situation when data cannot be formatted before being printed.

By applying this method, we do not need to calculate the number of required pages in function CGDIView::OnPrepareDC(...). Instead, we can first set it to the maximum possible value in function CGDIView::OnPreparePrinting(...):

(Code omitted)

Please note that we must use 0xFFFFFFFE instead of 0xFFFFFFFF to set the maximum page range, the latter will result in printing only the first page.

If we do not add further control, the printing will not stop until 0xFFFFFFFE pages have been printed. To stop printing after all the image has been printed out, we need to calculate the total required number of pages and check if the page currently being printed is the last page in function CGDIView::OnPrint(...). If so, we set the page range again to stop printing:

(Code omitted)

In our case the number of pages can actually be decided before the printing begins, so it seems not necessary to stop printing this way. However, for applications that the number of pages cannot be decided beforehand, this is the only method to implement multiple-page printing.

12.6 Customizing Print Dialog Box

In this section, we will discuss the topics on how to enhance the user interface for implementing print set up, which has nothing to do with GDI.

Customizing Common Controls

One customization we want to make is to disable radio buttons labeled "Pages" and "Selection", along with the edit boxes labeled "from:" and "to:" after the user executes "File | Print" command (Figure 12-8). This dialog box is implemented by print common dialog box, which was not discussed in Chapter 7. The purpose of this dialog box is to let the user setup the printer before the printing task starts. In MFC, the class used to implement this dialog box is CPrintDialog.

In standard SDI and MDI applications, we don't need to add anything in order to include the print dialog box. The print dialog box can be used to do either print setup or printer setup. The constructor of CPrintDialog must have at least one input parameter, which indicates if the dialog box should be implemented for print setup or printer setup:

(Code omitted)

Since the initialization procedure of the print dialog box is implemented within other MFC member functions, as a programmer, we can only modify the dialog box after it is created. Remember in function CGDIView::OnPreparePrinting(...), one of the input parameters is a CPrintInfo type pointer. The print dialog box is embedded in this class. The member used to store the dialog box is CPrintDialog type pointer CPrintInfo::m_pPD. We can modify any of its member to change the dialog box style before function CView::DoPreparePrinting(...) is called.

Class CPrintDialog contains a PRINTDLG type object m_pd that allows us to customize the style of the dialog box. Here structure PRINTDLG is similar to OPENFILENAME structure of class CFileDialog. It also has a member Flags that allows us to specify the styles of the print dialog box. Two flags we will use in the sample are listed in the following table:

(Table omitted)

Sample 12.6-1\GDI is based on sample 12.5-3\GDI, it demonstrates how to disable these controls. The following is a portion of function CGDIView::OnPreparePrinting(...) of sample 12.6-1\GDI showing how the styles of the print dialog box are customized:

(Code omitted)

Other styles can also be customized by using this method.

Using Custom Dialog Template

Like other common dialog boxes, we can use our own dialog template to replace the standard one. This procedure is similar to that of other common dialog boxes. To use custom dialog template, we need to make changes to the default values of the following members contained in structure PRINTDLG: 1) Enable PD_ENABLEPRINTTEMPLATE flag. 2) Assign the custom dialog template name to member lpPrintTemplateName. 3) Assign the application instance handle to member hInstance.

Sample 12.6-2\GDI is based on sample 12.5-3\GDI and demonstrates how to use programmer- provided dialog template to implement print dialog box.

The first step of implementing customized print dialog box is to add a new dialog template. We can copy this dialog template from file "Commdlg.dll" and modify it. Please note that we cannot delete the original controls contained in the template. If we want to hide certain controls, we can either move them out of the template, or disable them in the dialog box's initialization stage. In sample 12.6-2\GDI, the new dialog template is PRINTDLG. For the purpose of demonstration, no change is made to the original template.

The next step is to derive a class from CPrintDialog. If we double click on the dialog template, we will be prompted to add a new class for it. Since CPrintDialog is not in the list of base classes, we can first choose CDialog as the base class and change all the keywords CDialog to CPrintDialog later. Please note that the constructors of CPrintDialog and CDialog are different, so if we choose automatic method, we also need to change the constructor created by the Class Wizard. In the sample, the new class is CPrnDlg. There is no new variable or function added to it because we don't want to make further modification. The only thing implemented in the new class is that flags PD_NOPAGENUMS and PD_NOSELECTION are set in the constructor so that the radio buttons and edit boxes will be disabled. This is the same with sample 12.6-1\GDI.

The place where we can use this dialog box is still in function CGDIView::OnPreparePrinting(...). Since there is a default print dialog box implemented, we must delete the old one before we can use our own. After we allocate memory and implement a new print dialog box, we must assign it to member CPrintInfo::m_pPD. Also, we must set template name, instance, and enable PD_ENABLEPRINTTEMPLATE flag. The following shows how the customized print dialog is implemented in function CGDIView:: OnPreparePrinting(...):

(Code omitted)

With the above implementation, the print dialog box will use the custom dialog template PRINTDLG.

Summary

1) To capture the screen, we need to obtain a DC of the desktop window, prepare blank memory bitmap, and call function CDC::BitBlt(...) to make the copy.

2) For palette devices, we must obtain the system palette after a snapshot is taken. This can be implemented by creating logical palette using PC_EXPLICIT flags.

3) One logical unit can be mapped to different size on the target device. If we use MM_TEXT mode, one logical unit will be mapped to one physical unit. We can also use other mapping modes to map one logical unit to an absolute length.

4) To fit the output within the target device, we need to know the resolution of the device, along with the number of pixels contained in one inch for both horizontal and vertical directions. These parameters can be obtained by calling function CDC::GetDeviceCaps(...) and using flags HORZRES, VERTRES, LOGPIXELSX and LOGPIXELSY.

5) If we want the output image to have the same size on any type of target devices, we need to use a mapping mode other than MM_TEXT.

6) If the number of pages is fixed for printing output, we can call CPrintInfo::SetMaxPage(...) in function CView::OnPreparePrinting(...) to set the page range. The page number will be passed to function CView::OnPrint(...) to let us customize the print output.

7) If the number of pages can only be decided just before the printing starts, we can call CPrintInfo::SetMaxPage(...) in function CView::BeginPrinting(...) or CView::OnPrepareDC(...).

8) If the number of pages must be decided when the printing is undergoing, we can

set the page range to its maximum value before the printing starts, and call CPrintInfo::SetMaxPage(...) to stop printing dynamically in function CView::OnPrint(...).

BACK TO INDEX

Chapter 13 Adding Special Features to Application

Normal applications created by Application Wizard cannot satisfy us all the time. For certain types of applications, we need to add special features to our programs. Since Application Wizard or Class Wizard does not directly support these features, we need to have in-depth knowledge on Windows(programming in order to customize standard applications. In this chapter, we will discuss how to create applications with special features such as multiple documents, multiple views, irregular-shaped window, customized non-client area. Also, we will discuss how to implement hook in the applications.

13.1 One Instance Application

By default, a Windows(application is allowed to have multiple instances working simultaneously. Most of the time this is the desired feature of an application. For example, a word processing program may have several instances working together, each editing a different file. But sometimes we may want an application to have only one instance working at any time, this is especially true for some communication programs. For example, for a file server application, if we allow two servers to work together at the same time, it may cause the inconsistency on the data contained in the files.

Window Creation

To implement one instance application, we must understand how the applications are created under Windows(. This is easily understood if we have the experience of writing Win32 Windows(applications. However, if we started everything from MFC, it is not very obvious how a window is created because MFC hides everything from the programmer. Although it is relatively easy to create an application by deriving classes from MFC without caring about the actual procedure of creating windows, if we rely too much on MFC, we also lose the power of customizing it.

Every visual object that is created under Windows(is a window. This includes the frame window, tool bar, menu, view, button and other controls. Actually, MFC is not the only tool that can be used to create windows. A window can be created by using any computer language such as C, Basic, Pascal so long as it abides by the rules of creating windows.

Under Windows(, a window can be described by structure WNDCLASS:

```
typedef struct _WNDCLASS {

UINT style;

WNDPROC lpfnWndProc;

int cbClsExtra;

int cbWndExtra;

HANDLE hInstance;

HICON hIcon;

HCURSOR hCursor;

HBRUSH hbrBackground;

LPCTSTR lpszMenuName;

LPCTSTR lpszClassName;

} WNDCLASS;
```

Member style specifies window styles, by setting different bits of this member we can create different type of windows. There are many styles that can be combined together, two most often used styles are CS_HREDRAW and CS_VREDRAW, which will cause the client area to be updated if the user resize the window in either horizontal or vertical direction. Member lpfnWndProc points to a callback function that will be used to process the incoming messages. When a window is created, it should contain several default objects: 1) icon, which will be used to draw the application when it is minimized; 2) cursor, which will be used to customize the mouse cursor when it is located within the client window of the application; 3) default mainframe menu; 4) brush, which will be used to erase the client area (This brush specifies the background pattern of the window). The above objects are described by following members of structure WNDCLASS respectively: hIcon, hCursor, hbrBackground and lpszMenuName.

Another very important member is lpszClassName, which describes the type of the window we will create. Every window under Windows(has a class name. Before creating a new type of window, we must register its class name to the system. After that we can use this class name to implement an instance of window. A class name is simply a string, which can be specified by the programmer.

If we write windows program in C, we must go through the following procedure in order to create a new window: register the window class name, implement a message handling routine, use the registered window class name to implement a new window instance. In MFC, this procedure is hidden behind the classes, when we use a class derived from CWnd to declare a variable, the class registration is completed sometime before the window is created. Also, we do not need to provide message handling routines because there exist default message handlers in MFC. If we want to trap certain messages, we can add member functions and use message mapping macros to associate the messages with functions.

It is relatively easy to implement one-instance application by programming in C: before registering a window class, we can first find out if there exists any instance implemented by the same class name in the system. If so, we simply exit and do not go on to create a new window. If not, we will implement the new window.

However, in MFC, we do not see the class registration procedure, so it is difficult to manipulate it. Also, in MFC, all the window class names are predefined, so we actually can not modify them. In order to create one-instance application, we need to discard the default registered window class name, and use our own class name to create new instance. By doing so, we are able to check if there already exists an instance of this window type before creating a new one.

Function CWnd::PreCreateWindow(...)

The styles of a window (including the class name) can be modified just before it is created. In MFC, function CWnd::PreCreateWindow(...) can be overridden for this purpose.

The input parameter of CWnd::PreCreateWindow(...) is a CREATESTRUCT type variable, which is passed to the function by reference:

```
virtual BOOL CWnd::PreCreateWindow(CREATESTRUCT& cs);
```

Structure CREATESTRUCT contains a variety of window styles:

```
typedef struct tagCREATESTRUCT {

LPVOID lpCreateParams;

HANDLE hInstance;
HMENU hMenu;

HWND hwndParent;

int cy;

int cx;

int y;

int x;

LONG style;

LPCSTR lpszName;

LPCSTR lpszClass;

DWORD dwExStyle;

} CREATESTRUCT;
```

We can specify a new menu and use it as the mainframe menu. We can set the initial size and position of the window. We can also specify window name, and customize many other styles. Within the structure, the window class name is specified by member lpszClass. By default, this structure is stuffed with standard values, however, we can change any of them to let the application have new styles. For example, if we want to use "My class" as the class name of our application rather than using the default one, we need to implement the overridden function as follows:

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)

{

cs.lpszClass="My class";

return CMDIFrameWnd::PreCreateWindow(cs);

}
```

One-Instance Application in MFC

In MFC, one-instance application can be implemented as follows: before the application is implemented, we need to find out if there is any registered application that uses specified window class name: if so, we should exit; otherwise, we can proceed to register our own window class name, and override function CFrameWnd::PreCreateWindow(...) to change the default class name to the new one. By doing so, an application can have only one instance implemented in the system at any time.

In MFC, an application starts from class CWinApp. After an application is executed, the very first function being called is the constructor of the class derived from CWinApp. Of course we can implement class name checking and registration here. However, a better place is in function CWinApp::InitInstance(), where the application is being initialized. For SDI and MDI applications, the mainframe window, document and view are implemented and bound together here, for dialog box based applications, the main dialog box is also implemented within this function.

Sample 13.1\Once

Sample 13.1\Once is a standard MDI application generated by Application Wizard, it demonstrates how to implement one-instance MDI application. The application has no functionality except that if we try to activate more than one copy of this application, instead of creating a new instance, the existing one will always be brought up and become active.

Instead of using default class name, we need to register a custom class name to the system. The first thing we need to do before frame window, document and view are implemented is to look up if there exists an application with the same class name in the system:

(Code omitted)

Function CWnd::FindWindow(...) is called to find the application with the same class name in the system. This function allows us to search windows with specific class name and/or window name. It has the following format:

static CWnd *CWnd::FindWindow(LPCTSTR lpszClassName, LPCTSTR lpszWindowName);

We can pass NULL to window name parameter (lpszWindowName) to match only the class name. If the pointer returned by this function is not NULL, we can activate that window, bring it to top, and activate all its child windows. This procedure is implemented by calling the following functions: 1) CWnd:: GetLastActivePopup(), which will find out the most recently activated pop-up window. 2) ::ShowWindow(...), which will restore the original state of the window being minimized (parameter SW_RESTORE can be used for this purpose). 3) CWnd::SetForegroundWindw(), which will bring the child window to foreground if there is a such kind of window. Steps 1) and 3) are necessary here because a mainframe window may own some pop up windows (For example, a dialog box implemented by a command of the mainframe menu). Once the mainframe window is brought to the top, we also need to bring its child pop up window to foreground.

After this is done, we need to return a FALSE value, which indicates that the procedure of creating mainframe window, document and view is not successful. This will cause the application to exit.

If no window with the same class name is found, we can proceed to register our own window class. This can be done by stuffing a WNDCLASS type object and passing the object address to function AfxRegisterClass(...), whose only parameter is a WNDCLASS type pointer:

```
BOOL AFXAPI AfxRegisterClass(WNDCLASS *lpWndClass);
```

In order to make sure that our application is the same with those implemented by default MFC window classes, we must stuff the class with appropriate values. Here is how this is done in the sample:

(Code omitted)

The class name string is defined using ONCE_CLASSNAME macro. Of course, when we override function CMainFrame::OnPreCreateWindow(...), we need to replace the default class name with it. The following code fragment shows how this function is overridden in the sample:

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)

{

cs.lpszClass=ONCE_CLASSNAME;

return CMDIFrameWnd::PreCreateWindow(cs);

}
```

Before the application exits, we must unregister the window class if it has been registered successfully. For this purpose, a Boolean type variable m_bRegistered is declared in class COnceApp, which will be set to TRUE after the class is registered successfully. When overriding function CWinApp::ExitInstance() (which should be overridden if we want to do some cleanup job before the application exits), we need to unregister the window class name if the value of m_bRegistered is TRUE. The following is the overridden function implemented in the sample:

(Code omitted)

That's all we need do for implementing one-instance application.

13.2 Creating Applications without Using Document/View Structure

Document/View structure provides us with much convenience on data storing and interpreting. But sometimes using this structure is burdensome, especially when we want to implement just a very simple application. For example, if we want to display a fixed content in the client window and do not want to write any data to the disk, there is no need to implement document in the application at all.

How Application, Document and View Are Bound Together

Although it seems that document and view are inborn for SDI or MDI applications, we do have way to get rid of them. Actually, the document and view are created in function CWinApp::InitInstance(...) and bound to mainframe window there. The following shows standard implementation of a typical SDI application, if we examine the SDI applications created before, we will find the following code in all of them:

(Code omitted)

Class CSingleDocTemplate binds together the mainframe window, view and document. It creates view and makes it client of the frame window. Obviously, by eliminating these statements, we are able to create our own client window without bothering to use document and view.

Creating Window

However, if we do not let the framework to create the mainframe window and a client window for us, we have to do it by ourselves. Fortunately creating a window is not so difficult, we can call function CFrameWnd::Create(...) at any time to dynamically create a window with both title bar and client window:

```
virtual BOOL CFrameWnd::Create

(

LPCTSTR lpszClassName, LPCTSTR lpszWindowName, DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID, CCreateContext* pContext = NULL

);
```

Here, we are asked to provide some information of the window that is about to be created. This includes class name, window name, window styles, window position and size, etc. The class name must be a registered one. Although we can register our own window class name as we did in the previous section, we can also pass NULL to parameter lpszClassName to let the default registered class name be used. Also, we can pass NULL to parameter dwStyle to use the default window style, and pass rectDefault to parameter rect to let the window have default position and size.

Sample 13.2\Gen

Sample 13.2\Gen demonstrates how to create applications without using document/view structure. Originally it is a standard SDI application generated by Application Wizard. Then it is modified to become an application that does not use document/view implementation.

Rather than creating mainframe window then using view as its client window, in the sample, only a mainframe window is created by calling function CFrameWnd::Create(...). This can be done in the constructor of class CMainFrame. In sample 13.2\Gen, the constructor is modified as follows:

(Code omitted)

We must provide a window name, which will be displayed in the caption bar of the window. In standard SDI or MDI applications, this string can be obtained from string resource IDR_MAINFRAME. To make the sample similar to a standard application, we can load this string and use it as the window name. Since we will not support any file type, in the sample string resource IDR_MAINFRAME contains only one simple string (This means it does not contain several sub-strings that are separated by character '\n' as in standard SDI or MDI applications).

For a simple application, there is no need to implement status bar and tool bar any more, so function CMainFrame::OnCreate(...) is removed, and variables CMainFrame::m_wndStatusBar, CMainFrame:: m_wndToolBar along with another global variable indicators are also deleted.

For any application implemented by MFC, it is originated from class CWinApp. This class has a CWnd type member pointer m_pMainWnd. When the mainframe window is created, its address is stored by this pointer. If we want to create window by ourselves, we must do the same thing in order to let the rest part of our application have MFC features.

With the above implementation, function CGenApp::InitInstance(...) can be greatly simplified, what we need to do here is implementing a CMainFrame type object, assigning its address to CGenApp:: m_pMainWnd, then calling functions CWnd::ShowWindow(...) and CWnd::UpdateWindow(...) to display the window. This last step is necessary, if we omit it, the window will not be displayed. The following is the modified function in the sample:

(Code omitted)

Here variable m_nCmdShow indicates how the window should be displayed (minimized, maximized, etc). It must be passed to function CWnd::ShowWindow(...) in order to initialize the window to a specified state.

Excluding Classes from Build

Although we do not need to use view and document classes anymore (CGenView and CGenDoc in the sample), it is difficult to remove them from the project once they are generated automatically. However, we can change the project settings so that the two classes will not be complied when the project is being built. This can be achieved through following steps: 1) Executing command Project | Settings.... 2) From the popped up dialog box, expand "Source Files" node in "Settings For:" window. 3) Select "GenView.cpp" and click on "General" tab on the right part of the dialog box. 4) Check "exclude file from build" check box (Figure 13-1). We can do the same thing for file "GenDoc.cpp" to exclude class CGenDoc from build.

Now there is no view in the application. If we want to output something to the client window, we have to implement it in class CMainFrame. Of course, there is no member function OnDraw(...) to override any more. In order to output anything to the client window, we need to override function CMainFrame:: OnPaint(...), which is the handler of WM_PAINT message. We have to prepare DC by ourselves, and, if we want, we need to add scroll bars and calculate the offset positions all by our own. By eliminating the document and view, we have a simple implementation of the application and a smaller executable file. But if we need a simple feature that is not supported by MFC, we have to implement everything by ourselves.

13.3 Implementing Multiple Views

Sometimes we do not want the default document/view implementation, but sometimes we need more than standard features. One thing we might think about when creating MDI applications is: is it possible to implement different types of views to interpret data stored in the document? A typical example of this is that we can use both "bar chart" and "pie chart" to interpret percentages (Figure 13-2).

Simple View Implementation

Since the document/view creating and binding procedure is not the task of programmer (when Application Wizard is used), it is easy to be neglected. The code for creating document and view then binding them together resides in function CWinApp::InitInstance(). For example, if we create an MDI application named "Chart", by default, the CWinApp derived class will be named CChartApp, also, the document and view classes will be named CChartView and CChartDoc respectively. In member function CChartApp::InitInstance(), the above objects are bound together by class CMultiDocTemplate:

(Code omitted)

The constructor of class CMultiDocTemplate has four parameters, the first of which is a string resource ID, which comprises several sub-strings for specifying the default child window title, document type, and so on. The rest three parameters must use RUNTIME_CLASS macro, and we can use the appropriate class name as its parameter. In the code listed above, the child window uses class CChildFrame to create the frame window, and uses CChartView to create the client window. The child window is attached to the document implemented by class CChartDoc.

Attaching Multiple Views to One Document

If we need only one type of view, this is enough. However, if we want to attach multiple views to a single document, we can call function CWinApp::AddDocTemplate(...) again to bind a new type of view to the document.

Sample 13.3\Chart

Sample 13.3\Chart demonstrates how to attach multiple views to one document. It is a standard MDI application generated by Application Wizard. The purpose of this application is to interpret data stored in the document in different ways. The original classes generated by Application Wizard are CChartApp, CChartDoc, CChartView, CMainFrame and CChildFrame. After the application skeleton is generated, a new class CPieView (derived from CView) is add to the application through using Class Wizard.

Data stored in the document is very simple, there are three variables declared in class CChartDoc: CChartDoc::m_nA, CChartDoc::m_nB and CChartDoc::m_nC. The variables are initialized in the constructor as follows:

(Code omitted)

Three variables each represents a percentage, so adding them up will result in 100. There are many different types of charts that can be used to interpret them, two most common ones are "bar chart" and "pie chart".

In the sample application, two different types of views are attached to one document, so the user can use either "Bar chart" or "Pie chart" to view the data. To obtain data from the document, function CChartDoc::GetABC(...) is implemented to let these values be accessible in the attached views.

In function CChartView::OnDraw(...), three bars are drawn using different colors, their heights represent the percentage of three member variables. For class CPieView, three pies are drawn in different colors and they form a closed circle, whose angles represent different percentages.

Two views are attached to the document in function CChartApp::InitInstance(). Besides the standard implementation, a new document template is created and added to the application. The following portion of function CChartApp::InitInstance() demonstrates how the two views are attached to the same document:

(Code omitted)

With the above implementation, when a new client is about to be created, the user will be prompted to choose from one of the two types of views. Here strings that are included in the prompt dialog box should be prepared as sub-strings contained in the string resources that are used to identify document type (IDR_CHARTTYPE and IDR_PIETYPE in the sample).

The format of the document type string is the same with that of a normal MDI application, which comprises several sub-strings. The most important ones are the first two sub-strings: one will be used as the default title of the client window and the other will be used in the prompt dialog box to let the user select an appropriate view when a new client window is about to be created. In the sample, the contents of string IDR_CHARTYPE and IDR_PIETYPE are as follows:

```
\nBar\nBar\n\n\nChart.Document\nChart Document
```

```
\nPie\nPie\n\n\nChart.Document\nChart Document
```

Both the window and the string contained in the prompt dialog box for chart view are set to "Bar", for pie view, they are set to "Pie".

Window Origin and View Port Origin

When implementing drawing on the target device, sometimes it is more convenient if we use appropriate coordinate system. As a programmer, when we write code to draw geometrical shapes, we are always working on Page-Space (logical space). The actual output would happen on Device-Space. By default, one logical unit is mapped to one device unit, and both origins are located at upper-left corners.

When drawing certain types of geometrical shapes, for example, a circle, it would be more convenient if we adjust the origin of the coordinate system so that it is located at the center of the circle (See Figure 13-3).

To offset origin in either page-space or device-space, we can use the following functions:

(Table omitted)

It is the ratio, rather than their absolute values, of window extents and view port extents that specify how a logical unit will be mapped in horizontal as well as vertical directions. It is important that if use MM_ISOTROPIC mode, after calling function CDC::SetMapMode(...), CDC::SetWindowExt(...) needs to be called before CDC::SetViewportExt(...).

Pie Chart Drawing

In the sample, we need to draw three pies that form a circle. Since we need to calculate the starting and ending points for each pie, it would be convenient if the origin of the coordinate system is located at the center of the circle. Also, to assure that the circle will not change to ellipse on any device, we need to set MM_ISOTROPIC mapping mode.

The following is the implementation of function CPieView::OnDraw(...):

(Code omitted)

In the above code, first MM_ISOTROPIC mode is set. Then the window extents is set to (100, 100). To map one logical unit to an absolute size, function CDC::GetDevice(...) is called using both LOGPIXELSX and LOGPIXELSY parameters. This will cause the function to return the number of pixels per logical inch in both horizontal and vertical directions. Then we use the returned values to set view port extents. This will cause 100 logical units to be mapped to 1 inch in both horizontal and vertical directions. By doing this, no matter where we run the program, the output will be the same dimension.

When calling function CDC::SetViewportExt(...), we set the vertical extent to a negative value. This will change the orientation of the y-axis so that the positive values locate at the upper part of the axis (See Figure 13-3).

Next, function CDC::SetViewportOrg(...) is called to set the device origin to the center of the window. This will simplify the calculation of starting and ending points when drawing pies.

13.4 Multiple Documents Implementation

Not only can we implement an application with more than one type of views, but also implement an application that supports more than one type of documents. For example, generally a graphic editor needs to support several types of image files, such as bitmap and GIF files. Although we can support all file formats within one document, for MDI applications, the source code will become easy to manage if we use one document type to support one file format.

Actually, the procedure of implementing more than one document is almost the same with adding more than one view to an application. All we need to modify is still function CWinApp::InitInstance(), within which we must create a new document template and call function CWinApp::AddDocTemplate(...) to bind the new document (along with a view) to the mainframe window.

From the sample application created in the previous section, we know that when a new document template is being created, we need to provide a resource ID, a document class name, a view class name, and a frame window class name. If we look at the menu and icon resources of an MDI application, we will find that ID IDR_MAINFRAME is used in three different places: there is a string resource using this ID, which will be used as the mainframe window caption text; there is a menu resource using this ID, which will be used to implement the application's main menu (when there is no child window open); there is an icon resource using this ID, which will be put to the top-left corner of the application (left side of the title bar).

When a document template is created, we also need to provide a resource ID, which will be used to implement the above resources when the client window is open. Like IDR_MAINFRAME, the corresponding string resource will be used to display the title of the child frame and document type; the menu resource will be used to implement application menu when the corresponding client window is open; the icon resource will be displayed in the top-left corner on the client window.

We may have noticed that in the previous sample, when we open a window implemented by class CPieView, the mainframe menu will be changed to IDR_MAINFRAME, and the top-left icon is a general icon. This is because we didn't prepare menu and icon resources for ID IDR_PIETYPE.

Sample 13.4\Chart is based on sample 13.3\Chart and demonstrates how to further support a new type of document in the application.

In the sample, just for the purpose of demonstration, a new type of document is added without implementing anything (It does not contain any data). Although it is a dummy document, by attaching a CEditView type view to it we know that several documents can co-exist in one application.

The class name of the new document is CTextDoc, and is added through using Class Wizard. The view that will be associated with it is CTextView, which is derived from class CEditView. No special change is made to the two classes. In function CChartApp::InitInstance(), after two views are attached to class CChartDoc, the new view is attached to the new document and they are bound together to the mainframe window:

(Code omitted)

Besides this, we also added following resources to the application: icons IDR_PIETYPE and IDR_TEXTTYPE; menus IDR_PIETYPE and IDR_TEXTTYPE; string IDR_TEXTTYPE.

13.5 Painting Caption Bar

Non-client Area and Related Messages

The caption (title) bar belongs to non-client area of a window. Actually, any window can be divided into client and non-client areas. By default, the application itself is responsible for implementing client area painting, and the system is responsible for non-client area painting. The non-client area includes caption bar, menu, frame and border.

Generally an application should not paint the non-client area. But sometimes we do need to customize the default implementation to create some special effects. While the client area painting is managed by message WM_PAINT, non-client area has a counterpart message WM_NCPAINT.

When painting the caption bar, we need to pay attention to the current window states. By default, a window's caption bar is painted with blue color if the application is in the foreground (In other word, when the application is active), and painted with gray color if it is in the background (When it is inactive). When customizing the caption bar, we also need to put some indication on it to distinguish between the above two states.

When painting the non-client area, the message used to distinguish active and inactive states of a window is WM_NCACTIVE. Its WPARAM parameter indicates whether the caption bar needs to be painted to indicate active or inactive state: if it is 0, the application is about to become inactive; otherwise the application is about to become active. Please note that in the latter case, the non-client area painting should not be processed. The active state of the non-client area should always be painted after receiving message WM_NCPAINT (instead of WM_NCACTIVE).

Since we only want to change the appearance of default caption bar, we will let the rest non-client area be painted by default implementation. To do this, after receiving WM_NCPAINT and WM_NCACTIVATE messages, we can first call the default message handlers (Which will cause the non-client area to be painted by the default method), then paint the caption bar using our own implementation.

The default handlers of the above two messages are functions CWnd::OnNcPaint() and CWnd:: OnNcActivate(...). By default, they paint the caption bar, draw the icon and system buttons on the caption bar, draw the frame and border.

The other two messages we must handle are WM_SETTEXT and WM_SYSCOMMAND. The first command corresponds to the situation when the caption text is first set or when it is changed. The second message corresponds to the situation when the application resumes from the iconic state to its original state. In the above two cases, after message WM_NCPAINT is sent to the application, the text will be put directly to the caption bar.

Caption Text Area

Figure 13-4 shows the composition of a caption bar: the outer frame, within which there is an icon located at the left side, and three system buttons located at the right of the caption bar. The minimize button and the maximize button are abut together, also, there is a space between them and the close button. By default, the caption text is aligned to the left.

The position and size of a caption window can be obtained by calling function CWnd:: GetWindowRect(...). We need to exclude the frame, icon and buttons in order to calculate the area where we can put the caption text.

So the actual caption text area can be calculated as follows:

left position =

(

left position of the caption window +

border width +

system button horizontal size +

frame width

)

top position = top position of the caption window + frame height

right position =

(

right position of the caption window -

border width -

frame width -

3*(system button horizontal size)

)

bottom position = top position + vertical size of the caption

If we use window DC, the coordinates of the window's top-left corner are (0, 0), this will simplify our calculation.

Please note that we must use class CWindowDC to create DC for painting the non-client area rather than using class CClientDC. Class CClientDC is designed to let us paint only within a window's client area, so its origin is located at left-top corner of the client window. Class CWindowDC can let us paint the whole window, including both client and non-client area.

Sample 13.5\Cap

Sample application 13.5\Cap demonstrates this technique. It is a standard SDI application generated by Application Wizard, and its caption window is painted yellow no matter what the corresponding system color is (The default caption bar color can be customized by the user). The modifications made to the application are all within class CMainFrame, and there are altogether four message handlers added to the application: CMainFrame::OnNcPaint() for message WM_NCPAINT, CMainFrame::OnNcActivate(...) for message WM_NCACTIVATE, CMainFrame::OnSetText() for message WM_SETTEXT and CMainFrame:: OnSysCommand(...) for message WM_SYSCOMMAND.

The function implemented for drawing caption text is CMainFrame::DrawCaption(...). This function has one COLORREF type parameter color, which will be used as the text background. Within this function, several system metrics are obtained, which will be used to calculate caption text area later:

(Code omitted)

The caption text is obtained by combining the name of currently opened document with the string stored in resource AFX_IDS_APP_TITLE. Here resource AFX_IDS_APP_TITLE stores application name, and function CDocument::GetTitle() returns the name of currently opened document.

Then the area where we can put caption text is calculated and stored in a local variable rectDraw. Before drawing the text, we need to fill it with the background color:

(Code omitted)

Since the DC is created using class CWindowDC, the coordinates of the window's origin are (0, 0). Before drawing the text, we need to set the text background mode to transparent. Also, in the sample, when the caption text is being drawn, it is centered instead of being aligned left:

(Code omitted)

Function CMainFrame::DrawCaption() is called in several places. When WM_NCACTIVATE message is received and the window state is about to become inactive, we paint the caption bar with cyan color. When WM_NCPAINT message is received, the caption bar is painted with yellow color.

Also, when WM_SETTEXT or WM_SYSCOMMAND messages are received, the caption needs to be updated. So within the two message handlers, message WM_NCPAINT is sent to the mainframe window:

(Code omitted)

Figure 13-5 shows the result of the above implementation.

(Figure 13-5 omitted)

13.6 Irregular Shape Window

Theoretically speaking, all windows created in Windows(system must be rectangular. This satisfies our needs most of the time. However, sometimes it would be more preferable to let windows have irregular shapes. For example, in multimedia type applications, sometimes we need to implement a special elliptical (or more complex shape) "callout" window with a pointer pointing to an object, with the explanation of the object displayed within the ellipse. The user may feel free to resize or move this window, and edit the text within it (Figure 13-6).

Problem

To implement such type of window, we can let the application paint only within the elliptical area and leave the rest area unchanged.

However, this will cause problem when the user moves or resizes the window. Although only the elliptical area is painted, the window is essentially rectangular. By default, the portion not covered by the ellipse will be treated as the background (It is not updated when message WM_PAINT is received). The application itself can handle WM_ERASEBKGND message to update the background. To let the window have an irregular shape, instead of painting this area with any pattern, we need to make it transparent. In order to achieve this, we shouldn't do anything after receiving message WM_ERASEBKGND. However, this still will cause new problem when the window is moved or resized: since the application does not update its background client area, original background pattern will remain unchanged after moving and resizing (This will cause something doesn't belong to the window background to move along with it).

Style WS_EX_TRANSPARENT

A window's background could be made transparent by using style WS_EX_TRANSPARENT when we create a window by calling function CWnd::CreateEx(...). Unfortunately, in MFC, the window creation procedure is deeply hidden in the base classes. Although it is very easy to create special windows such as frame windows, views, dialog boxes, buttons, we actually have very few controls over their styles.

Another difficult thing is that, if we want to create an irregular shape window, normally we do not want it to have caption bar. If we want to create the window by ourselves instead of using MFC, we need to choose appropriate window styles and take care everything by ourselves, which may be a very complex issue.

Using Dialog Box

To simplify this procedure, we can start from creating a dialog box and change it to an irregular shape window. Since dialog box is also a type of window, it has all the customizable styles belonging to a normal window (A dialog box does not have to contain any common controls).

In property sheet "Dialog Properties", we have a lot of choices for changing the styles of a dialog box. This property sheet contains several pages such as "General", "Styles", "More Styles" and "Extended Styles". Within each property page, we can set different window styles. The following table lists some important issues need to be takern into consideration when designing a window with irregular shape:

(Table omitted)

All other styles remain unchanged, we need to use default settings for them.

Sample 13.6\Balloon is implemented this way. It is a dialog box based application generated by the Application Wizard, and the two classes used to implement the application are CBalloonApp and CBalloonDlg. After the skeleton is created we can open the default dialog box template (In the sample, the template is IDD_BALLOON_DIALOG), remove the default buttons and controls, then customize the window styles as mentioned above.

Disabling Default Background Painting

By default, class CDialog will paint the background with gray color (button face color), although the dialog box's background is transparent, the client area will still be painted with default brush when being erased. Thus we will see temporary gray background when the dialog box is being resized or moved. To prevent the background from being erased with brush, we need to handle message WM_ERASEBKGND to bypass the default background erasing activities. In the sample, this message is mapped to function CBalloonDlg:: OnEraseBkgnd(...), which does nothing but returning a TRUE value to give the system an impression that the background erasing has completed:

```
BOOL CBalloonDlg::OnEraseBkgnd(CDC *pDC)

{

return TRUE;

}
```

Disabling Non-client Area Painting

We also need to pay attention to non-client area. Although the window does not have a caption bar, it has resizable border, which also belongs to non-client area. By default, the border will be painted as a 3D frame, which can be used for resizing the window. To make it transparent, we need to handle message WM_NCPAINT. We don't need to provide any implementation here because there is nothing to be painted. In the sample, this message is handled in CBalloonDlg::OnNcPaint(), which is an empty function:

```
void CBalloonDlg::OnNcPaint()

{

}
```

We do need to override CBalloonDlg::OnPaint() to paint the elliptical area. Within this function, an ellipse is drawn in the client area, also a pointer is drawn at the left bottom corner. Both ellipse and its pointer are painted using yellow color.

Moving the Window with Mouse

Because the application does not have caption bar, we must provide a method to let the window be moved through using mouse. This should be implemented by handling three messages: WM_LBUTTONDOWN, WM_LBUTTONUP and WM_MOUSEMOVE. When the left button is pressed down, we need to set the window capture. As the mouse moves (with left button held down), we need to move the window according to the new mouse position by calling function CWnd::MoveWindow(...). When the left button is released, we need to release the window capture.

Two variables are declared in class CBalloonDlg for this purpose: Boolean type variable CBalloonDlg::m_bCapture and CPoint type variable CBalloonDlg::m_ptMouse. Here, CBalloonDlg:: m_ptMouse is used to record the previous mouse position. Whenever the window needs to be moved, we call function CWnd::GetWindowRect(...) to obtain its current position and size, then offset the rectangle (The size and position of the window is stored in a CRect type variable) according to both current and previous mouse positions, and update variable CBalloonDlg::m_ptMosue. Since all the calculation is carried out in the desktop coordinate system, we need to call function CWnd::ClientToScreen(...) to convert the mouse position before using it (Mouse position is passed to the message handlers in the coordinate system of the application window). The following is the WM_MOUSEMOVE message handler implemented in the sample:

(Code omitted)

This implementation is slow, because when mouse moves a small distance, the whole window need to be redrawn at the new position. An alternative solution is to draw only the rectangular outline when mouse is moving (with the left button held down), and update the window only when the left button is released. To implement this, instead of calling function CWnd::MoveWindow(...) in WM_MOUSEMOVE message handler, we need to call function CDC::DrawDragRect(...) to erase the previous rectangle outline and draw a new one.

For this sample, as the left button is clicked on any portion of the rectangular window area, the application will respond. If we want the window to be movable only when the clicking happens within the elliptical area, we need to use region. To implement this, instead of calling function CRect::PtInRect(...), we can call CRgn::PtInRegion(...) to respond to the left button clicking. Also, if we want to make resizing more flexible, we can test if the mouse cursor is over the border of the ellipse rather let the resizing be handled automatically (By default, window's rectangular frame will be used as the resizing border). To implement this, we need to change mouse cursor's shape when it is over the border of the ellipse, set capture when the left button is pressed down, release capture when the button is released, and resize the window according to the mouse movement.

Although the application does not resemble a dialog box, we can still find some of its features: it can be closed by pressing ENTER or ESC key. To modify this feature, we need to override function CDialog::OnPreTranslateMsg(...). If we implement this, we must provide a way to close the window, otherwise the user has to ask OS to end its life everytime.

13.7 Saving Initial States

It would be preferable to let the application remember its current states when being closed, and resume them next time it is activated. This feature is especially useful for an editor-like application. Generally, we need to save the mainframe window's size, position, and state (is it minimized or maximized?). If there is splitter window implemented in the application, we also need to remember the size of each individual pane. Besides this, we can make the application more attractive by saving information of each tool bar, which include the on/off state, docking state, size and position.

Where to Save the Information

To save this kind of information, we need to write it to hard disk when the application exits. Of course we can manage this by opening a file and write our data to it. However, under Windows(, there is a better way to implement it. We can either store all the information in an ".ini" file under certain directory or store it in the registry. The latter is a better solution because this will make the file system much cleaner. For every application generated by Application wizard, we can find the following statement in function CXXXApp::InitInstance():

```
SetRegistryKey(_T("Local AppWizard-Generated Applications"));
```

This will set a registration key in the registry, all the information stored by the application will be under this key. In the above statement the registration key is "Local AppWizard-Generated Applications". By default, all the applications generated by the Application Wizard will share this key. If we want the application to use a different key, we can simply change this default string.

Functions Used to Write and Read Information

We have four standard functions to save and load the information. By using these functions, we can either save a string or an integer, and read them back:

(Code omitted)

Functions CWinApp::WriteProfileInt(...) and CWinApp::GetProfileInt(...) can be used to save and load integers, and the rest two functions can be used to save and load strings.

Format of ".ini" File

The stored information is organized into sections and entries, we can save relevant information under one section using different entries. For example, the following is a portion of an ".ini" file, within which the window size, position and splitter window information is stored:

```
[Window Position]

Window Position=0, 0, 200, 200;

Window State=Normal;

[Splitter Window]

Vertical Size=100;
```

There are two sections here, the section keys are "Window Position" and "Splitter Window" respectively. Under "Window Position" section, there are two entries, the first is "Window Position" and the second is "Window State", both of them store strings. The second section is "Splitter Window", it has only one entry "Vertical Size", which stores an integer. When we store and retrieve a particular entry, we need to provide the correct section key and entry key.

Sample 13.7\Ini

Now that we know how to save and load the information, we need to find out what kind of information need to be saved. Sample 13.7\Ini demonstrates how to create an application that can resume its previous states including the window state (minimized, maximized, or normal state), size, position and the states of the tool bar. It is a standard SDI application generated by Application Wizard, which has a default tool bar. Its client window is implemented by a two-pane splitter window.

The most appropriate place to save the state information is before the application is about to exit. This corresponds to receiving message WM_CLOSE, which indicates that the application will terminate shortly. Since most information concerns the top parent window of the application, it would be more convenient if we handle this message in the mainframe window.

To retrieve a window's position and size, we can call function CWnd::GetWindowRect(...). The values obtained through calling this function will be in the coordinate system of the desktop window. When the application is invoked next time, we need to call function CWnd::MoveWindow(...) to resume its previous position and size. This should be handled in function CMainFrame::OnCreate(...). The following two functions show how the frame window information is saved and loaded:

(Code omitted)

The window state information is retrieved by calling functions CWnd::IsIconic() and CWnd:: IsZoomed(). If both of them return FALSE, the window is in normal state. To set the window to zoomed or iconic state, we need to set variable CIniApp::m_nCmdShow to either SW_SHOWMINIMIZED or SW_SHOWMAXIMIZED in function CWinApp::InitInstance()(This variable is declared in base class CWinApp). The following portion of function CIniApp::InitInstance() shows how the window state is set:

(Code omitted)

Finally, saving and loading the states of tool bar is very simple, all we need to do is calling function CFrameWnd::SaveBarState(...) to save the tool bar state after receiving message WM_CLOSE and calling function CFrameWnd::LoadBarState(...) in CFrameWnd::OnCreate(...) to load them. No matter how many tool bars are implemented in the application, all of their states will be saved and loaded automatically. The following code fragment demonstrates this:

(Code omitted)

With the above implementation, the application is able to remember its previous states.

13.8 Exchanging User-Defined Messages Among Applications

We all know that standard Windows(messages can be sent to other windows by calling functions CWnd::SendMessage(...) and CWnd::PostMessage(...), we also know that we can create user defined messages that can be used within one process. However, user defined messages can only be sent within one application, there is no way to send them to other applications.

Registering User Defined Messages

Under Windows(, there are several ways to share information and data among several processes, among them sending message is the simplest one. If we want to share user-defined messages among different processes, rather than defining a message with ID greater than WM_USER, we need to register the messages to the OS so that they are unique in the whole system.

The function used for registering user defined messages is ::RegisterWindowMessage(...). The input parameter to this function should be anull-terminated string, and its returned value is the message ID that could be used for later communication. If another application wants to use this message, it must first complete the message registration by calling the above function.

Using registered messages is almost the same with using standard messages. We can call either CWnd::SendMessage(...) or CWnd::PostMessage(...) to send out the message. When doing this, we can specify both WPARAM and LPARAM parameters. Finally, we can map the registered messages to member functions using message mapping macros.

To map registered messages to member functions, we need to use macro ON_REGISTERED_MESSAGE. This macro has two parameters, the first should be the value returned from function ::RegisterWindowMessage(...), which need to be stored in a global (or static) variable. The second parameter should be the name of the member function that will process the message.

Sample

Sample 13.8\Sender and 13.8\MsgRcv demonstrate how to send user-defined messages between two applications. Here, "Sender" is a dialog box based application, and "MsgRcv" is a list view based application. Both of them register two messages: MSG_SENDSTRING and MSG_RECEIVED. The two macros are defined in "Common.h" header file, which is included by both projects. The user can freely input any number in one of the edit box contained in "Sender", and press "Send" button to send the message to "MsgRcv". Before sending the message, "Sender" will search for "MsgRcv", if the application exists, it will send MSG_SENDSTRING message to it, with the number input by the user as the message parameter. Upon receiving the message, "MsgRcv" will add the number to its list, then send back an MSG_RECEIVED message. Upon receiving this message, "Sender" increments a counter indicating how many messages are sent successfully, and its value will be displayed in the dialog box.

Finding Window & Sending Message

To find application "MsgRcv", function CWnd::FindWindow(...) needs to be called. We can base our search on two things: window class name and window name. Because the window name may actually change during its lifetime, it is better to base our searching on class name. As we know from section 13.1, in order to designate a special class name to a certain window, we need to register the window class name by ourselves and use it for creating the window. In sample 13.8\MsgRcv, the window class name is registered in function CMsgRcvApp::InitInstance() and unregistered in function CMsgRcvApp::ExitInstance(). When stuffing structure WNDCLASS, we use

IDR_MAINFRAME to set the window's default icon and menu resources, so there will be no difference between our application and standard ones. The customized class name (macro CLASS_NAME_RECIEVER) is defined in header file "13.8\Common.h" and is shared by both applications. In function CMainFrame::PreCreateWindow(...) of application "MsgRcv", the mainframe window class name is changed before the window is created:

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT &cs)

{

cs.lpszClass=CLASS_NAME_RECIEVER;

return CFrameWnd::PreCreateWindow(cs);

}
```

Two global variables are declared to store the registered message IDs in file "MainFrm.cpp" for both applications:

```
UINT g_uMsgSendStr=0;

UINT g_uMsgReceived=0;
```

For both applications, in function CMainFrame::OnCreate(...), the messages are registered as follows:

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)

{

g_uMsgSendStr=::RegisterWindowMessage(MSG_SENDSTRING);

g_uMsgReceived=::RegisterWindowMessage(MSG_RECEIVED);

......

}
```

For application "MsgRcv", message MSG_SENDSTRING is mapped to function CMainFrm::OnSendStr(...) as follows:

```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
```

......

ON_REGISTERED_MESSAGE(g_uMsgSendStr, OnSendStr)

END_MESSAGE_MAP()

We use WPARAM and LPARAM parameters to pass the window handle of "Sender" and the number input by the user to "MsgRcv". By sending the handle with message MSG_SENDSTRING, the message receiver can use it to make reply immediately, there is no need to find the window again.

Function CMainFrame::OnSendStr(...) is listed as follows, it demonstrates how message MSG_SENDSTRING is handled in "MsgRcv". Like a normal WM_MESSAGE type message handler, this function has two parameters that are used to pass WPARAM and LPARAM. In the sample, WPARAM parameter is used to pass the window handle of "Sender", from which we can obtain a CWnd type pointer and use it to send MSG_RECEIVED message back. After that the value obtained from LPARAM parameter is added to the list view. The following is this function:

(Code omitted)

On the "Sender" side, if the user presses "Send" button, we call function CWnd::FindWindow(...) to find the mainframe window of "MsgRcv". If it exists, message MSG_SENDSTRING will be sent to it, with the WPARAM parameter set to the window handle of the dialog box and LPARAM parameter set to the number contained in the edit box:

(Code omitted)

Message MSG_RECEIVED is mapped to function CSenderDlg::OnReceive(...). Upon receiving message MSG_RECEIVED, we simply increment the counter and display the new value in the dialog box:

(Code omitted)

It is fun to play with the two applications, because they implement the simplest communication protocol: sending the message, replying with the acknowledgment. By using message communication, we can send only simple information (like integers) to another application. If we want to send complex data structure to other processes, we need to use other memory sharing techniques such as file mapping along with message sending.

13.9 Z-Order

Z-order represents the third dimension of a window besides its x and y position. Under

Windows(, a window can be placed before or after another window, and none of the two windows can have a same Z-order (This means if the two windows share a common area, one of them must be overlapped by the other).

Under Windows(, the Z-order of a top-most application window (the window that does not have parent window) is managed by the OS. When the user clicks the mouse on an application, this window will be brought to the top of the Z-order by default. If this happens, also, the orders of all other windows will be changed accordingly.

A window's Z-order can also be changed from within the application. The function that can be used to implement this is CWnd::SetWindowPos(...). Besides Z-order, this function can also be used to change the x-y position and the dimension of a window. It is more powerful than function CWnd::MoveWindow(...), which could be used to move a window only in the x-y plane.

Function CWnd::SetWindowPos(...) has six parameters:

```
BOOL CWnd::SetWindowPos

(

const CWnd *pWndInsertAfter, int x, int y, int cx, int cy, UINT nFlags

);
```

The middle four parameters (x, y, cx and cy) can be used to change a window's x-y position and size. If we want to change only the Z-order of a window, we can set these variables to 0s and set nFlags to SWP_NOMOVE | SWP_NOSIZE.

The first parameter is a CWnd type pointer, it indicates where the window should be placed. This gives us the power to place a window before or after any existing window in the system. More over, we can specify other four parameters: CWnd::wndBottom, CWnd::wndTop, CWnd::wndTopMost, CWnd:: wndNoTopMost. Among these parameters the most interesting parameter is CWnd::wndTopMost, if we use this parameter, the window will always stay on top of other windows under any condition.

Sample 13.9\ZOrder demonstrates how to change a window's Z-order. It is a dialog based application generated by the Application Wizard. The only controls contained in the dialog template are four radio buttons. If the user click on one of them, function CWnd::SetWindowPos(...) will be called using the corresponding parameter:

(Code omitted)

By checking "wndTopMost" radio button, the dialog box will always stay on top of other windows.

13.10 Hook

Hook is a very powerful method in Windows(programming. Remember when creating the snapshot application in Chapter 12, when the application was made hidden to let the user make preparation, a timer was started. The capture would be made just after the timer times out. This is a little inconvenient, because the time that allows the user to make preparation is fixed. The ideal implementation would be like this: instead of setting a timer, we can predefine a key stroke; the user can feel free to make any preparation as the application is hidden; the capture will be made only after the user presses the predefined key.

However, it seems almost impossible to implement this using normal Windows(programming technique. As the application loses its focus, it will not receive any keyboard related messages, this means we cannot direct the keystrokes to it.

The solution to this problem is using hook, which is a mechanism to intercept the Windows(messages and process them before they reach the destinations. There are many type of system information that can be intercepted, such as keystroke, mouse move, system messages, and so on.

Hook Installation

Hooks can be installed either system wide or specific to a single thread. In the former case, we can monitor the activities in the whole system. To install a hook, we need to provide a hook procedure, which will be used to handle the intercepted message. For different kind of hooks, we need to provide different procedures. For example, the mouse hook procedure and the keyboard hook procedure look like the following:

```
LRESULT CALLBACK KeyboardProc(int code, WPARAM wParam, LPARAM lParam);

LRESULT CALLBACK MouseProc(int nCode, WPARAM wParam, LPARAM lParam);
```

Although they look the same, the meanings of their parameters are different. For keyboard hook procedure, WPARAM parameter represents virtual-key code, and lParam parameter represents keystroke information. For mouse hook procedure, WPARAM parameter represents message ID, and LPARAM represents mouse coordinates. Different types of hooks have different hook procedures, they should be provided by the programmer when one or more types of hooks are implemented.

To install a hook, we need to call function ::SetWindowsHookEx(...):

```
HHOOK ::SetWindowsHookEx(int idHook, HOOKPROC lpfn, HINSTANCE hMod,
DWORD dwThreadId);
```

The first parameter indicates the type of hook, for a keyboard hook, it should be WH_KEYBOARD, for a mouse hook, it should be WH_MOUSE. The second parameter is the address of the hook procedure described above. The third and fourth parameters should be set differently for system wide hook and thread specific hook.

System Wide Hook

The complex aspect of hook is that if we want to install a system wide hook, the hook procedure must reside in a DLL (Except for journal record hook and journal playback hook, which will be introduced in the next section). In this case, parameter hMod must be the instance of the DLL, and dwThreadId should be 0. If we want to install a thread specific hook and the hook procedure resides within the application rather than a separate DLL, parameter hMode must be NULL.

Variables in DLL

Obviously in our case, we want the hook to be system wide, so we have to build a separate DLL. This causes us some inconvenience. When the hook procedure receives the hot-key stroke, it needs to activate the application. But since the DLL and the application are separate from one another, it is difficult to access the application process from the DLL.

Suppose we want to implement the hot-key based screen capturing, as we execute Capture | Go! command (see Chapter 12), we can hide the application by calling function CWnd::ShowWindow(...) using SW_HIDE flag. From now on the application has no way to receive keystrokes, we have to process them in the keyboard hook procedure residing in the DLL. As we get the predefined key stoke, we need to make the capture and call function CWnd::ShowWindow(...) using flag SW_SHOW to activate the application.

We can implement this by sending message from DLL to the application. If the DLL knows the window handle of the application's mainframe window, this can be easily implemented. To pass the window handle to the DLL, we can call a function exported from the DLL when the hook is installed, and ask the DLL to keep this handle for later use.

However, data stored in the DLL is slightly different from data stored in a common process. For a normal process, it has its own memory space, the static variables stored in this space will not change throughout their lifetime. However, for a DLL, since it can be shared by several processes, its variables are mapped to the application's memory space separately. By doing this, for a variable contained in the DLL, different applications may have different values. This will eliminate data conflicting among different processes.

This causes the following situation: if several processes share the same variable contained in the DLL, on the DLL side, the value of this variable may change as the window focus shifts from one process to another.

This will get us into trouble: since the variables relative to one process will only be mapped to it while the process is active, as we hide our application, the handle stored in the DLL previously will not represent the correct value anymore.

Defining Data Segment

To solve this problem, DLL has another feature that enables all the processes using one DLL to share common data among them. In order to do this, we need to specify a special data segment for storing such type of variables. We could use macro #pragma data_seg to specify the data segment, and use -SEGMENT switch to link the project.

DLL Implementation

Sample 13.10\Hook demonstrates keyboard hook implementation. The hook procedure stays in a separate DLL file: "HookLib.Dll". Creating a DLL is slightly different from creating MFC based applications, there is no skeleton generated at the beginning. After using the application wizard to generate a Win32 based DLL project, we are not provided with a single line of source code.

Since our DLL is relatively small, we can use just one ".c" and ".h" file to implement it. We can create these two files by opening new text files, then executing command Project | Add To Project | Files... to add them into the project.

In the sample, the DLL is implemented by "HookLib.h" and "HookLib.c" files.

File "HookLib.h" declares all the functions that will be included in the DLL:

(Code omitted)

Function LibMain(...) and WEP(...) are the entry point and the exit point of the DLL respectively. The reason for using so many #if macros here is that it enables us to use the same header file for both the DLL and the application that links the DLL. As we build the DLL, we want to export functions so that they can be called outside the DLL; in the application, we need to import these functions from the DLL. Macro __declspec(dllimport) declares an import function and __declspec(dllexport) declares an export function. As we can see, if macro __DLLIMPORT__ is defined, function LibMain(...), WEP(...) and KeyboardProc(...) will be declared (they will be used only in the DLL). In this case, two other functions SetKeyboardHook(...) and UnsetKeyboardHook() will be declared as import function. If the macro is not defined, the two functions will be declared as export functions.

The reason for using macro EXTERN_C is that the DLL is built with C convention and our application is built with C++ convention. To make them compatible, we must explicitly specify how to build the functions in the DLL. In the sample, two functions are exported from the DLL: SetKeyboardHook(...) will be used by the application to install hook; UnsetKeyboardHook() will be used to remove the hook.

In file "HookLib.c", first two static variables are declared:

(Code omitted)

We use #pragma data_seg("SHARDATA") to specify that g_hWnd and g_hHook will be stored in "SHARDATA" segment instead of being mapped to calling processes.

Function SetKeyboardHook(...) installs system wide keyboard hook by calling function SetWindowsHookEx(...). When using this function, we must provide the instance handle of the DLL library and the handle of the application's mainframe window:

(Code omitted)

The handle of application's mainframe window is stalled in variable g_hWnd for later use. The handle of the hook is stored in variable g_hHook and will be used in function UnsetKeyboardHook() to remove the keyboard hook:

```
STDENTRY_(BOOL) UnsetKeyboardHook()

{

return UnhookWindowsHookEx(g_hHook);

}
```

Function KeyboardProc(...) is the hook procedure. When there is a keystroke, this function will be called, and the keystroke information will be processed within this function:

(Code omitted)

The first parameter, code, indicates the type of keystroke activities. We need to respond only when there is a keystroke action, in which case parameter code is HC_ACTION. If code is less than 0, we must pass the message to other hooks without processing it (This is because there may be more than one hook installed in the system). In order not to change the behavior of other applications, after processing the keystroke message, we also need to call function ::CallNextHookEx(...) to let the message reach its original destination.

If the keystroke is CTRL+F3, we will check if the application window is visible. If not, function ::ShowWindow(...) is called to activate it. In this case, the keyboard hook will be removed.

We need to use -SECTION link option in order to implement "SHARDATA" data segment. This can be done through executing Project | Settings... command (Or pressing ALT+F7 hot key) then clicking "Link" tab on the popped up property sheet. In the window labeled "Project Options", we need to add "-SECTION:SHARDATA,RWS" at the end of link option. This will make the data in this segment to be readable, writable, and be shared by several processes (Figure 13-7).

Sample 13.6\Hook

Sample 13.6\Hook is a standard SDI application generated by the Application Wizard. In the sample, header file "HookLib.h" is included in the implementation file "MainFrm.cpp", also, macro __DLLIMPORT__ is defined there. This will import two functions contained in the DLL. The following portion of file "MainFrm.cpp" shows how the header file is included and how the macro is defined:

```
#define __DLLIMPORT__

#include "stdafx.h"

#include "..\HookLib\HookLib.h"

#include "Hook.h"
```

To use the functions in DLL, we need to link file "HookLib.Lib" which is generated when the DLL is being built. This can be done by executing command Project | Setting..., clicking tab "Link" from the popped up property page, and entering the path of file "HookLib.Lib" in edit box labeled "Object/Library Modules" (Figure 13-8).

The keyboard hook is installed in function CMainFrame::OnCreate(...). Also, within the function, DLL is dynamically loaded by calling function ::LoadLibrary(...). The returned value of this function is the DLL's instance (if the DLL is loaded successfully), which will be used to install the keyboard hook. The following code fragment shows how the DLL is loaded:

(Code omitted)

The DLL library is released before the application is about to exit in function CMainFrame::OnClose():

(Code omitted)

A command Hide | Go! is added to the mainframe menu IDR_MAINFRAME, this command installs keyboard hook and hides the application. We can press CTRL+F4 to show the application after it becomes hidden. The following is the implementation of this command:

(Code omitted)

The application and the DLL should be in the same directory in order let the DLL be loaded successfully. Or, the DLL may be stored under "Windows" or "Windows\System" directory.

13.11 Journal Record and Journal Playback Hooks

Journal hooks are very interesting, they allow us to write program that can record and playback events happened in the system (such as mouse move, clicking, key stroking, etc.). With journal hooks, it is easy to implement advanced features such as macro recording.

Events recording can be implemented by journal record hook, events playback can be implemented by journal playback hook. Both of the hooks are system wide, and the hook procedures can reside in either a DLL or EXE file. The installation of the two hooks is the same with that of keyboard hook, except that we need to use WH_JOURNALRECORD or WH_JOURNALPLAYBACK parameter when calling function ::SetWindowHookEx(...). Similarly, we need to provide a hook procedure for each installed hook. Usually the names of journal hook procedures are JournalPlaybackProc(...) and JournalRecordProc(...) respectively. Like procedure KeyboardProc(...), both functions have three parameters, however, their meanings are different here:

LRESULT CALLBACK JournalPlaybackProc(int code, WPARAM wParam, LPARAM lParam);

LRESULT CALLBACK JournalRecordProc(int code, WPARAM wParam, LPARAM lParam);

For journal record procedure, we need to record event only when parameter code is HC_ACTION. At this time, the event message is stored in an EVENTMSG type object, whose address can be obtained from parameter lParam. Structure EVENTMSG stores hardware message sent to the system message queue, along with time stamp indicating when the message was posted. We can use the information contained in this structure to implement playback.

Analyzing Events

Another thing we need to pay attention to is that we need to provide a way of letting the user stop recording and rendering playback at any time. By default, Windows(allows journal hook to be stopped by any of the following key stroking: CTRL+ESC, ALT+ESC and CTRL+ALT+DEL. Besides the default feature, it is desirable to provide a build-in function for stopping the journal hook. We can predefine a key for this purpose. The key pressing events can be trapped by analyzing EVENTMSG type object, which has the following format:

(Code omitted)

Member message specifies event type, in order to trap keystroke, we need to respond to WM_KEYDOWN activities. In this case, the virtual key code is stored in the lower byte of member paramL.

Sample 13.11\Hook demonstrates journal record and playback hook implementation. Like previous section, here hook functions are also implemented in the DLL. For this sample, 13.11\Hook is based on 13.10\Hook, and 13.11\HookLib is based on 13.10\HookLib.

For events other than specified key stroke, we need to allocate enough buffers for storing an EVENTMSG type object and bind them together using linked list. To implement this, we can define a new structure of our own:

```
typedef struct tagEVENTNODE

{

EVENTMSG Event;

struct tagEVENTNODE *lpNextEvent;

}EVENTNODE, *LPEVENTNODE;
```

Pointer lpNextEvent will point to the next EVENTMSG structure. This will form a singly linked list.

The following code fragment shows how events are recorded in journal record hook procedure. Like other types of hooks, we need to call CallNextHookEx() to pass the event to other hooks if parameter code is less than 0:

(Code omitted)

In the above code fragment, we check if the event is CTRL+F3 key stroking. If so, we remove the hook and send a message to the application's mainframe window indicating that the recording is finished.

(Code omitted)

In the above code fragment, we try to allocate buffers for recording the event. If this is not successful, we also need to finish the recording.

(Code omitted)

In the above code fragment, if lpeventTail is NULL, this is the first event we will record after the journal record hook has been installed. We must record the current time so that we can play back the recorded events at the rates they were generated.

(Code omitted)

The above code fragment shows how the event is recorded.

Playing back the Recorded Events

The playback is just the reverse procedure. Instead of allocating buffers and recording events, we need to analyze the recorded singly linked list and playback every event then free the buffers. When implementing the playback, parameter code indicates what we need to do: if it is HC_SKIP, we need to get the next event for playback; if it is HC_GETNEXT, we need to copy the event to an EVENTMSG type object whose address can be obtained from parameter lParam. Here is how we get the next event when parameter code is HC_SKIP:

(Code omitted)

Here, if there is no more event, we need to reset everything and send a message (the message is a user defined message WM_FINISHJOURNAL, see below) to the mainframe window of the application indicating that the playback is over. If there are still events left, we get the next recorded event and free the buffers holding the event that is being played back. If parameter code is HC_GETNEXT, we need to obtain an EVENTMSG type pointer from parameter lParam, and copy the event pointed by lpEventPlay to the object pointed by this pointer. When doing this copy, we need to add an offset to the time stamp because originally it indicates the time when the events were recorded:

(Code omitted)

Using Functions Contained in DLL

To notify the mainframe window about journal finishing event, a user registered message is used to communicate between the application and DLL. In the DLL, function ::RegisterMessage() is called after it is loaded, which will register WM_FINISHJOURNAL message in DLL. This message is also registered in the application's CMainFrame::OnCreate(...) function. In the sample, two menu commands Macro | Record and Macro | Playback are added to the application. To avoid journal playback, journal record and keyboard hook from getting entangled, only one of the three hook related commands are enabled at any time. Besides this, playback hook could not be installed if journal record hook has not been installed. This is controlled by the following two variables declared in the application: CMainFrame::m_bEnableMenu, CMainFrame::m_bPlayAvailable.

Installing and removing the hooks is simple. The following three functions show how the journal record hook and journal playback hook are implemented in the application. Here, CMainFrame::OnMacroRecord() implements command Macro | Record, CMainFrame::OnMacroPlayback() implements command Macro | Playback, and CMainFrame::OnFinishJournal(...) handles message WM_FINISHJOURNAL:

(Code omitted)

To test the program, we can first execute Macro | Record command, then use mouse or keyboard to generate a series of events. Next, we can press CTRL+F3 to stop recording. Finally we can execute Macro | Playback command to see what has been recorded.

13.12 Memory Sharing Among Processes

In sample applications created in section 13.8, we demonstrated how to share user defined messages among different processes. However, with this method, we can send only simple parameter (integer) with every message. Sometimes it is necessary to share complex data among different processes, in which case we must apply memory sharing method.

Problem with Global Memory

Is it possible to share buffers allocated by ::GlobalAlloc(...) function among different processes? If so, we can embed memory handle in the message parameters and send it to another process. Upon receiving the message, the corresponding process can obtain the global memory handle and call ::GlobalLock(...) to access all the memory buffers.

Unfortunately, although it is a possible method to share data among different processes for Win16 applications, it is not possible for us to do so for Win32 based applications. In a 32-bit operating system, virtual memory is used to map physical memory to logical memory for each separate process, so each process has its own memory space from 0 to infinity (ideally). Therefore, if the two processes have the same logical address, they actually indicate different physical addresses. So if we send memory address from one process to another, it will not indicate the original buffer.

File Mapping

To solve this problem, in Win32 platform, there is a new technique that allows different processes to share a same block of memory. This technique is called file mapping, and can be used to let different processes share either a file or a block of memory.

The file or memory used for this purpose is called File Mapping Object and must be created using special function. After it is created successfully, each process can open a view of the file or memory, which will be mapped to the address space of the calling process.

File Mapping Functions

There are three functions that can be used to implement file or memory mapping:

(Code omitted)

File mapping object can be initiated by calling function ::CreateFileMapping(...). If we want to share a file, we need to pass the file handle to the first parameter (hFile) of this function. If we want to share a block of memory, we need to pass 0xFFFFFFFF to this parameter. The fourth and fifth parameters specify the size of the object. For file sharing, they can be set to zero, in which case the whole file will be shared. In the case of memory sharing, they must be greater than zero. Parameter lpFileMappingAttributes can be used to specify the security attributes of the object, in most cases we can assign zero to it and use the default attributes. Parameter flprotect specifies read and write permission. The most important parameter is the last one, which must be the address of buffers that contain a name assigned to the file mapping object. If any other process wants to access this object, it must also use the same name to create a view of the file mapping object.

After the file mapping object is created successfully, the owner (the process that created the object) can create a view of file to map the buffers to its own address space by calling function ::MapViewOfFile(...). When doing this, we must pass the handle returned by function ::CreateFileMapping(...) to parameter hFileMappingObject. If we pass 0 to parameters dwFileOffsetHigh, dwFileOffsetLow and dwNumberOfBytesToMap, the whole file or memory will be mapped. Finally, parameter dwDesiredAccess allows us to specify desired access right. This function will return a void type pointer, which could be cast to any type of pointer.

If any other process wants to access the file mapping object, it must call functions ::OpenFileMapping(...) and ::MapViewOfFile(...) to first access it then create a view of file. When calling function ::OpenFileMapping(...), it must pass the object name (specified by function ::CreateFileMapping(...) when the file mapping object was created) to parameter lpName. The buffers can be mapped to the address space of the process by calling function ::MapViewOffile(...), which is exactly the same with creating view of file for the owner of the object.

Samples

Samples 13.12\Send and 13.12\MsgRcv demonstrate how to share a block of memory between two applications. They are based on samples 13.8\Send and 13.8\MsgRcv respectively. First, the "Sender" application is modified so that its edit box will allow multiple line text input (When inputting the text, CTRL+RETURN key stroke can be used to start a new line), and the original variable CSenderDlg::m_nSent is replaced by CSenderDlg::m_szText, which is a CString type variable. The file mapping object is created in function CSenderDlg::OnInitDialog() as follows:

(Code omitted)

Here, BUFFER_SIZE is a macro defined as an integer in header file "Common.h". Also, MAPPING_PROJECT is a macro defined as a string. They will be used by both "Sender" and "MsgRcv". In message handler CSenderDlg::OnButtonSend(), before the message is sent out, we obtain the text from the edit box, create a view of file and put the text into the buffers. Then message MSG_SENDSTRING is sent to "MsgRcv":

(Code omitted)

In project "MsgRcv", the client window is implemented using edit view instead of original list view, this makes it easier for us to display text. After receiving the message, we open the file mapping object, create a view of file, then retrieve text from the buffers. Then we access the edit view, select all the text contained in the view, and replace the selected text with the newly obtained text. Finally, the acknowledge message is sent back:

(Code omitted)

There are other methods for sharing memory among different processes, such as DDE and OLE. Comparing to the two methods, file mapping method is relatively simple and easy to implement.

Summary

1) Before a window is created, we must stuff a WNDCLASS type object, register the window class name, and use this name to create the window. Class WNDCLASS contains useful information about the window such as mainframe menu, default icon, default cursor shape, brush that will be used to erase the background, and the window class name.

2) To implement one-instance application, we need to register our own window class name, and override function CWnd::PreCreateWindow(...). Before the window is created, we need to replace the default window class name with the new one. By doing this, we can implement one-instance application by searching for registered window class name: before registering the window class, we can check if there already exists a window that has the same class name. If so, the application simply exits.

3) We can call function CWnd::FindWindow(...) to find out a window with a specific class name or window name in the system.

4) The document/view structure is implemented by class CSingleDocTemplate or CMutiDocTemplate. If we want to create an application that does not use document/view structure, we need to eliminate the procedure of creating CSingleDocTemplate or CMutiDocTemplate type object and call function CWnd::Create(...) to create the mainframe window by ourselves.

5) We can create several CMultiDocTemplate type objects in an application to let it support multiple views or multiple documents.

6) Caption bar and window frame belong to non-client area. To paint non-client area, we need to handle messages WM_NCPAINT and WM_NCACTIVATE.

7) To create a window with transparent background, we need to specify style WS_EX_TRANSPARENT while creating the window.

8) An application can save its states in the system registry by calling function CWinApp::SetRegistryKey(...). The information can be saved and loaded by calling the following functions: CWinApp::WriteProfileInt(...), CWinApp::WriteProfileString(...), CWinApp:: GetProfileInt(...)., CWinApp::GetProfileString(...).

9) To exchange user defined messages between two different processes, we must use function ::RegisterWindowMessage(...) to register the messages.

10) Calling function CWnd::SetWindowPos(...) using parameter CWnd::wndTopMost

will make a window always stay on top of any other window.

11) Hook can be installed to let a process intercept and process Windows(messages before they reach destinations. There are several types of hooks, which include mouse hook, keyboard hook, journal record hook, journal playback hook, etc.

12) A hook can be installed by calling function ::SetWindowsHookEx(...) and removed by calling function ::UnhookWindowsHookEx(...).

13) A DLL does not have its own memory space, instead, its variables are mapped to the memory spaces of the calling processes. To declare static variables in the DLL, we need to specify a data segment by using #pragma data_seg macro and -SECTION link option.

14) To share a file or a block of memory among different processes, we need to create file mapping object. Any process that wants to access the memory must create a view of file, which will map the memory to its own address space.

BACK TO INDEX

Chapter 14 Views

We are going to introduce various types of views in this chapter. In MFC, there are several types of standard views that are supported by MFC classes. These views include edit view, rich edit view, list view, form view and tree view. The classes that can be used to implement them are CEditView, CRichEditView, CListView, CFormView and CTreeView respectively. They can be used to display plain text, formatted text, tree, list, etc.

14.1 Edit View

Edit view is most suitable for implementing plain text editor. Remember in Chapter 9 when building one-line text editor, we had to write a lot of code in order to add some basic editing functions. Actually, if we use edit view to implement text editor, we can build a notepad-like application within just few minutes. With the edit view, the serialization can be implemented by the member function of class CEditView, so we don't even need to write code for handling data loading and storing. Also, this class supports commands such as string search & replace, cut, copy and paste.

Generating the Application

A plain text editor can be generated through using Application Wizard. We can let the editor support files with ".txt" extension as we go through the steps of generating application skeleton (In order to do this, we need to click "Advanced" button in step 4, and input appropriate file extension into the edit box labeled with "File extension"), or we can edit IDR_MAINFRAME string resource after the application is created. We need to select CEditView as the base class of the view in the last step. By doing so, after the application is first compiled, we will have a very simple notepad-like application.

Sample 14.1\NotePad is generated this way. We even don't need to customize function CDocument:: Serialize(...) in order to support file I/O. Lets take a look at the default implementation of serialization:

```
void CNotePadDoc::Serialize(CArchive& ar)
```

```
{

((CEditView*)m_viewList.GetHead())->SerializeRaw(ar);

}
```

Everything is handled by function CEditView::SerializeRaw(...), which includes both data reading and writing.

Also, there is no need for us to write message handlers for Edit | Undo, Edit | Cut, Edit | Copy and Edit | Paste commands in order to enable them. If the mainframe menu contains the following IDs for these commands (which is the default feature for any project generated by Application Wizard), the application will automatically support the above commands:

(Table omitted)

The reason for this is that CEditView already maps commands with the above-mentioned IDs to its built-in member functions that handle undo, cut, copy and paste commands. The name of these functions are not documented in the current version of Visual C++, this means these function are not guaranteed to be supported in the future.

If we want to use other command IDs instead of the recommended ones, we need to implement command message mapping by ourselves. In order to do so, we need to look at the MFC source code that contains member functions of class CEditView, find out the function names that support these commands, and map the WM_COMMAND type messages to the appropriate functions.

Search Related Commands

In the sample, three other standard commands, Search | Find..., Search | Replace..., and Search | Find Next are implemented this way. The three commands are used for searching a specific string in the text contained in the view, replacing an old string with a new one, or repeating searching. The default IDs for these commands are listed in the following table:

(Table omitted)

Message UPDATE_COMMAND_UI is also handled for the above commands.

In the sample, we use following non-standard command IDs to implement find, replace, and repeat commands:

(Table omitted)

The message mapping is done in the implementation file "NPView.cpp":

(Code omitted)

With the above implementation, there is no need for us to declare and define new member functions to handle the above commands, everything will be handled automatically.

Other Commands

In the sample, some other commands that are not supported by class CEditVew are also implemented. These command include Edit | Delete, which can be used to delete the current selection; Edit | Select All, which can be used to select all the text contained in the window, and Edit | Time/Date, which can be used to insert a time/date stamp at the current caret position. For these commands, the message mapping and message handlers need to be implemented by ourselves.

Edit view is implemented by an embedded edit control, which can be accessed by calling function CEditView::GetEditCtrl(). Once this is done, we can call any member function of CEdit and make change to the text contained in the window. For example, if we want to replace the selected text with a new string or insert a string at the current caret position, we can call function CEdit::ReplaceSel(...) to do so. If we want to select all the text, we can call function CEdit::SetSel(...) and pass 0 and -1 to its first two parameters. If we want to delete the selected text, we just need to call function CEdit::Clear().

The following shows how command Edit | Time/Date is implemented in the sample:

(Code omitted)

First the current time is obtained by calling function CTime::GetCurrentTime(), which is stored in a CTime type variable. Then it is formatted and output to a CString type variable by calling function CTime:: Format(...). Finally, function CEdit::ReplaceSel(...) is called to insert the time stamp.

By now, our sample is almost the same with standard "Notepad" application. Obviously deriving class from standard MFC class saves us a lot of work. If we implement the file I/O and formatted text display by ourselves, we need to write a lot of source code.

14.2 Rich Edit View

Sample 14.2\Wordpad is an SDI application generated from Application Wizard, it will be implemented as a "Wordpad" application.

The edit view has very limited feature: it can store only text less than 64Kbytes, and it supports only one type of font. Furthermore, we can not edit graphics in the editor.

A more advanced editor can be implemented by using Rich edit view and Rich edit document. Two classes that support this type of view and document are CRichEditView and CRichEditDoc respectively. With the two classes, we can easily build a Wordpad-like application, which supports rich text format (one of the formats supported by Microsoft Word), OLE container and many default editing functions.

It seems that building a Wordpad-like application is very difficult, because it has too many features. However, with CRichEditView and CRichEditDoc classes, this procedure is almost equally simple with creating a "Notepad" application as we did in the previous section. To build a Wordpad-like application, we can start from Application Wizard to build a standard SDI application. To support ".rtf" file extension in our application (which is the default extension for rich edit format files), in step 4, we need to click "Advanced" button, input "rtf" in the edit box labeled with "File extension". In the final step, we also need to select CRichEditView as the base class for view implementation.

Generally, the base class for implementing application's document can not be customized. It will be derived from CDocument by default. However, if class CRichEditView is selected as base class for view implementation, the base class for document implementation will automatically be changed to CRichEditDoc. Also, some new commands will be added to the default mainframe menu IDR_MAINFRAME under "Edit" sub-menu (These new commands will be added by Application Wizard).

In order to include graphic objects (or other OLE server objects), we need to make the application an OLE container. By doing this, we can embed all OLE servers in the application. One example of such type of applications is "Paint". With this implementation, the application can be used to edit not only text, but also graphic objects. The OLE container selection can be set in the second step of Application Wizard. If we forgot this, we will be prompted to do so when button "Finish" is clicked in the final step.

By only working with Application Wizard, we are able to create a simple Wordpad. If we compile the application at this point, we will have a standard editor that allows us to insert various types of objects. The type of objects that can be embedded in the application depends on which OLE servers have registered in the system. To insert an object, we can execute command Edit | Insert New Object..., and select an object from the popped dialog box. If we select "Bitmap Image" object, the application will turn into a "Paint" application, with will let us edit bitmap in place.

Another feature of this editor is Edit | Paste Special... command. Because the editor supports not only text, but also graphic editing now, the paste command should support multiple-format data. If we execute command Edit | Paste Special..., a dialog box will pop up indicating the available data formats contained in the clipboard, from which we can make the selection. For example, if we paste a bitmap, we have the choice to paste it either as bitmap format, metafile format or DIB format.

The third feature of this editor is font selection. We may want to format different portion of the text using a different font. This is the default feature of class CRichEditView. The command ID that can be used to format the selected text using a specific font is ID_FORMAT_FONT (class CRichEditView supports this ID). So all we need to do is adding a Format | Font... command to menu IDR_MAINFRAME. With this simple implementation, we are able to format the text with any font that is available in the system.

Although it is very easy to build a fully functional application with a lot of enticing features, it is relatively difficult to make modifications. For example, the standard Wordpad application under Windows(has a ruler and a format bar, if we want to add these features, we need to add them by ourselves.

Customizing File Open Dialog Box

Class CRichEditView and CRichEditDoc can handle not only rich text format, but also plain text format (or ASCII format). By default, the two classes interpret input data using rich text format, if the format is different, the file will not be loaded. To let the application also support plain text format, we need to include multiple document types in "File Open" and "Save As" file dialog boxes, this gives the user an option for specifying file type. Although we can register more than one type of document to implement this, in the case of rich edit view, it is not an efficient way. This is because both formats are already supported by class CRichEditDoc.

To implement customized "File Open" dialog box, we can override function CWinApp::OnFileOpen(...). We need to provide "File Open" dialog box, let the user pick up a file name, and pass this name to function CWinApp::OpenDocumentFile(...). Here we need to pay special attention to file formats. Because we support more than one file format here, we need to implement a "File Open" dialog box supporting multiple file filters, and inform document the file format that was selected by the user. To customize a file open dialog box, we need the knowledge of Chapter 6; to let the document support a different file format, we can set a Boolean type member variable of class CRichEditDoc: CRichEditDoc::m_bRTF, which is a public member variable. If this variable is set to TRUE, the data in the file will be treated as formatted data stream; if it is set to FALSE, the data will be treated as unformatted data stream (plain ASCII text). We need to set this flag before function CWinApp::OpenDocumentFile(...) is called.

The following code fragment shows how function CWordPadApp::OnFileOpen() is implemented in the sample:

(Code omitted)

We must map command ID_FILE_OPEN to this function in order to make it effective. In the sample, WM_COMMAND message mapping for this command is customized as follows:

Original mapping:

```
ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
```

New mapping:

```
ON_COMMAND(ID_FILE_OPEN, OnFileOpen)
```

Customizing "Save As" Dialog Box

Besides file open, we also need to think about file saving. This is more complex than file open, because we need to allow the user to save the file being edited with a different format. In case the user changes data format, we must also change the original file extension (from "rtf" to "txt" or vice versa).

To customize file saving to support multiple file formats, we need to override function CDocument:: DoSave(...). This is an undocumented member function of MFC. Unfortunately, because file dialog box is implemented within this function, we have no other choice to support multiple file format without modifying it. Although using undocumented functions is not recommendable, sometimes we have to do so in order to make our applications perfect.

Function CDocument::DoSave(...) has two parameters:

```
BOOL CDocument::DoSave(LPCTSTR lpszPathName, BOOL bReplace);
```

The first parameter is a pointer to the buffers containing the file name, the second is a Boolean type variable indicating if the file name should be changed. Actually, this parameter is always set to TRUE so we can neglect its value.

Pointer lpszPathName gives us the file name that should be used for saving data. But this pointer can also be NULL. If the file being edited is created through File | New command and the user has selected File | Save or File | Save As command, lpszPathName will be NULL. The following table lists the values of lpszPathName and bReplace under different situations:

(Table omitted)

Based on the above analysis, we can override function CDocument::DoSave(...) as follows: obtaining the file name from lpszPathName, if it is NULL, we implement the "Save As" dialog box with multiple file filters. After the user has selected a file name, we need to add extension to it according to the filter selected by the user. Also, we need to set data format for file saving. Then we can call function CDocument:: OnSaveDocument(...) and pass the file name to it to implement file saving.

In the derived function CWordPadDoc::DoSave(...), we need to implement the customized "Save As" dialog box and also, add file extension, change file format if necessary. The following is a portion of this function:

(Code omitted)

We can compare CWordPadDoc::DoSave(...) with the default MFC function CDocument::DoSave(...).

Formatting Text

Another feature we want to let this editor have is text formatting. For example, we may let the user format the selected text using bolded, italic or underlined style. Or we may let the user change the alignment of the selected paragraph (make the whole paragraph aligned left, centered or aligned right). The two types of formatting are called Character Formatting and Paragraph Formatting respectively, they can be implemented through calling functions CRichEditView::SetParaFormat(...) and CRichEditView:: SetCharFormat(...). The following shows the formats of the two functions:

```
void CRichEditView::SetParaFormat(PARAFORMAT &pf);

void CRichEditView::SetCharFormat(CHARFORMAT cf);
```

Because there are many properties we can set, we need to use structures PARAFORMAT and CHARFORMAT to specify which properties will be customized. The following is the format of structure PARAFORMAT:

```
typedef struct _paraformat {

UINT cbSize;

_WPAD _wPad1;

DWORD dwMask;
WORD wNumbering;
WORD wReserved;
LONG dxStartIndent;
LONG dxRightIndent;
LONG dxOffset;
WORD wAlignment;
SHORT cTabCount;
LONG rgxTabs[MAX_TAB_STOPS];
} PARAFORMAT;
```

We need to set the corresponding bits of member dwMask in order use other members of this structure. For example, if we want to set paragraph alignment, we need to assign member wAllignment an appropriate value, and set PFM_ALIGNMENT bit of member dwMask. If this bit is not set, member wAlignment will have no effect when function CRichEditView::SetParaFormat(...) is called.

There are a lot of features we can set through using this function, which include text numbering (using bullets at the beginning of each line), paragraph start indent, right indent, second line offset, paragraph alignment and tabs.

The usage of function CRichEditView::SetCharFormat(...) is similar. Here we have another structure CHARFORMAT that could be used to set appropriate properties for the selected text:

```
typedef struct _charformat {
UINT cbSize;
_WPAD _wPad1;
DWORD dwMask;
DWORD dwEffects;
LONG yHeight;
LONG yOffset;

COLORREF crTextColor;

BYTE bCharSet;

BYTE bPitchAndFamily;

CHAR szFaceName[LF_FACESIZE];

_WPAD _wPad2;

} CHARFORMAT;
```

Again, member dwMask should be used to specify which properties will be customized. We can make change to character effects (make it bolded, italic, strikeout, underlined, or change its color), modify the size of characters, customize character's offset from the base line (this is useful for implementing superscript or subsript), or select a different type of font.

Counterpart functions of CRichEditView::SetParaFormat(...) and CRichEditView:: SetCharFormat(...) are CRichEditView::GetParaFormat(...) and CRichEditView:: GetCharFormat(...) respectively. They allow us to retrieve the properties of the current paragraph or the selected text (If no text is selected, the properties indicate the text at the current caret position). Similarly, we need to specify corresponding bits of member dwMask in order to retrieve certain properties: those members who have corresponding zero bits in member dwMask will not be stuffed with the paragraph or character information.

It seems that by using the above four functions, we can build a very useful editor that supports rich edit text format. However, in class CRichEditView, there exist more powerful functions that can be used to format the selected text or paragraph. These functions are also undocumented, but using them can save us much effort:

Functions used for paragraph formatting:

```
CRichEditView::OnParaCenter();

CRichEditView::OnParaRight();

CRichEditView::OnParaLeft();
```

Functions used for character formatting:

```
CRichEditView::OnCharBold();

CRichEditView::OnCharUnderline();

CRichEditView::OnCharItalic();
```

Functions used to handle UPDATE_COMMAND_UI messages:

```
CRichEditView::OnUpdateCharBold();

CRichEditView::OnUpdateCharUnderline();

CRichEditView::OnUpdateCharItalic();

CRichEditView::OnUpdateParaCenter();

CRichEditView::OnUpdateParaLeft();

CRichEditView::OnUpdateParaRight();
```

Instead of implementing our own message handlers, we can just add commands to the mainframe menu or tool bar, then map the commands to these functions. In the sample, we add six buttons for character and paragraphing formatting (Figure 14-1), and map the command messages to the above functions as follows:

(Code omitted)

With the above implementation, the editor can let the user set the character and paragraph properties.

14.3 Simple Explorer, Step 1: Preparation

Starting from this section we are going to create an Explorer-like application using classes CTreeView and CListView. The application is based on an SDI application whose client window is implemented by a 2-pane splitter window. We will use CTreeView to create the left pane, and use CListView to create the right pane. One the left pane, the file system (directories) will be displayed in a tree form, the user can click on any node to select a directory, or double click on it to expand the node (show all the sub-directories). On the right pane, all files and sub-directories contained in the currently selected directory will be listed, they can be displayed in one of the four styles supported by list view.

We have introduced how to create splitter window in Chapter 3. Obviously here we need to create static splitter windows. Application Wizard does have a choice to let us create splitter window, however, it can only help us with creating dynamic splitter window. We can modify the dynamic splitter window to static splitter window after the application is generated. To let the Application Wizard generate code for implementing dynamic splitter window, we can click "Advanced..." button (in step 4) and check "Use split widow" check box in the popped up dialog box.

In order to create a splitter window with two panes implemented by different types of views, first we must implement two view classes. Here, one of the views can be implemented as we go through the Application Wizard's project creation steps: in the final step, we can select CListView as the view's base class. The other class can be added after the project is generated by Class Wizard.

Sample 14.3\Explorer is created this way. It is a standard SDI application, with first view generated by Application Wizard whose name is CExplorerView. The second view is added by Class Wizard, whose base class is CTreeView and the class name is CDirView. In function CMainFrame::OnCreateClient(...), the splitter window is created using the above two classes:

(Code omitted)

Functions CSplitterWnd::CreateStatic(...) and CSplitterWnd::CreateView(...) are called to create the splitter window and each individual pane. Please note that we must include "afxcview.h" in the header files of both class CExplorerView and CDirView, otherwise the compilation will generate errors.

The only result of this sample is a two-way splitted window. We will add further features in the next several sections.

14.4 Simple Explorer, Step 2: List Drives

Sample 14.4\Explorer is based on sample 14.3\Explorer.

We will display four types of items in the tree view window (left pane of splitter window): desktop, computer, drives, directories. The root node is desktop node, and there will be only one such type of node. Under the desktop node, there will be a computer node, which lists the computer name. Under the computer node, all the available drives will be listed, under each drive node, directories will be listed. With this structure, the file system of the whole computer can be displayed.

Creating Image List

Each node can have a label and also an associated image. Although they both are optional features, implementing them can make our application look more professional. To use images, we must create an image list and select it into the tree control. The image list can be created from either DIB images or icons. As we know from Chapter 5, the simplest way to create image list is to prepare images as the resources then load them at run time. In the sample, five images are prepared for the tree control, whose usage is listed in the following table:

(Table omitted)

Like all other types of views, the best place to initialize the tree is in function CDirView:: OnInitialUpdate(). In order to do so, we need to create the image list, select the image list into the tree control, and create the tree. Image list creation can be implemented by calling functions CImageList:: Create(...) and CImageList::Add(...). We can use the first function to create the image list, specify the image size and number of images that will be included in the list. Then we can call the second function to add each single image. In the sample, this procedure is implemented as follows:

(Code omitted)

Alternative Ways of Creating Image List

The image list is created using bitmap images. It can also be created from icons. Also, we can use one single image to create an image list that contains several images. In order to do so, we need to combine all the images together to form one image (align them horizontally, just as the image used for creating tool bar), and call one of the following versions of function CImageList::Create(...):

BOOL CImageList::Create(UINT nBitmapID, int cx, int nGrow, COLORREF crMask);

BOOL CImageList::Create(LPCTSTR lpszBitmapID, int cx, int nGrow, COLORREF crMask);

The image list can also be created from two existing image lists by calling the following version of this function:

(Code omitted)

We set the background color to white so that all image's portion with white color will be treated as transparent region.

Setting Styles of Tree Control

We can set the styles of the tree control by calling function ::SetWindowLong(...). This function is not the member function of class CTreeView, and can be called to change the styles of any window. In order to know which styles can be customized, we can look at the documentation of function CTreeCtrl:: Create(...). Generally, any style that can be set to parameter dwStyle of this function can also be used by function ::SetWindowLong(...) to change the style of a tree control. The following code fragment shows how the default styles of the tree control are customized in the sample (within function CDirView:: OnInitialUpdate()):

(Code omitted)

Style TVS_HASLINES will let the nodes be connected by dotted lines, TVS_LINESATROOT will add a line at the root node, and TVS_HASBUTTONS will add a rectangle button (displays either "+" or "-") for each expandable node. If we do not specify these styles, the tree control will look slightly different.

These styles can also be customized in function CView::PreCreateWindw(...). In order to do so, we need to set the corresponding style flags for member dwExStyle of structure CREATESTRUCT. The difference between two methods is that using ::SetWindowLong(...) can let us change the styles of a window dynamically.

Adding Root Node

The next step is to add nodes to the tree. We need to call function CTreeCtrl::InsertItem(...) to add a node to the tree. To call this function, we need to prepare a TV_INSERTSTRUCT type object, and specify the properties of node. For example, we can specify the node's parent, associated image, label and states. This procedure has been discussed in chapter 5. One thing we need to pay attention to is that since InsertItem(...) is not a member function of CTreeView, we must first call function CTreeView::GetTreeCtrl() to obtain the tree control before adding any node. The following portion of function CDirView::OnInitialUpdate() shows how the root node is added:

(Code omitted)

Finding out Available Drives in the System

By stuffing TV_INSERTSTRUCT type object and calling function CTreeCtrl::InsertItem(...) repeatedly, the tree can be created. However, before proceeding to create other nodes, we need to find out all the available drives in the system.

Currently there can be at most 26 drives contained in one system, which are labeled from "A:" to "Z:". We can call runtime function _chdrive(...) to change the current working drive (Calling this function has the same effect with typing command "a:" in a DOS prompt window). Function _chdrive(...) has one parameter:

```
int _chdrive(int drive);
```

Parameter drive specifies target drive. It can be any number from 1 to 26, which represents drive A:, B:, C:... and so on. The function will return 0 if the working drive is changed successfully, otherwise it returns -1.

So we can call this function repeatedly by passing 1, 2, 3...26 to it and examining the returned value. If the function returns 0, this means the drive is available, and we need to add it to the tree control. If the function returns -1, we can just go on to check the next drive.

Because we do not want to change the default working drive, we need to save the current working drive before calling function _chdrive(...), and resume it after the checking is over. The current working drive can be retrieved by calling runtime function _getdrive(). The following portion of function CDirView:: OnInitialUpdate() shows how the drives are added to the tree view window:

(Code omitted)

When program exits, we must do some cleanup job, which includes removing all the nodes and destroying the image list. In the sample, this is implemented in WM_DESTROY message handler:

(Code omitted)

14.5 Simple Explorer, Step 3: Listing Directories

Sample 14.5\Explorer is based on sample 14.4\Explorer.

In the previous section, we called function _chdrive(...) to find out all the available drives in the system. In order to add the directories (sub-directories) to the tree view window, we need to enumerate directories and files.

Enumerating Files and Directories

In MFC, class CFileFind can be used to enumerate files and directories under certain path. If we call this function for all the directories and sub-directories contained in the system, finally we will get all the information about the file system.

To enumerate files and directories contained in the current working directory, we can start from calling function CFileFind::FindFile(). Then we can call function CFileFind::FindNextFile() repeatedly until it returns a FALSE value. The following code fragment shows how to enumerate all the files and directories contained in the current working directory:

(Code omitted)

Note we can also use wildcard characters when calling function

CFileFind::FindFile(...) to match file name with specific patterns. If we do not pass any parameter to it, it will be equal to passing "*.*" to the function. In this case all the files and directories will be enumerated. If function CFileFind:: FindNextFile() returns a non-zero value, we can call several other member functions of class CFileFind to obtain the properties of the enumerated file such as file name, file path, file attributes, created time and updated time.

Adding Directory Nodes

In order to add directory node to the tree view window, we need to implement a loop and enumerate directories for each available drive, then add the enumerated directories to the tree.

Besides directories, we also need to enumerate sub-directories for the expanded nodes. This is because our tree control supports buttons (we have set TVS_HASBUTTONS style), which indicates whether an item has child items or not. For a node that contains child items, its button will contain a "+" sign when the node is in collapsed state, indicating that the node is expandable. For a node that does not contain child items, there will be no such type of buttons.

So the directory enumeration can be added to the loop of enumerating all drives: after an existing drive is found, we can change the current working directory to the root directory of this drive, then enumerate all the first-level directories and all the sub-directories of each first-level directory.

Function CDirView::AddDirs(...) is implemented in the sample, it will be used to add directory items to a specified node. It has two parameters, the first is the handle of the target tree item, and the second is a Boolean type variable indicating if we should further add sub-directories for each added directory node. The following is the format of this function:

```
void CDirView::AddDirs(HTREEITEM hTreeItem, BOOL bFindChild);
```

Before calling this function, we need to change the current working directory to the directory we want to examine. So in function CDirView::OnInitialUpdate(...), after one drive node is added to the tree view window, we change the current working directory to root directory of that drive, and call function CDirView::AddDirs(...) to add nodes for the directories. The following is the modified portion of function CDirView::OnInitialUpdate(...):

(Code omitted)

For the root directory, we need to find out not only the directories under it, but also the sub-directories of each first-level directory. So we pass a TRUE value to the second parameter of function CDirView:: AddDir(...). The function will recursively enumerate sub-directories for all the directories found within the function if the parameter is TRUE.

At the beginning of function CDirView::AddDirs(...), we initialize a TV_INSERTSTRUCT type object and call function CFileFind::FindFile(). If it returns TRUE, we can further call function CFileFind:: FindNextFile() and get all the attributes of the enumerated file (directory). Then we repeat file (directory) enumerating until function CFileFind::FindNextFile() returns a FALSE value:

(Code omitted)

We can examine if the enumerated object is a directory or a file by calling function CFileFind:: IsDirectory(). This is necessary because only the directories will be added to the tree view window. A directory node is added by first stuffing TV_INSERTSTRUCT type object then calling function CTreeCtrl:: InsertItem(...):

(Code omitted)

If parameter bFindChild is TRUE, we need to enumerate sub-directories for each added directory node. However, since "." and ".." are also two types of directories (indicating the current and parent directories respectively), if we apply this operation on them, it will cause infinite loop. To examine if a directory is one the above two types of directories, we can call function CFileFind::IsDots(). If the function returns FALSE, we can call function CDirView::AddDirs(...) again to add sub-directory nodes. Before calling this function, we also need to change the current working directory. After the function is called, we need to resume the original working directory:

(Code omitted)

If we execute the sample, we need to wait for a while before the procedure of building directory tree is completed. This waiting time is especially long for a system containing many drives and directories. This is why we only add fist and second level directories to the tree view window at the beginning. If we build the whole directory map before bringing up the window, the user will experience a very long waiting time. We will add new nodes to the tree only when a node is expanded and its sub-level contents need to be revealed.

14.6 Simple Explorer, Step 4: Displaying Files

In the sample, all files will be listed in the list view that is located at the right pane of the client splitter window. Like what is implemented in class CDirView, we need to enumerate files under a directory and add corresponding nodes to the list control in order to display the files.

The list view window will display all the directories and files contained in the selected directory. If the currently selected directory changes, we must destroy the list view and create a new one. For this purpose, in the sample, a new function CExplorerView::ChangeDir() is implemented, which can be used to create the list view from the currently selected directory.

Image Lists

Before adding any file to the list view, we need to prepare image lists. This procedure is almost the same with that of tree view. The only difference between the two is that for list view we have more choices. This is because the items contained in a list view can be displayed in different styles, and for each style we can use a different type of images.

A list view can display items in one of the four styles: big icon (default style), small icon, list, report. We can prepare two image lists, one for big icon style, one for other three styles.

We can display different file types using different icons, this is how the files are displayed in real "Explorer" application. Under Windows(, each type of files can register both big and small icons in the system, and "Explorer" will use the registered icons for file displaying. To get the registered icons, we need to call some special functions. We will implement this method in later sections. Here, we will prepare our own icons for displaying files. In the sample, two sets of image resources are included in the applications, one of them will be used for displaying directories and the other for displaying files. Their IDs are IDB_BITMAP_CLOSEFOLDERBIG, IDB_BITMAP_CLOSEFOLDER, IDB_BITMAP_FILEBIG and IDB_BITMAP_FILE.

In the sample, big icon image list is created from IDB_BITMAP_CLOSEFOLDERBIG and IDB_BITMAP_FILEBIG. Small icon image list is created from IDB_BITMAP_CLOSEFOLDER and IDB_BITMAP_FILE. The creation of image list is the same with what we did for the tree view. When an image list is selected into the list control, we must specify the type of image list. The following portion of function CExplorerView::ChangeDir() shows how the image lists are selected into the list control in the sample:

(Code omitted)

Here pointer pilSmall and pilNormal point to two different image lists. We use LVSIL_SMALL and LVSIL_NORMAL to specify the type of the image list.

Adding Columns

First we need to add columns to the list control. The columns will appear in the list control window when the items contained in it are displayed in "Report" style. For each item, usually the small icon associated with the item and item label will be displayed at the left most column (column 0). For other columns, we can display text to list other properties of the item.

The columns are added through stuffing LV_COLUMN type object and calling function CListCtrl:: InsertColumn(...). Like other structures such as TV_INSERTSTRUCT, LV_COLUMN also has a member mask that lets us specify which of the other members of this structure will be used. For example, we can specify text alignment format (is the text aligned left, right or is it centered?) by setting LVCF_FMT bit of member mask and assigning appropriate value to member fmt; we can specify the width of each column by setting LVCF_WIDTH bit and using cx member; we can set the column caption by setting LVCF_TEXT bit and using pszText member. In the sample, text of each column is aligned left, the width of each column is set to 150, and the column texts are: "Name", "Size", "Type", and "Modified" respectively.

To make it convenient to add columns, the following global variables are declared in the sample:

```
#define NUM_COLUMNS 4
```

(Code omitted)

In function CExplorerView::ChangeDir(), the columns are added as follows:

(Code omitted)

Listing Files

In the list view, each item represents a file under certain directory. When the items are displayed in "big icon", "small icon" and "list" styles, each file is represented by an icon contained in the list view window. When they are displayed in the "report" style, the file is represented by both an icon and several text strings. In this case, column 0 contains icon and the file name, and the rest columns contain other information about the file (These items are called the sub-items).

The procedure of adding items to list control is similar to adding directory nodes to tree control, except that we don't need to worry about enumerating sub-directories here. Also, for each item, we need to set not only the image number and item text (contained in column 0), but also the sub-item text (contained in the rest of the columns). For this purpose, we can store the text of sub-items in a string array. After all the items are added, we can set sub-item text for each item.

The file enumerating can be implemented by calling functions CFileFind::FindFile() and CFileFind::FindNextFile() repeatedly. After a file is found, we stuff an LV_ITEM type object and call CListCtrl::InsertItem(...) to add a new item to the list control. Here is how it is implemented in function CExplorerView::ChangeDir():

(Code omitted)

Unlike tree control, there is no handle here to identify a special item. All the items are identified by their indices, this means if we display items in "list" or "report" style, the item located at the first row is item 0, the next row is item 1... and so on. When inserting an item, we need to specify the item index by using member iItem of LV_ITEM structure.

We store file size (for directory, display nothing), file type ("File" or "Folder"), the updated time in a string array that will be used to add text for the sub-items. These attributes of file can be retrieved by calling functions CFileFind::GetLength(), CFileFind::IsDirectory() and CFileFind:: GetLastWriteTime(...). When calling the third function to obtain the update time of a file, we get a CTime type variable. To store the time in a CString type variable in ASCII format, we need to call function CTime::Format(...). The following portion of function CExplorerView::ChangeDir() shows how the string array is created:

(Code omitted)

The text of sub-items is added by calling function CListCtrl::SetItemText(...). This can also be implemented by stuffing LV_ITEM type object (specifying item and sub-item indices) and calling function CListCtrl::SetItem(...). The following portion of function CExplorerView::ChangeDir() shows how this is implemented in the sample:

(Code omitted)

Destroying the Old List

Whenever the current working directory is changed, we need to call function CExplorerView:: ChangeDir() to create new file list. Before building a new one, we need to delete the old list. In the sample, this is implemented by function CExplorerView::DestroyList(). Within this function, both list items and image lists are deleted:

(Code omitted)

Since we can retrieve the pointers of image list from the list control, there is no need for us to store them as variables. This function is called in function CExplorerView::ChangeDir() and WM_DESTROY message handler.

Using Function CExplorerView::ChangeDir()

At this point, we still do not allow the user to select a directory by clicking on a directory node in the tree view window. So we can only display the files and directories contained in the root directory when the application is first invoked. This is implemented in function CExplorerView::OnInitialUpdate(), where we find the first available drive, and call function CExplorerView::ChangeDir() to create the list view:

(Code omitted)

14.7 Simple Explorer, Step 5: Displaying Registered Icons

Windows(encourages all types of files to register specific icons to the system, so that when they are displayed in certain applications such as "Explorer", the registered icons (also called Shell Icon) can be used to distinguish between different type of files. However, some file types do not have registered icons and some files contain icons within themselves (such as files with ".exe" or ".dll" extension).

Which Icon to Use

Windows(always try to display a file using the appropriate icons. If a file contains icon itself, this icon will be used. If a file doesn't contain any icon but has registered icons (such as some special document files like "*.bmp", "*.doc"), the registered icons will be used. If no registered icons are found, a default icon will be assigned to the file.

There is a shell function that can be used to retrieve the icon information for a file:

```
WINSHELLAPI DWORD WINAPI SHGetFileInfo

(

LPCTSTR pszPath, DWORD dwFileAttributes, SHFILEINFO FAR *psfi,

UINT cbFileInfo, UINT uFlags

);
```

Here parameter pszPath is a pointer to a string specifying the file path; dwFileAttributes specifies the file attributes, and the file information can be retrieved into a SHFILEINFO type object which is pointed by pointer psfi; cbFileInfo specifies the size of SHFILEINFO structure; uFlags specifies what information is being retrieved. In our case, we can combine SHGFI_ICON with one of the following flags and pass the result to parameter uFlags:

(Table omitted)

To display each file with embedded or registered icons, before adding an item to the list control, we need to first customize the image list. If any icon is found by calling function ::SHGetFileInfo(), we will add it to the image list. If we could not find an icon using this method, the default icon will be associated with the corresponding file.

Sample

Sample 14.7\Explorer is based on sample 14.6\Explorer. In this sample, the embedded and registered icons are retrieved for displaying files in the list view.

In the sample, a new member function is added for retrieving icons for a file:

HICON CExplorerView::GetIconFromFile(CString szFileName, UINT uFlags);

The returned value is an icon handle. Within this function, ::SHGetFileInfo(...) is called to get the icon information of a file. The following is the implementation of this function:

(Code omitted)

Function CExplorerView::ChangeDir() is modified as follows: after a file is found, function CExplorerView::GetIconFromFile(...) is called to find its embedded or registered icons; if this is successful, the newly obtained icons will be added to the image list and associated with the file; otherwise the default images will be used. The following portion of function CExplorerView::ChangeDir() shows how we try to find the embedded icons of a file:

(Code omitted)

In rare cases, some files may have small embedded or registered icon but no corresponding big icon, or vice versa. In any case, the embedded or registered icon has the highre priority to be used. The newly obtained icon is added to the image list by calling function CImageList::Add(...).

14.8 Simple Explorer, Step 6: Clicking and Double Clicking

Sample 14.8\Exoplorer is based on sample 14.7\Explorer. In this sample, when the user clicks or double clicks the left button on a directory node contained in the tree control, the current working directory will be changed and the contents of the list view will also be updated.

If the double clicking expands a node, we need to pay attention to the newly revealed nodes: if any directory contains sub-directories, we need to add new nodes so that the node button will be automatically enabled.

Tree Control Messages

Mouse clicking events are sent through WM_NOTIFY messages. For tree control, various activities of tree items can be handled by processing this message. In MFC, this message can be mapped to a member function by using macro ON_NOTIFY_REFLECT. This macro has two parameters, the first specifies the event type, the second specifies the member function name. There are many types of events, for example, mouse button clicking event is defined as NM_CLICK, and the node expanding event is defined as TVN_ITEMEXPANDING. Fortunately, in MFC, message mapping for these events can be easily implemented through using Class Wizard. In sample 14.8\Explorer, we trap mouse clicking and node expanding events to functions CDirView:: OnClick(...) and CDirView::OnItemExpanding(...) respectively.

Obtaining Full Path

When the user clicks mouse on a node, first we need to find out the path represented by that node. Although the directory name is stored as the item text for each node, it is not a full path. We need a full path in order to change the current working directory. In the sample, function CDirView::GetDir(...) is implemented to obtain the full path represented by any item. Within this function, we keep on retrieving the item's parent node until root is reached, and combining the obtained directory names to form a full path.

Finding out the Clicked Item

After receiving NM_CLICK notification, we can call function CTreeCtrl::HitTest(...) to find out the handle of the item that was clicked. We need to pass the current mouse position to this function. The returned value should be the handle of the item that is currently under the mouse cursor. If the mouse is not over any item, the function will return NULL. Please note that when calling this function, the coordinates of the mouse cursor should be in the coordinate system of the client window. We need to call function CWnd::ScreenToClient(...) to make the conversion.

When an Item Is Clicked

Special attention needs to be paid to directory nodes labeled with "." and "..". When they are clicked, the current working directory should be changed differently. For the purpose of demonstration, in the sample, no change will be made if the user clicks any of the two types of directories. If the user clicks a normal directory node, we need to change the current working directory to the selected one and notify the list view to update its contents. This notification is made through calling function CBrowserDoc::ChangePath(), within which function CBrowserView::ChangeDir() is called. However, if the user clicks on the currently selected directory, no change will be made. The following is the message handler for mouse left button clicking:

(Code omitted)

When a Node Expands

When a node is expanding, we need to check the newly revealed nodes to see if they always contain child nodes. If not, we need to add child nodes (if they have sub-directories) to them because this will enable node button automatically. In the sample, function CDirView::AddChildrenChildren() is implemented to let us add new nodes for all the child nodes of a given node. Within this function, we check each child node to see if it already has child items (which means the sub-directories have already been added for this node). If not, we call function CDirView::AddDirs(...) to add new nodes to it. A node's child item can be enumerated by calling function CTreeCtrl::GetChildItem(...) first then calling CTreeCtrl::GetNextSiblingItem(...) repeatedly until it returns NULL value. Also, we can call function CTreeCtrl::ItemHasChildren(...) to examine if a node already has child nodes. The following is the implementation of function CDirView::AddChildrenChildren(...):

(Code omitted)

This function is called when a node is about to expand in function CDirView::OnItemExpanding(...) as follows:

(Code omitted)

To make the application more user friendly, the rest part of this function swaps the directory node image from the image representing open directory (IDB_BITMAP_OPENFOLDER) to the one representing closed directory (IDB_BITMAP_CLOSEFOLDER) when the node is collapsing and vice versa when it is expanding.

14.9 Simple Explorer, Step 7: File Sort

Sample 14.9\Explorer is based on sample 14.8\Explorer, it implements file sorting.

When the list items are displayed in "Report" style, one thing we can implement is to sort all the files by different attributes. For example, if we click on "Name" column, all the files should be sorted by their names; if we click on "Size" column, all files should be sorted by their sizes; if we click on "Type" column, all the files should be sorted by their extensions; if we click on "Updated" column, all the files should be sorted by their updated dates and times.

Sort Related Functions

We can call function CListCtrl::SortItems(...) to implement item sorting. This function has two parameters:

```
BOOL CListCtrl::SortItems(PFNLVCOMPARE pfnCompare, DWORD dwData);
```

The function's first parameter is a little special, which is the pointer to a callback function provided by the programmer. The callback function will be used to perform actual comparison. This is because when comparing two items, class CListCtrl has no way of knowing which item should precede the other. In order to provide our own rules of making comparison, we need to implement the callback function.

The callback function has the following format:

int CALLBACK CompareFunc(LPARAM lParam1, LPARAM lParam2, LPARAM lParamSort);

In order to compare two items, we need to provide each item with a parameter, which is an LPARAM type value. When two items are compared, their parameters will be passed to the callback function, which will return different values indicating which item should precede the other. If the first item (whose parameter is lParam1) should precede the second item (whose parameter is lparam2), the function needs to return -1; if the first item should follow the second item, the function needs to return 1; if the two items are equal, the function needs to return 0.

When calling function CListView::SortItems(...), we can pass different pre-defined values to parameter dwData, which will be further passed to parameter lParamSort of the callback function. This provides us with a way of specifying different types of sorting methods.

Adding Parameters to Items

By now, when creating an item, we did not specify any parameter for it. Actually, any item in the list control can store a 32-bit parameter which can be used to distinguish one item from another. Of course the item index can also be used for this purpose. However, since the relative positions of two items can change frequently, the index of an item is also not fixed. Since the only information passed to the callback function about an item is its parameter, we must make it unique for any item.

In the sample, function CBrowserView::ChangeDir() is customized as follows: when a new item is added to the list control, we set its parameter to its initial index and use it as the identification of this item. This parameter will not change throughout its lifetime:

(Code omitted)

Functions Implementing Comparisons

Four static member functions are implemented for doing different types of comparisons:

(Code omitted)

Actually, in the callback function, one of the above functions is called to perform the comparison according to parameter lParamSort:

(Code omitted)

Please note that the callback function must be either a global function or a static member function. So within it we cannot call CListView::GetListCtrl() directly to obtain the list control. Instead, we must first obtain the current instance of list view, then use it to call function CListView::GetListCtrl() and obtain the list control. This is why at the beginning of the callback function the current active document is first obtained, from which the current active list view (and the list control) is obtained.

Using Parameter to Find an Item

Within the function that implements comparison, the only information we know about an item is its parameter. This is not enough for making comparison. We need to obtain the item and get more information (In our sample, this includes file name, extension, type, and updated time) before proceeding to compare the two items.

An item can be obtained from its parameter by calling function CListCtrl::FindItem(...). In order to call this function, we need to stuff a LV_FINDINFO type object specifying what information is provided for item searching. To search an item by its parameter, we need to set LVFI_PARAM bit of member flags of the structure, and assign the parameter to member lParam. In the sample, a static member function CExplorerView::FindItem(...) is implemented, it can be called from any static member function to find an item using its parameter:

(Code omitted)

Here a LV_FINDINFO type object is stuffed, with LVFI_PARAM bit of member flags set to "1" and the item parameter assigned to member lParam. Then the object is passed to function CListCtrl:: FindItem(...) to search the item in the list control. Function CExplorerView::FindItem(...)'s second parameter is a CListCtrl type reference, this is because within static member function, we must use the instance of an object to call any of its non-static functions. This function returns the current index of the corresponding item.

Comparing Two Items by File Names

The procedure of comparing two items is described in the following paragraphs.

First we pass the parameters of the items to function CExplorerView::FindItem(...) to retrieve their current indices. After the indices are obtained, we stuff a LV_ITEM type object, set LVIF_IMAGE bit of member mask to "1" and call function CListCtrl::GetItem(...). Since in the sample, a directory item is always associated with the default image (In the image list, the image index is 0), we can use an item's image index to tell if it represents a directory or a file. For different situations, the comparing function will return different values (In the sample, a directory always preceeds a file item):

(Code omitted)

In case both items are directories or files, we need to further compare their names. Since file names under Windows(are case insensitive, we neglect character case when performing the comparison. The comparison is done within a for loop, which starts from the first characters and ends under one of the following situations: 1) The two compared characters are different, in which case the character that has the greater value belongs to the item that should follow the other. 2) One of the strings reaches its end. In this case the item with longer file name should follow the other. If two strings are exactly the same, the function returns 0:

(Code omitted)

Notification LVN_COLUMNCLICK

When the user clicks on one of the columns, the list control sends a notification message LVN_COLUMNCLICK to its parent window. If we want to handle this message within the list view, we need to use macro ON_NOTIFY_REFLECT to map the message to one of its member functions. In the sample, this message mapping is added through using Class Wizard:

(Code omitted)

Within message handler CExplorerView::OnColumnClick(...), function CListCtrl::SortItems(...) is called to perform file sorting:

(Code omitted)

14.10 Using Form View

Form view is easy to use because the procedure of implementing it is similar to that of a dialog box. We can start from building a dialog template, then adding common controls to the template. Generally everything we can implement in a dialog box can also be implemented in the form view. The difference between the two is that when creating dialog template for the form view, we must set its style to "child" and "no border" (Figure 14-2).

Both tree control and list control can be implemented in a form view. Sample 14.10\Explorer is based on sample 14.9\Explorer whose left pane of the splitter window is implemented by a form view. Within the form view, a tree control is implemented for displaying directories. We will see, it is almost the same to use tree control in a form view with using a tree view directly.

New Class and Dialog Template

We must add a new view that is based on class CFormView. This can be implemented by using Class Wizard. Because a form view must be associated with dialog template, before adding the new class, we need to add a dialog template (Of course, we can first generate the class then the dialog template, and change the ID contained in the class to the ID of the dialog template later). When creating dialog template, we need to delete all the default controls, and customize its style to "Child" and "No border". Then we can add a tree control to the template, set the following styles: "Has button", "Has lines", "Line at root". This equals to calling function ::SetWindowLong(...) and setting styles TVS_HASLINES, TVS_LINESATROOT, TVS_HASBUTTONS for the tree control.

When generating the new class, we can associate the ID of the dialog template to it. Then, we can add a control variable for the tree control. By doing this, the tree control can be accessed by directly referring to this variable instead of calling function CWnd::GetDlgItem(...). In the sample, the newly generated class is named CDirFormView, the ID of the dialog template is IDD_DIALOG_VIEW, the ID of the tree control is IDC_TREECTRL, and the variable added for the tree control is CDirFormView::m_tcDir.

Implementing New Member Functions

We can just copy all the member functions from CTreeView and implement them in CDirFormView. The only difference is that we must replace all function calls of CTreeView::GetTreeCtrl() by m_tcDir. Also, we can implement a GetTreeCtrl() function in class CDirFormView and let it return the reference of m_tcDir. By doing this, we don't need to change anything else.

The following variables and functions are declared in class CDirFormView, they are implemented exactly the same as in class CDirView:

```
char CDirFormView::m_szPath[_MAX_PATH];

CString CDirFormView GetDir(HTREEITEM);

void CDirFormView AddDirs(HTREEITEM, BOOL);
```

void CDirFormView::AddChildrenChildren(HTREEITEM);

void CDirFormView::OnInitialUpdate();

afx_msg void CDirFormView::OnClickTreeCtrl(NMHDR* pNMHDR, LRESULT* pResult);

afx_msg void CDirFormView::OnItemExpandingTreeCtrl(NMHDR* pNMHDR, LRESULT* pResult);

Resizing Tree Control

Since the controls contained in the form view does not get resized or repositioned automatically, we need to move or resize them after receiving message WM_SIZE. This will make the form view better balanced.

In the sample, function CDirFormView::ResizeTreeView() is implemented to resize the tree control. After this function is called, the tree control will be resized so that it just fits within the form view window (A border is left around the tree control). The following is the implementation of this function:

(Code omitted)

Mouse Cursor Coordinates

Because the dimension of the tree control is not the same with that of the form view, when function CTreeCtrl::HitTest(...) is called in response to notification NM_CLICK, we need to convert the coordinates of mouse cursor from form view window to the tree control window. The following code fragment shows how function CTreeCtrl::HitTest(...) is called when mouse's left button is pressed:

(Code omitted)

Replacing CDirView with CDirFormView

Now we can replace class CDirView with CDirFormView when creating the splitter window. This is fairly simple, all we need is to use class CDirFormView to create the left pane of the splitter window in function CMainFrame::OnCreateClient(...):

(Code omitted)

This new version of Explorer behaves exactly the same with the previous one. However, with form view, we can add other common controls such as buttons to the left pane. This will give us more flexibility in improving our application.

Summary

1) A standard text editor can be implemented by class CEditView. This class contains some member functions that can be used to implement a lot of useful commands:

i) Serialization can be implemented by function CEditView::SerializeRaw(...).

ii) Undo, Cut, Copy and Paste commands can be implemented by the following undocumented functions: CEditView::OnEditUndo(), CEditView::OnEditCut(), CEditView::OnEditCopy(), CEditView::OnEditPaste().

iii) String search related commands can be implemented by the following undocumented functions: CEditView::OnEditFind, CEditView::OnEditReplace, CEditView::OnEditFindRepeat().

2) Class CRichEditView and CRichEditDoc support formatted text editing. The classes support two file formats: "rtf" format and plain text format. There is a member variable contained in class CRichEditDoc: CRichEditDoc::m_bRTF. If we want to open "rtf" type files, we need to set this variable to TRUE. If we want to edit plain ASCII text, we need to set this variable to FALSE.

3) To get or set the format of a paragraph, we can stuff structure PARAFORMAT and call function CRichEditView::GetParaFormat(...) or CRichEditView::SetParaFormat(...); to get or set the format of characters, we can stuff structure CHARFORMAT and call function CRichEditView::SetCharFormat(...) or CRichEditView::SetCharFormat(...).

4) To customize "File Open" dialog box, we need to override function CWinApp::OnFileOpen(...). To customize "Save As" dialog box, we need to override undocumented function CDocument::DoSave().

5) The following undocumented member functions can be used to implement commands for formatting characters or paragraph in a rich edit view:

CRichEditView::OnParaCenter();

CRichEditView::OnParaRight();

CRichEditView::OnParaLeft();

CRichEditView::OnCharBold();

CRichEditView::OnCharUnderline();

CRichEditView::OnCharItalic();

6) The styles of tree view and list view can be set by calling function ::SetWindowLong(...). Any style that can be used in function CTreeCtrl::Create(...) or CListCtrl::Create(...) can be changed dynamically by using this function.

7) We can call function _chdrive(...) to test if a drive (from drive A to drive Z) exits in the system. If the function returns -1, the drive does not exit. If it returns 0, the drive is available. We can also call this function to change the current working drive.

8) Class CFileFind can be used to enumerate all the files and directories under certain directory. To enumerate files and directories, we can call function CFileFind::FileFind() first then call function CFileFind::FindNextFile() repeatedly until it returns FALSE.

9) Function CFileFind::IsDirectory() can be used to check if an enumerated object is a directory. Function CFileFind::IsDot() can be used to check if a directory is "." or "..".

10) Function ::SHGetFileInfo(...) can be used to obtain the embedded or shell icons for a file.

11) Notification NM_CLICK can be used to trap mouse clicking events on the tree control. Notification TVN_ITEMEXPANDING indicates that a node is about to expand.

12) When the user clicks mouse on the tree control, we can call function CTreeCtrl::HitTest(...) to find out the handle of the item that was clicked.

13) We can call function CListCtrl::SortItems(...) to implement item sorting in the list control. In order to do this, we must assign each item contained in the list control a parameter, which will be used as the identification of the item. Then we need to prepare a callback function, within which rules of comparison are implemented.

14) To search for a specific item by its parameter, we can stuff structure LV_FINDINFO and call function CListCtrl::FIndItem(...).

15) To respond to the mouse clicking events on the columns of the list control, we need to trap notification LVN_COLUMNCLICK.

BACK TO INDEX

Chapter 15 DDE

Dynamic Data Exchange (DDE) is one of the ways to exchange information and data among different applications. From samples of previous chapters we already have some experience on implementing registered message and file sharing, which allow one process to send simple message or a block of data to another process. DDE is another way of implementing data exchange, it can handle more complicated situation than the other two methods.

DDE is constructed under client-server model: a server application exposes its services to all the applications in the system; any client application can ask the server for certain type of services, such as getting data from server, sending data to the server, asking the server to execute a command. The best way to implement DDE in an application is to use Dynamic Data Exchange Management Library (DDEML), which is supported by the Windows(. With this library, the implementation of DDE becomes easy. All the samples in this chapter are based on this library.

MFC does not have a class that encapsulates DDE, so we have to call all the functions in the DDEML.

15.1 DDE Registration

To support DDE functions, every application (server or client) must implement the following: 1) Initialize DDE. 2) Provide a callback function that will be used to handle DDE messages. 3) Uninitialize DDE when the application exits (or when DDE functions are no longer supported). For the server application, it needs additional two steps: 1) Register name services after the DDE is initialized. 2) Unregister the name services before DDE is uninitialized.

DDE Initialization, Uninitialization, Service Registration, Unregistration

DDEML provides functions for everything described above. The following discusses functions contained in the DDEML that can be used for these purposes:

DDE initialization:

```
UINT ::DdeInitialize

(

LPDWORD pidInst, PFNCALLBACK pfnCallback, DWORD afCmd, DWORD ulRes

);
```

An application must implement a DWORD type variable for storing its instance identifier, which is somehow similar to application's instance handle. When a server is making conversation with a client, they both must use their instance identifiers to verify that the message is directed to them. This unique instance ID is obtained through calling function ::DdeInitialize(...). When calling this function, we need to pass the address of the DWORD type variable to its first parameter, and the variable will be filled with the instance ID.

The second parameter is the pointer to a callback function, which will be discussed later.

The third parameter is the combination of different flags, which can be used to specify what kind of DDE messages we want to receive. This is useful because there are a lot of services provided by the DDE model. Sometimes we do not want certain types of messages to be sent to our applications. In this case, we can pass parameter afCmd a combination of filter flags, which will allow only certain type of messages to be sent to the application. The following table shows some examples:

(Table omitted)

DDE uninitialization:

```
BOOL ::DdeUninitialize(DWORD idInst);
```

The only parameter of this function is the value obtained from function ::DdeInitialize(...).

DDE name service registration and unregistration:

```
HDDEDATA ::DdeNameService(DWORD idInst, HSZ hsz1, HSZ hsz2, UINT afCmd);
```

Again, idInst is the DDE instance ID which is obtained from function ::DdeInitialize(...). The second parameter is the handle of the name service string, which is a new type of variable. Actually, it is just a new type of handle that can be obtained by calling another function supported by DDEML.

In DDE, data is exchanged through sending string handles among different applications. For example, if we have a string that is contained in a series of buffers, we cannot send the buffers' starting address to another process and let the string be accessed there. To let the string be shared by other DDE applications, we need to create a string handle and let it be shared by other applications. With the handle, any application can access the contents contained in the buffers. In DDEML, following two functions can be used to create and free string handle:

```
HSZ ::DdeCreateStringHandle(DWORD idInst, LPTSTR psz, int iCodePage);
```

```
BOOL ::DdeFreeStringHandle(DWORD idInst, HSZ hsz);
```

For the first function, we can pass the string pointer to parameter psz, and the function will return a string handle. When the string is no longer useful, we need to call function ::DdeFreeStringHandle(...) to free the string handle.

When registering a service name, we must use a string handle rather than the service name itself.

A service name is the identifier of the server that lets the client applications find the server when requesting a service. We can use any string as the service name so long as it is not identical to other DDE service names present in the system. When a client requests for services from the server, it must obtain a string handle for the service name and use it to communicate with the server.

DDE Callback Function

The DDE callback function is similar to the callback functions implemented for common dialog boxes and hooks, except that the situation is more complicated here. A DDE callback function has 8 parameters, which may have different meanings for different messages. The basic form of the callback function is as follows:

(Code omitted)

The most important parameter here is uType, which indicates what kind of message has been sent to this application. There are many types of DDE messages. In the callback function, we can return NULL if we do not want to process the message. Standard DDE messages will be explained in the following sections.

Server

Sample 15.1\DDE\Server is a standard SDI application generated by Application Wizard. The view of this application is based on class CEditView, so that it can be used to display current server states. The sample is a very basic DDE server, which implements DDE registration and name service registration but actually provides no service. In the sample, the service name is "Server". All the DDE functions are implemented in the frame window. In the sample, function CMainFrame::InitializeDDE() is called to initialize DDE. This function is called after the client view is created:

(Code omitted)

Function CMainFrame::InitializeDDE() simply calls ::DdeInitialize(...) with appropriate parameters, if the function returns DMLERR_NO_ERROR, it further proceeds to register the name service:

(Code omitted)

Here CMainFrame::m_dwInst is a DWORD type static variable and CMainFram::DdeCallback(...) is a static member function. This is because a callback function must be either a global function or a static member function. Function CMainFrame::Hszize() can be used to obtain a string handle for the name service and store the handle in static variable CMainFrame::m_hszServiceName. When the application exits, the name service is unregistered and also, the DDE is uninitialized there. This is implemented in WM_CLOSE message handler:

(Code omitted)

Also, the string handle is freed here. In the sample, functions CMainFrame::Hszize() and CMainFrame:: UnHszize() are implemented for obtaining and freeing string handles respectively:

(Code omitted)

(Code omitted)

Here CMainFrame::m_szService is a static variable.

Monitoring DDE Activities

To monitor the activities of the server, function CMainFrame::Printf(...) is implemented in the sample. This function imitates the well-known function prinf(...), and outputs a formatted string to the client window. We can use it exactly the same way as we use function printf(...). Since the view of the sample is based on class CEditView, it is easy for us to display text in the client window. Since the client window is solely used for displaying information, we need to set the window's ES_READONLY style before the it is created:

(Code omitted)

Because function CEditCtrl::ReplaceSel(...) is used to output text to the edit view in function CMainFrame::Printf(...), we further need to prevent the cursor position from being changed by the user (Function CEditCtrl::ReplaceSel(...) will always output text at the current caret position). For this reason, the following messages are handled to bypass the default implementations in the sample: WM_LBUTTONDOWN, WM_LBUTTONUP, WM_MOSUEMOVE and ON_WM_LBUTTONDBLCLK. The handlers of these messages are empty. This will prevent the application from responding to the mouse events so that the cursor position can not be changed under any condition.

If we execute the sample at this point, messages will appear on the client window indicating if the DDE initialization is successful.

15.2 Connecting to Server

Sample 15.2\DDE\Server is based on sample 15.1\DDE\Server. Also, a new sample 15.2\DDE\Client is created in this section, which is a dialog-based application.

At the beginning, the client also needs to do DDE initialization in order to support DDE. The client initialization procedure is almost the same with that of server except that it does not need to register the service name. The client needs to call functions ::DdeInitialize(...) and ::DdeUninitialize(...) for DDE initialization and clean up. Besides, it also needs to prepare a static (or global) DWORD variable for storing its instance identifier, and prepare a callback function that will be used to receive DDE messages.

DDE Connection: Client Side

Before requesting any service from the server, the client needs to make DDE connection. This is somehow similar to making a phone call: we need to dial number and make the connection first, then we can request the other side to do anything for us.

The client can call function ::DdeConnect(...) to set up a connection with the server:

```
HCONV ::DdeConnect(DWORD idInst, HSZ hszService, HSZ hszTopic, PCONVCONTEXT
pCC);
```

We need to provide three parameters in order to make the connection: the client instance ID, the server's service name, and the topic name that is supported by the server.

While the service name is the identification that can be used by the client to locate the server in the system, a topic name indicates the type of service which is provided by the server. A server can provide more than one service, whose properties and features can be defined by the programmer. For example, we can implement a DDE server managing images, and a client can request the server to send any image to it. For this service, we can name the topic name "image" (or whatever).

Like the service name, a string handle must be obtained for the topic name before it is used. When function ::DdeConnect(...) is called from the client side, this handle must be passed to its hszTopic parameter.

The server does not need to register the topic name. When the client makes the connection, the server will receive a XTYP_CONNECT message, and the topic name string handle will be passed as one of the parameters to the DDE call back function. Upon receiving this message, the handle passed with the message can be compared with the topic string handles stored on the server side (which represent all the topics supported by the server). If there is a match, it means the server supports this topic, otherwise the server should reject the connection.

DDE Connection: Server Side

On the server side, we need to handle XTYP_CONNECT message in the callback function. In the previous section, NULL is returned for all messages sent to the server. In order to respond to the connection request from the client, XTYP_CONNECT message must be processed.

As we mentioned before, the callback function has 8 parameters. Parameter uType indicates the type of message, if the message is a connection request, this message should be XTYP_CONNECT. The meanings of other 7 parameters are listed as follows:

(Table omitted)

In the sample of the previous section, a service name "Server" is registered on the server side. To let the connection be set up between the server and the client, we also need to prepare a topic name that will be used by both sides. In the sample 15.2\DDE\Server, string "Topic" is used as the topic name, whose string handle is obtained in function CMainFrame::Hszize() and freed in function CMainFrame::UnHszize().

The client should also obtain a string handle for the topic name and use it to make connection. The server will receive the handles of service name and topic name together with message XTYP_CONNECT. Upon receiving this message, the server needs to check if the service name and topic name requested by the client are supported by itself.

To compare two strings by their handles, we can call function

::DdeCmpStringHandles(...), which will compare two DDE strings. When calling this function, we need to pass the string handles to its two parameters. The function will return -1, 0 or 1 indicating if the first string is less than, equal to, or greater than the second string. The following is the format of this function:

```
int ::DdeCmpStringHandles(HSZ hsz1, HSZ hsz2);
```

Now it is clear what the server should do after receiving XTYP_CONNECT message: comparing hszAppName with its registered service name, and comparing hszTopic with its supported topic names by calling function ::DdeCmpStringHandles(...). If the comparisons are successful, the callback function should return TRUE, this indicates the connection request is accepted. Otherwise it should return FALSE, in which case the connection request is rejected. The following code fragment shows how this is implemented in sample 15.2\DDE\Server:

(Code omitted)

Client Implementation

Sample 15.2\DDE\Client is implemented as a dialog based application using Application Wizard. Similar to the server, here an edit control is included in the dialog template that will be used to display DDE activities. The DDE initialization procedure is implemented in function CDDEDialog::OnInitDialog(). There is also another edit box and a button labeled "Connect" in the dialog box. When the user clicks this button, the application will retrieve the string from this edit box, use it as the topic name and call function ::DdeConnect(...) to connect to the server:

(Code omitted)

The value returned by function ::DdeConnect(...) is a handle used for conversation. Every time the client want to make a transaction to the server, it must present this handle. By using the handle, the server knows with whom it is talking with.

In case the connection is not successful, function ::DdeConnect(...) will return a FALSE value.

Confirm Connection

On the server side, if the connection is successful, it will further receive an XTYP_CONNECT_CONFIRM message from the client. In this case, apart from hszTopic and hszAppName parameters, hconv is also used to provide the conversation handle that should be used by the server to make further conversation with the client. The following code segment shows how this message is processed on the server side:

(Code omitted)

Variable m_hConvServer is a static variable declared in class CMainFrame and is initialized to NULL in the constructor.

DDE Disconnection

After the connection is set up, the client and server can proceed to initiate various types of transactions. We will show this in later sections. After all the transactions are finished, or if one side wants to exit, the conversation must be terminated. Either the server or the client can terminate the conversion. This can be implemented by calling function ::DdeDisconnect(...) using the conversion handle. This function can be called either from the server or from the client side. Once the message is sent, the other side will be notified of the disconnection by receiving an XTYP_DISCONNECT message.

In the sample, once the connection is set up, it can be terminated from the client side by pressing "Disconnect" button or from the server side through executing DDE | Disconnect command. This is implemented as follows:

The server side:

(Code omitted)

The client side:

(Code omitted)

The following shows how message XTYP_DISCONNECT is handled in both sides:

The client side:

(Code omitted)

The server side:

(Code omitted)

Test

We can test this version of DDE client and server by starting both applications, then inputting "Topic" into the edit box contained in the client dialog box, and clicking "Connect" button (see Figure 15-1). After this, we will see that the server will output some information indicating if the connection is successful. Then, we can terminate the conversation either from the client or from the server side. We can also try to use an incorrect topic name while making the connection, this will cause the connection to be unsuccessful.

To prevent the application from exiting without terminating the conversation, both client and server need to check if the conversation is still undergoing while exiting. If necessary, function ::DdeDisconnect(...) will be called before the applications exit.

Figure 15-2 explains the procedure of DDE connection and disconnection.

(Figure 15-2 omitted)

15.3 Transaction: Data Request

Samples 15.3\DDE\Server and 15.3\DDE\Client are based on samples 15.2\DDE\Server and 15.2\DDE\Client respectively.

There are several types of transactions, one of which is XTYPE_REQUEST, it can be used by the client to request the server to send up-to-date value of an item to it.

We know that there are two names that are used to set up a connection: the service name and the topic name. The service name is used to locate the server and the topic name is used to specify a general topic. Under any topic there may exist several items, each item also needs an item name. When the client wants to request a transaction from the server, it needs to specify the item name.

For example, if a server provides a service that sends different type of data to the client(s), there may be different topics such as "Image", "Text" (indicating different type of data). Under each topic, there may exist different items such as "Image A", "Image B", "Text 1", "Text 2".

Data Request Transaction: Client Side

The client should call function ::DdeClientTransaction(...) to initiate a transaction:

(Code omitted)

We need to pass different types of data to parameters pData, cbData, wFmt and wType for different types of transactions. If the transaction requires the client to pass data to the server (Besides data request transaction, there exist other transactions that can be used by the client to send data to server), we need to use parameters pData and cbData to pass the data, and use parameter wFmt to specify the data format. If the client is requesting data from the server, it should specify the format of data by passing standard or user-defined format to parameter wFmt. In case the client is not sending data to the server, we can neglect parameters pData and cbData. Parameter wType can be used to specify the type of transaction the client is requesting from the server. In the case of data request transaction, this parameter must be set to XTYP_REQUEST. The meanings of the rest four parameters are the same for all types of transactions, which are listed in the following table:

(Table omitted)

For synchronous transmission mode (the asynchronous transmission mode will be introduced in a later section), after the client calls this function, a timer will be set (the time out value is specified by parameter dwTimeout). If there is no response from the server, the function does not return until the timer times out. We can specify an appropriate value to prevent the program from getting into the deadlock.

Data Request Transaction: Server Side

Calling this function from the client side will cause the server to receive an XTYP_REQUEST message. The server should check parameter uFmt, hConv, hszTopic and hszAppName, whose meanings are listed in the following table:

(Table omitted)

The server must check if it supports the topic and the specific item, as well as the data format. If it supports all of them, the server must prepare data using the required format and send the data back to client.

Preparing Data

One way to send data to the client is to prepare data in server's local buffers, then create a handle for this data, and send the handle to the client. The data handle can be obtained by calling function ::DdeCreateDataHandle(...), which has the following format:

(Code omitted)

The data can also be sent through a pointer. We will discuss this in a later section.

The following table explains the meanings of the above parameters:

(Code omitted)

We can use standard clipboard format such CF_TEXT, CF_DIB to pass data. If we define a special data format, we must register it before passing the data.

Receiving Data

Because data is not sent directly, the client must obtain the required data from the handle first. For synchronous transmission mode, the data handle will be returned directly from function ::DdeClientTransacton(). In case the transaction is not successful, a NULL value will be returned.

After receiving the handle, the client must first call function ::DdeAccessData(...) to access the data. After the data is processed, function ::DdeUnaccessData(...) must be called to "unaccess" the data. If the data is created with HDATA_APPOWNED flag, the client should not free the data. Otherwise, it can release the data by calling function ::DdeFreeDataHandle(...).

Samples

In the sample application, an item "Time" is supported by the server under "Topic" topic name. When the client request an XTYPE_REQUEST transaction on this item, the server get the current system time and send it to the client. The client then displays the time in one of its edit box.

Compared with the samples in the previous section, a new edit box and a button labeled "Request" are added to the dialog box (Figure 15-3). The edit box is read-only, which will be used to display the time obtained from the server. The button is used to let the user initiate the request transaction.

The following function shows how the client initiates the transaction after the user clicks "Request" button, and how the client updates the time displayed in the edit box if the transaction is successful:

(Code omitted)

The following code fragment shows how the server responds to message XTYPE_REQUEST, prepares data and sends it to the client:

(Code omitted)

15.4 Transaction: Advise

Basics

Another interesting transaction is Advise, which provides a way to let the server inform the client after an item stored on the server side has changed. The advise transaction can be initiated from the client side by initiating XTYP_ADVSTART type transaction. As the server receives this message, it keeps an eye on the item that is required by the client for advise service. If the data changes, the server will receive an XTYP_ADVREQ message indicating that the item has changed (This message is posted by the server itself, which could be triggered by any event indicating that the topic item has changed. For example, for "time" item discussed in the previous section, it could be triggered by message WM_TIMER). After receiving message XTYP_ADVREQ, the server sends an XTYP_ADVDATA message along with the handle of the updated data to the client. After the client receives this message, it updates the advised topic item.

Now that we understand how XTYP_REQUEST type transaction is handled, it is easier for us to figure out how the advise transaction should be implemented. First, the client initiates advise transaction by calling function ::DdeClientTransaction(...) and passing XTYP_ADVSTART to parameter wType. Upon receiving this message, the server must return TURE if it supports the specified topic and item; otherwise, it should return FALSE. Once the data has changed, the server should call function ::DdePostAdvise(...) to let its callback function receive an XTYP_ADVREQ message. Upon receiving this message, the server needs to prepare the data and send the data handle to the client (There is no special function for doing this, all the server needs to do is returning the data handle from the callback function). Once the server has provided advise, the client will receive an XTYP_ADVDATA message along with the data handle. After receiving this message, the client should update the advised item. The client can terminate the advise service at any time by sending XTYP_ADVSTOP message through calling function ::DdeClientTransaction(...).

Initiating Advise Transaction

In the samples contained in 15.4\DDE\, a new topic item "Text" is added to both client and server, which will be used as an example for advise transaction. Like "Time" item, a new edit box and a button labeled "Advise" are also added to the dialog box (Figure 15-4). On the server side, a variable CMainFrame::m_szText is declared and the user can edit this string through dialog box IDD_DIALOG_ADVISE. The following code fragment shows how the client initiates XTYPE_ADVSTART transaction after the user clicks "Advise" button:

(Code omitted)

It is more or less the same with XTYP_REQUEST transaction. If the transaction is successful, we set a Boolean type variable CDDECliDlg::m_bAdvise to TRUE and change the button's text to "Unadvise". By doing this way, the button can be used for both advise starting and stopping.

Advise Transaction Responding

On the server side, after message XTYP_ADSTART is received, it checks the topic name, item name, and the data format specified by the client (just like data request transaction). If the server supports all of them, it sets a Boolean type variable CMainFrame::m_bAdvise to TRUE. This is a flag, if it is TRUE, whenever the user modifies the content of CMainFrame::m_szText variable (it is the advised item stored on the server side, which can be modified by a new command added to the application, see below for detail), function ::DdePostAdvise(...) will be called. The following code fragment shows how message XTYP_ADSTART is processed on the server side:

(Code omitted)

On the server side, a new command DDE | Advise is added to mainframe menu IDR_MAINFRAME. The following function shows how this command is implemented. If the user changes the content contained in variable CMainFrame::m_szText (if flag CMainFrame::m_bAdvise is TRUE at this time), message XTYP_ADVREQ will be posted to the server:

(Code omitted)

Upon receiving message XTYP_ADVREQ in the DDE callback function, we must prepare a string handle for CMainFrame::m_szText, and return it when the function exits:

(Code omitted)

Again, the data handle is created by calling function ::DdeCreateDataHandle(...). If the server doesn't support this service, it should return FALSE.

Upon Receiving Advise

This will again cause the client to receive XTYP_ADVDATA message. The data handle will be passed through parameter hData, which should be accessed through calling function ::DdeAccessData(...). After the data is obtained, the client need to call function ::DdeUnaccessData(...) to stop accessing the data, and call function ::DdeFreeDataHandle(...) to free the data. If these procedures are successful, the client's DDE callback function should return a DDE_FACK value, otherwise it should return a DDE_FNOTPROCESSED value. The following code fragment shows how message XTYP_ADVDATA is processed on the client side:

(Code omitted)

Terminating Advise Transaction

The advise can be terminated from the client side by sending message XTYP_ADVSTOP to the server:

(Code omitted)

On the server side, after receiving this message, it simply turns off CMainFrame::m_bAdvise flag. After this, if the user updates variable CMainFrame::m_szText, function ::DdePostAdvise(...) will not be called.

15.5 Transactions: Poke and Execute

Sample 15.5\DDE\Server and 15.5\DDE\Client are based on sample 15.4\DDE\Server and 15.4\DDE\Client respectively.

The poke transaction is the opposite of request transaction: the client can use this transaction to send data to the server. The transaction is relatively simple: the client initiates the transaction by sending a message along with the data handle to the server. After the server receives the message, it can obtain the data from the data handle.

Like require and advise transactions, poke transaction also needs a topic name and an item name. In the samples contained in 15.5\DDE\, a new item "Poke" is supported by both the server and the client. Also, a new edit box and a new button labeled "Poke" are added to the client dialog box (see Figure 15-5). The user can input any string into the edit box, and use poke transaction to send it to the server.

Poke Transaction: Client Side

The procedure of preparing data and calling function ::DdeClientTransaction(...) is very similar to that of request transaction. The difference between two types of transactions is that here it is the client that needs to prepare the data handle. In sample 15.5\DDE\Client, the user can input any string into the edit box and send it to the server.

By now when sending data between two DDE applications, we always use data handle. An alternate way of sending data is to use a pointer that contains the address to the data buffers. (The samples contained in 15.5\DDE use data handle. We will see an example using pointer to transfer data in a later section). When the client calls function ::DdeClientTransaction(...), the first parameter can be set to either a string handle or a string pointer. If it is string handle, the second parameter must be 0xFFFFFFFF. If the first parameter is a string pointer, the second parameter must specify the length of the buffers.

The following code fragment shows how the client initiates a poke transaction in sample 15.5\DDE\Client:

(Code omitted)

Poke Transaction: Server Side

The server will receive an XTYP_POKE message. In the samples contained in 15.5\DDE, the server obtains the data from the string handle and calls function CMainFrame::Printf(...) to display the string in the client window, then return a DDE_FACK value. If the server does not support either topic name, item name or the format, it should return a DDE_FNOTPROCESSED value.

Transaction: Executing Commands

Another type of transaction is execute. By using this type of transaction, the client can send commands to the server and let them be executed at the server side. This transaction is different from all other type of transactions because it does not need an item name and does not require format specification. We need to pass NULL to parameter hszItem (string handle of the item name) and parameter wFmt when calling function ::DdeClientTransaction(...) to initiate execute transaction. However, a text string must be used to specify the command (The command must be specified by parameters pData and cbData). In the samples, the execute transaction is implemented similar to that of poke transaction: an edit box and a button labeled "Execute" are added to the dialog box, the user can input any command into the edit box and click "Execute" button to execute it. The following code fragment shows how this is implemented on the client side:

(Code omitted)

Rather than using data handle, the address of the buffers will be used to transfer data. So when the client calls ::DdeClientTransaction(...) to initiate transaction, the first parameter of this function is set to the address rather that the handle of the buffers. Also, the second parameter is set to the size of buffers instead of 0xFFFFFFFF. Since there is no need to specify item name and data format, the fourth and fifth parameters are set to NULL.

The address of the buffers will not be sent directly to the server. Instead, they will be used to create a DDE object that contains the data. So as the server receives the corresponding message, it actually gets the handle of this DDE object instead of the buffer address. In order to access the data, it must prepare some buffers allocated locally and copy the data from the DDE object into these buffers by calling function ::DdeGetData(...), whose format is as follows:

DWORD ::DdeGetData(HDDEDATA hData, LPBYTE pDst, DWORD cbMax, DWORD cbOff);

Parameter hData is the handle received from the DDE callback function that identifies the DDE object. Parameter pDst is a pointer to the buffers that will be used to receive data, whose size is specified by parameter cbMax. Parameter cbOff specifies the offset within the DDE object.

Generally we do not know the size of data beforehand, so before allocating buffers, we can call this function and pass NULL to its pDst parameter, which will cause the function to return the actual size of the data. Then we can prepare enough buffers, pass the address of buffers and the buffer length to this function to actually receive data.

In the samples, when the client initiates an execute transaction, the server does nothing but displaying the command in its client window. Although it seems like poke transaction, there are some radical differences between two types of transactions:

1) The poke transaction can be used to transmit any type of data, it requires a format specification. The execute transaction only transmit a simple command.

2) The poke transaction must specify an item name. The execute transaction does not require this.

3) As we will see in section 15.7, the server will respond to execute transaction by executing a command. The poke transaction just update data.

The following code fragment shows how the execute transaction is implemented on the server side:

(Code omitted)

15.6 Asynchronous Transaction

Synchronous vs. Asynchronous

By now all our transactions are implemented by synchronous transmission mode. An alternative way of implementing them is using asynchronous transmission mode. The difference between two types of transactions is listed as follows:

Synchronous transaction: after the client initialized the transaction, it will start a timer and wait till it gets the response from the server. If there is no response when timer times out, the transaction will be terminated.

Asynchronous transaction: after the client initialized the transaction, it does not wait. Instead, the client will go on to do other job. After the server finished the transaction, the client will receive a message indicating that the previous transaction has been finished.

Synchronous transaction is simple to implement. However, if the server responds very slowly, it will cause severe overhead. In our sample, synchronous transaction is good enough because the server supports only very limited types of services and each transaction won't take much time. But generally a server may need to serve multiple clients simultaneously, so when a client initialized a transaction, the server may be busy with another transaction. If we use synchronous transaction, the client may need to wait until the server comes to serve it. If we have several clients waiting concurrently, it will waste the system resource.

The asynchronous transaction is implemented more efficiently. As the client initialized the transaction, it just moves on to do other job; when the server finishes this transaction, the client will receive a message telling it the result of the transaction. In this case the client's waiting time is eliminated.

Implementing Asynchronous Transaction

In DDEML, asynchronous transaction can be initiated very easily. The only difference between initializing a synchronous transaction and an asynchronous transaction is that instead of specifying a time out value, we need to pass TIMEOUT_ASYNC to parameter dwTimeout when calling function ::DdeClientTransaction(...). Another difference is that instead of receiving a value (usually the result of the transaction) from function ::DdeClientTransaction(...) directly, the client will receive an XTYP_XACT_COMPLETE message and the result of the transaction will be passed through parameter hData of the callback function. Figure 15-6 illustrates the difference between two type of transactions.

Samples

Samples 15.6\DDE\Server and 15.6\DDE\Client are based on Samples 15.5\DDE\Server and 15.5\DDE\Client respectively. In the two samples, the poke transaction is implemented in asynchronous mode. The following code fragment shows how the transaction is initiated:

(Code omitted)

The following code fragment shows how the transaction result is processed when the client receives an XTYP_XACT_COMPLETE message:

(Code omitted)

There is no change on the server side.

All types of transactions can be implemented by either synchronous or asynchronous mode.

15.7 Program Manager: A DDE Server

By now, we have introduced some of the most useful DDE transactions. We can use these data transfer protocols to design and build applications that can share data and information. Besides this, we can also write client application to communicate with standard DDE servers contained in the system.

Program Manager

Under Windows, all types of applications are managed into groups. By clicking on Start | Programs command on the task bar, we will see many groups, such as "Accessories", "Startup". Within each group, there may exist some items that are linked directly to executable files, or there may exist some sub-groups. Sometimes we may want to modify this structure by adding a new item or deleting an existing item.

Actually this structure is managed by Program Manager, which is a DDE server. We can interact with this server to create group, delete group, add items to the group, delete items from a group through initiating Execute transactions.

The client sample from the previous section is modified so that it can be used to communicate only to Program Manager. Here, the service name of the Program Manager is "Progman". As we can see, the DDE initialization for this client is exactly the same as we did before.

The five commands we will ask the server to execute are: creating group; bringing up an existing group; deleting a group; creating an item; deleting an item. All the DDE commands must start and end with square braces ('[' and ']'). The following table lists the formats of five commands:

(Table omitted)

A combo box for storing the command types is added to sample's dialog box. Besides this, the dialog box also contains an edit box and a button labeled "Command". The edit box will be used to let the user input parameters for the command. For example, if we want to create a group with name "Test", we can select "Create group" from the combo box and input "Test" into the edit box then click "Command" button. The application will initiate an execute transaction to the Program Manager and send "[CreateGroup(Test)]" command to it.

We can use this program to create groups and add items to a group. We can also delete unwanted groups or remove items from a group.

Summary

1) To support DDE in an application, function ::DdeInitialize(...) must be called to initialize it; also, before the application exits, function ::DdeUninitialize(...) must be called to uninitialize the DDE.

2) The server must register DDE service by calling function ::DdeNameService(...). Before the server exits, it must call the same function to unregister the service.

3) Before DDE client initiates any transaction, it must call function ::DdeConnect(...) to obtain a conversation handle that can be used to identify the server in the following transactions.

4) A server can support several topics, under each topic there may be several items supported. When initiating a transaction, the client need to specify both of them.

5) Data can not be exchanged directly between two processes. Instead, it must be sent either by data handle or by DDE object. In the former case, the data can be accessed by calling functions ::DdeAccessData(...). Function ::DdeUnaccessData(...) can be used to unaccess data and ::DdeFreeDataHandle(...) can be used to free data. In the later case, the data can be retrieved by calling function ::DdeGetData(...). To create a data handle, we can use function ::DdeCreateDataHandle(...).

6) In DDE model, two strings can be compared by their handles. In order to do this, we need to call function ::DdeCmpStringHandles(...).

7) Data request transaction can be used to request data from the server.

8) Advise transaction can be used to let the client monitor the changes on the data stored on the server side.

9) Poke transaction can be used by the client to update the data stored on the server side.

10) Execute transaction can be used by the client to let commands be executed on the server side.

11) There are two types of transmission modes: synchronous and asynchronous. Synchronous transaction is easy to implement, but asynchronous transaction is more efficient under certain conditions.

12) Program Manager is a DDE server that supports execute transaction.

BACK TO INDEX

Chapter 16 Context Sensitive Help

Help is a very important feature for all types of applications. Since the user interface cannot be made intuitive enough to eliminate guessing when the user interacts with a program, we need to include context sensitive help to tell the user what exactly each command means. Although the help development is usually not a part of job for programmers, the persons who are in charge of application development should cooperate closely with the help developers to create high-quality applications.

16.1 Context Sensitive Help for Menu Commands

Context Sensitive Help

Context sensitive help is supported by the up-to-date versions of MFC. It is more user friendly than the old style on-line help, and provides an easier way to let the user find out pertinent help topics. Generally, context sensitive help can be enabled when an SDI or MDI application is being generated by the Application Wizard (Figure 16-1).

After we've enabled the context sensitive help, a "question mark" button will appear on the mainframe tool bar (Figure 16-2). If we click on it, the mouse cursor will change to a question mark. If we use this cursor to click any menu command (or any part of the application window), the help window will pop up displaying the description of the item that was just clicked.

Context Sensitive Help for Menu Commands

By default, only the standard items such as caption bar, status bar and standard commands (File | Open, File | Save, File | print) will support context sensitive help. Menu commands (or mainframe tool bar commands) added by the programmer will not support context sensitive help automatically.

By default, contents of help are stored in a rich text format file generated by the Application Wizard: "AfxCore.rtf". We can find many footnotes within this file, each footnote corresponds to one help page. To implement the context sensitive help for a custom menu command, we need to add a new footnote to file "AfxCore.rtf", and link it to the command.

If the context sensitive help is enabled within the Application Wizard in step 4 (see Figure 1-1), the help project will be generated automatically. All the files used to build the help will be contained in a "hlp" directory under the project directory. For example, if we use the Application Wizard to generate an SDI application named "Help", the help project will be generated under "Help\hlp\" directory. There will be a batch file "Makehelp.bat" under the "Help" directory. Also, there are some other important files under "Help\hlp" directory that are used to build the help. The following table lists the usages of these files:

(Table omitted)

The help is compiled by a utility named "Microsoft Help Workshop". The executable file "Hcw.exe" can be found under Visual C directory "~DevStudio\Vc\Bin\". This utility can compile "*.hpj" file to generate a target help file.

By double clicking on the "*.hpj" file or "*.cnt" file (In "Explorer", they are described as "Help project file" and "Help contents file" respectively), we can compile the help project in the help workshop environment. Also, when we compile the application in Developer Studio, the help project will also be compiled. The help project file (".hpj) is similar to a make file when we execute C compilers, it contains information about how to generate the target help file. The help contents file (".cnt") contains the information of "help topics". If we execute Help | Help Topics command from the application, the help contents will be displayed in a "Help Topics" property sheet (Figure 16-3). All the descriptions about the commands and the application are included in "AfxCore.rtf" and "AfxPrint.rtf" files, we must edit them in order to add custom help descriptions.

Sample

Because the default help project already has many help items, without the knowledge of the help project, it is difficult for a programmer to add items for the newly added commands. Sample 16.1\Help is a standard SDI application generated by Application Wizard, which demonstrates how to add new help items and link them to the application commands to support context sensitive help.

New Commands

First, after the standard project is generated, four new commands are added to the application. These commands are implemented in both the mainframe menu and the tool bar, and their IDs are ID_HELPTEST_TESTA, ID_HELPTEST_TESTB, ID_HELPTEST_TESTC and ID_HELPTEST_TESTD respectively. In IDR_MAINFRAME menu, the new commands are Help Test | Test A, Help Test | Test B, Help Test | Test C, and Help Test | Test D. In the IDR_MAINFRAME tool bar, we also have four buttons corresponding to the four command IDs. The message handlers of these commands are all blank, because we just want to demonstrate how to implement context sensitive help for them.

Editing "AfxCore.rtf"

We must add four items to the help file in order to link a command to the help. In order to do this, we must modify file "AfxCore.rtf". The rich text format file can be edited by a word processor like Microsoft(Word (WordPad is not powerful enough for this purpose, because it does not support footnote editing). Within this file, each help item is managed as a footnote. If we want to add a new help item, we just need to add a new footnote.

To show how to add a footnote to the ".rtf" file, lets assume that we want to add a footnote for ID_HELPTEST_TESTA command.

Each footnote must be associated with a tag, so that it can be referenced from inside or outside the file. In Microsoft(Word, we can either add number-based tags or user-defined tags. To make the tags meaningful, usually we define tags by ourselves. The tags can be any string, usually they will have some relationship with the command IDs implemented in the application. For example, we can use "help_test_A" as the tag for the footnote that will be used to implement help for the command whose ID is ID_HELPTEST_TESTA.

To add a new tag within Microsoft(Word, first we need to move the cursor to the bottom of the file, then execute Insert | Break... command. From the popped up dialog box, we need to check the radio button labeled "Page Break". If we click "OK" button, a new page will be added to the file (Generally each footnote needs to use one page, so we need to add page break whenever a new footnote is added). Now execute Insert | Footnote command, and check the radio button labeled "Footnote" from the popped up dialog box (in "Insert" section). Then, check radio button labeled "Custom Mark" (in "Numbering" section) and input a '#' into the edit box beside it (Figure 16-4). Finally, click "OK" button. Now the client window of Word will split into two panes, the lower of which will show all footnote tags. The caret will be placed right after the '#' sign waiting for us to type in the footnote tag. We can type "help_test_A", then click on the upper pane. Then we can input any help description for command ID_HELPTEST_TESTA.

Tag "help_test_A" can be referenced either from other footnote pages or from the application. If we want to let the user jump from one help item to another (for example, when viewing help on "telp_test_A", the user may want to jump to footnote "help_test_B" by clicking on a link within the same page), we need to implement links by editing the ".rtf" file.

Suppose we have added four footnotes for the newly implemented commands: "help_test_A", "help_test_B", "help_test_C", and "help_test_D", and we want to add links to the rest of three commands within each footnote page. For example, help item "help_test_A" may be implemented as illustrated in Figure 16-5:

Under "See Also" statement, there are three links that will direct mouse clicking to footnotes "help_test_B", "help_test_C" and "help_test_D".

To create this type of links, we need to use special font format. We need to use double underline style to format the text that will be linked to a footnote (By doing this, the text formatted with double underline can respond to mouse clicking and bringing up another help item). Following the underlined text, we need to place the footnote tag using "Hidden" font style. To let the hidden text be displayed in the Word editor, we can execute command Tools | Options... and click tab View on the popped up property sheet. Then within "Nonprinting Characters" section, check "Hidden Text" check box.

Now we can make a link very easily. First we need to type in and format the text as illustrated in Figure 16-6:

To format text using double underline, we can select the text, then execute command Format | Font.... From the popped up property sheet, we can go to "Font" page and select "Double" form "Underline" combo box. By doing this, the selected text will be double underlined. To format text using "Hidden Text" style, we can first select the text, then execute command Format | Font..., go to "Font" page, make sure that "None" is selected from the "Underline" combo box, and check "Hidden" check box within "Effects" section.

ID Mapping

In order to support context sensitive help, we must link the footnotes to their corresponding command IDs. In our case, we need to link "help_test_A" to ID_HELPTEST_TESTA, "help_test_B" to ID_HELPTEST_TESTB... and so on.

This ID mappings are implemented in ".hm" file, so by opening file "Help.hm" with a text editor, we will see all the IDs of the help items supported in the sample. Generally help IDs are generated according to certain rules: it bases the help ID of a control (or a window) on its resource ID. By default, for a command whose ID starts with "ID_", MFC generates symbolic help ID by prefixing a character 'H' to the resource ID. For example, for command ID_HELPTEST_TESTA, the help ID generated by MFC is HID_HELPTEST_TESTA. For the actual ID value, MFC generates it by adding a fixed number to the corresponding resource ID value. For example, if the value of ID_HELPTEST_TESTA is 0x8004, the value of HID_HELPTEST_TESTA would be 0x18004 ¾ here a number of 0x1000 is added to the command resource ID.

By doing this, when the user executes a command in the help mode (When the mouse cursor changes to a question mark after the user clicks command ID_CONTEXT_HELP), the application will first find out the resource ID of the command being executed, add a fixed value, then pass the result to the help. By using this rule to generate ID for a help item, it is easier for us to implement context-sensitive help.

Although the ID mapping is customizable, which means we can prefix different character(s) to a resource ID for generating help ID (For example, the help ID of command ID_HELPTEST_TESTA could be HLID_HELPTEST_TESTA), and we can add any value to the resource ID to generate a help ID, it is more convenient if we stick to the rules of MFC. For example, if we add 0x20000 instead of 0x10000 to the resource ID to make a help ID, we also need to add some code to the program to customize its default behavior.

Another problem still remains: since we've already added footnotes and defined our own tags, the command IDs are still not directly linked to the footnote tags. For example, the help ID of command ID_HELPTEST_TESTA is HID_HELPTEST_TESTA, while the footnote tag implemented in the help file is "help_test_A". This can be solved by defining alias names in the help project file. By opening "*.HPJ" file, we will see an [ALIAS] session. Under this session, a help ID can be linked directly to a footnote tag. In our case, the footnote tags are "help_test_A", "help_test_B"..., and the help IDs automatically generated by MFC are HID_HELPTEST_TESTA, HID_HELPTEST_TESTB.... To link them together, we can add the following to alias session:

```
[ALIAS]

......

HID_HELPTEST_TESTA=help_test_A

HID_HELPTEST_TESTB=help_test_B

HID_HELPTEST_TESTC=help_test_C

HID_HELPTEST_TESTD=help_test_D
```

Obviousely, if we use the default help ID strings to implement footnote tags (i.e., use HID_HELPTEST_TESTA instead of help_test_A as the footnote tag), the ID mapping could be eleminated.

By doing this, when the user executes certain command in the help mode, the corresponding help page will be automatically brought up.

Help Topics Dialog Box

The footnotes can also be referenced from "Help Topics" dialog box. This dialog box is activated when the user executes command ID_HELP_FINDER. The contents contained in "Help Topics" dialog box are stored in ".cnt" file. To edit this file, we can open it using Help Workshop. In the Help Workshop, we can add two types of items: Heading and Tab Entry. A heading will not be linked to any footnote. Under each heading, we can add several tab entries, which need to be linked to footnotes. A new item can be added by clicking buttons labeled "Add Above...", "Add Below...". After that, a dialog box will pop up asking us to input the description text as well as the help ID (the footnote tag). The description of an item should be input into the edit box labeled "Title", and the footnote tag should be input into the edit box labeled "Topic ID" (Figure 16-7).

After making any change to the help project, we need to recompile the project in order to get the up-to-date help.

To make the help working, both ".hlp" and ".cnt" file must be copied to the directory that contains the application executable file.

16.2 Context Sensitive Help for Common Controls

Sample 16.2-1\Help and 16.2-2\Help are based on sample 16.1\Help, they demonstrate how to support context sensitive help in a dialog box.

Supporting Context Sensitive Help in Dialog Box

The previous section describes how to add context sensitive help for commands. We can also add this fancy feature for common controls contained in a dialog box. To implement context sensitive help for a dialog box, we can check "Context Help" check box under "More Styles" page of the "Dialog Properties" property sheet (Figure 16-8). After this type of dialog boxes are invoked, a question mark button will appear on their caption bars (Figure 16-9). This button has the same functionality with command ID_CONTEXT_HELP described in the previous section. By clicking on this button, we will enter help mode, with the mouse cursor becoming a question mark. As we click any controls contained in the dialog box with this cursor, the help will be activated and the corresponding help window will be brought up.

ID Naming Rules

Adding context sensitive help for common controls is more complicated than that of menu commands. First we must add footnotes for each common control contained in the dialog box, then we need to do the ID mappings.

By default, MFC will generate help IDs for the commands or controls whose resource IDs start from "ID_", "IDM_", "IDP_", "IDR_", "IDD_" and "IDW_" by prefixing a single character 'H' to them. Also, the values of these help IDs are generated by adding a fixed number to the corresponding resource IDs. This fixed number is different for different types of IDs. For example, for "ID_XXX" and "IDM_XXX" types of IDs, 0x10000 will be added to the resource ID; for "IDP_XXX" type IDs, 0x30000 will be added; for "IDR_XXX" and "IDD_XXX" types of IDs, 0x20000 will be added; for "IDW_XXX" type IDs, 0x50000 will be added.

By default, the IDs of the common controls in a dialog box all start with "IDC_" prefix, which is not included in the default mapping. This means we must generate help IDs by ourselves. Actually, this can be easily achieved. If we open file "Makehelp.bat" (in our case, this file should be located in "16.2\Help\" directory), we will see that the help IDs are generated through "Makehm" utility, which can be executed under DOS prompt.

"Makehm" has the following syntax:

```
Makehm argument 1, argument 2, argument 3, argument 4 >> "File name"
```

where

argument 1: prefix of resource ID

argument 2: prefix of help ID

argument 3: base number

argument 4: resource header file name

For example, if we want to generate symbolic help IDs for those IDs prefixed with "IDC_", and add 0x10000 to the resource IDs to generate actual help IDs, we can execute this command as follows (Here, the application resource header file name is "resource.h", and the ".hm" file name is "Help.hm", it is located under the directory of "Hlp"):

```
Makehm IDC_, HIDC_, 0x10000, resource.h >> "hlp\Help.hm"
```

If we include the above statement in file "Makehelp.bat", after executing it, we will see that all the common controls will have corresponding help IDs in file "Help.hm".

Enabling Context Sensitive Help for Common Controls

In sample 16.2-1\Help, a dialog box IDD_DIALOG is added to the project. The dialog box supports context sensitive help implementation. There are four controls included in the dialog box: edit box IDC_EDIT, radio button IDC_RADIO, combo box IDC_COMBO and a push button IDC_BUTTON. After adding the "Makehm" command to "Makehelp.bat" file and executing it, we will see the following help IDs in file "Help.hm":

```
// Common Controls (IDC_*)
```

HIDC_EDIT 0x103E8

HIDC_BUTTON 0x103E9

HIDC_RADIO 0x103EA

HIDC_COMBO 0x103EB

Please note that file "Makehm.exe" is located in directory "...DevStudio\Vc\Bin\". We must set path to this directory in order to execute it from DOS prompt. However, if we compile the help through Developer Studio or Help Workshop, there is no need to set the path. An alternate solution that can let us run the batch file from DOS prompt without setting path is to copy file "Makehm.exe" to the directory that contains file "Makehelp.bat".

In the sample application 16.2-1\Help, four new footnotes with tags "common_button", "common_combobox", "common_edit" and "common_radio" are added to file "AfxCore.rtf". Their alias names are specified in the help project file:

[ALIAS]

......

HIDC_BUTTON=common_button

HIDC_COMBO=common_combobox

HIDC_EDIT=common_edit

HIDC_RADIO=common_radio

This still does not finish context sensitive help implementation for the common controls. When the user uses question mark cursor to click a common control contained in the dialog box, by default, it is the ID of the dialog box, not the ID of the control that will be used to activate the help. To enable context help for each individual control, we need to call function CWnd::SetWindowContextHelpId(...) for it, which has the following format:

BOOL CWnd::SetWindowContextHelpId(DWORD dwContextHelpId);

Parameter dwContextHelpId is the help ID of the control. This function can be used to link a control's resource ID to its help ID.

To make things work, we must use the IDs generated by "Makehm" utility for each control. In the sample, a function CHelpDlg::SetContextHelpId() (CHelpDlg is the class used to implement the dialog box) is implemented to set help IDs for all the common controls contained in the dialog box:

(Code omitted)

First we call function CWnd::GetWindow(...) and use flag GW_CHILD to find out the first child window contained in the dialog box. Then we call this function repeatedly using GW_HWNDNEXT flag to find out all the other child windows. The loop will be stopped after the last child window is enumerated. For each child window (common control), we obtain its resource ID value by calling function CWnd::GetDlgCtrlID(), adding 0x10000 to it, and using this value to set help ID.

We must make sure that both utility "Makehm" and function CWnd::SetWindowContextHelpID(...) add the same value to the resource IDs to result in help IDs.

Function CWnd::OnHelpInfo(...)

We also need to override function CWnd::OnHelpInfo(...) in order to let the help jump to a special page when being activated. The following code fragment shows how this function is overridden in the sample:

(Code omitted)

When the user uses question mark cursor to click a control, by default, function CWnd::OnHelpInfo(...) will be called, this will not activate help for the common control being clicked. To customize this, we must check if the help ID corresponds to one of the common controls contained in the dialog box. If so, we need to activate the help by ourselves (and jump to the corresponding help page).

The information of the control (or window) will be passed through a HELPINFO type object. Especially, the help ID will be stored in dwContextId member of this structure. If we have called function CWnd:: SetWindowContextHelpId(...) for a control, the value of this ID will be the one we set there. So in our case, the resource ID can be obtained by subtracting 0x10000 from the help ID.

In the above function, we first check if the control is one of the four controls that support context sensitive help. If so, function CWinApp::WinHelp(...) is called to activate the help (The help ID is passed to the first parameter of this function). Otherwise, default implementation of this function will be called.

Function CWinApp::WinHelp(...) has two parameters:

```
virtual void CWinApp::WinHelp(DWORD dwData, UINT nCmd=HELP_CONTEXT);
```

The help will be activated in different styles according to the values of parameter nCmd. If we do not specify this parameter, the help will be implemented in default style, and jump to the footnote corresponding to the help ID specified by parameter dwData. If the help ID could not be found, the first footnote contained in the help will be displayed and an error message will pop up.

Displaying Help in a Pop up Window

In sample 16.2-2\Help, context help for the controls are displayed in different styles:

(Code omitted)

The edit box and radio button still use the default style, whose help window contains standard menu and buttons. The help windows of push button and combo box controls are implemented by pop up window, which is a smaller window with a yellowish background, and does not contain other controls (Figure 16-10). By passing different parameters to function CWinApp::WinHelp(...), we can activate "Help Topic" dialog box. Also, we can adjust its position and size of the help window before it is displayed,.

Summary

1) In order to implement context sensitive help for a command, we need to add a footnote to the "AfxCore.rtf" file, then link the command ID to the tag of the footnote.

2) Usually the help ID is generated by prefixing character(s) to the resource ID of the command, and the value of the help ID is generated by adding a fixed number to the value of the resource ID.

3) The resource ID can be linked to a footnote tag by specifying an alias name in ".hpj" file.

4) A help page can be referenced either from other help pages or from the application.

5) To support context sensitive help in the dialog box, we need to call function CWnd:: SetWindowContextHelpId(...) for every common control that will support context sensitive help, and override function CWnd::OnHelpInfo(...) to activate customized help window.

6) By default, function CWnd::OnHelpInfo(...) does not support context sensitive help for common controls. In this case, we can call CWinApp::WinHelp(...) to customize the default help implementation.

7) When calling function CWinApp::WinHelp(...), we can use various parameters to activate help windows in different styles. For example, using parameter HELP_CONTEXTPOPUP will invoke a pop up help window.

BACK TO INDEX