Module 18: Using ActiveX Controls

Program examples compiled using Visual C++ 6.0 (MFC 6.0) compiler on Windows XP Pro machine with Service Pack 2. Topics and sub topics for this Tutorial are listed below:

Using ActiveX Controls

Microsoft Visual Basic (VB) was introduced in 1991 and has proven to be a wildly popular and successful application development system for Microsoft Windows. Part of its success is attributable to its open-ended nature. The 16-bit versions of VB (versions 1 through 3) supported Visual Basic controls (VBXs), ready-to-run software components that VB developers could buy or write themselves. VBXs became the center of a whole industry, and pretty soon there were hundreds of them. At Microsoft, the Microsoft Foundation Class (MFC) team figured out a way for Microsoft Visual C++ programmers to use VBXs in their programs, too.

The VBX standard, which was highly dependent on the 16-bit segment architecture, did not make it to the 32-bit world. Now **ActiveX Controls** (formerly known as **Object Linking and Embedding** (OLE) controls or OCXs) are the industrial-strength replacement for VBXs based on Microsoft **Component Object Model** (COM) technology. ActiveX controls can be used by application developers in both VB and Visual C++ 6.0. While VBXs were written mostly in plain C, ActiveX controls can be written in C++ with the help of the MFC library or with the help of the **ActiveX Template Library** (ATL).

This module is not about writing ActiveX controls; it's about using them in a Visual C++ application. The premise here is that you can learn to use ActiveX controls without knowing much about the COM on which they're based. After all, Microsoft doesn't require that VB programmers be COM experts. To effectively write ActiveX controls, however, you need to know a bit more, starting with the fundamentals of COM. Consider picking up a copy of Adam Denning's ActiveX Controls Inside Out (Microsoft Press, 1997) if you're serious about creating ActiveX controls. Of course, knowing more ActiveX Control theory won't hurt when you're using the controls in your programs.

ActiveX Controls vs. Ordinary Windows Controls

An ActiveX control is a **software module** that plugs into your C++ program the same way a **Windows control** does. At least that's the way it seems at first. It's worthwhile here to analyze the similarities and differences between ActiveX controls and the controls you already know.

Ordinary Controls: A Frame of Reference

In Module 5, you used **ordinary Windows controls** such as the edit control and the list box, and you saw the **Windows common controls** that work in much the same way. These controls are all child windows that you use most often in dialogs, and they are represented by MFC classes such as `CEdit` and `CTreeCtrl`. The client program is always responsible for the creation of the control's child window.
Ordinary controls send notification command messages (standard Windows messages), such as `BN_CLICKED`, to the dialog. If you want to perform an action on the control, you call a C++ control class member function, which sends a Windows message to the control. The controls are all windows in their own right. All the MFC control classes are derived from `CWnd`, so if you want to get the text from an edit control, you call `CWnd::GetWindowText`. But even that function works by sending a message to the control.
Windows controls are an integral part of Windows, even though the Windows common controls are in a separate DLL.
Another species of ordinary control, the so-called **custom control**, is a programmer-created control that acts as an ordinary control in that it sends `WM_COMMAND` notifications to its parent window and receives user-defined messages. You'll see one of these in Module 16. So many controls huh!

How ActiveX Controls Are Similar to Ordinary Controls

You can consider an ActiveX control to be a child window, just as an ordinary control is. If you want to include an ActiveX control in a dialog, you use the dialog editor to place it there, and the identifier for the control turns up in the resource template. If you're creating an ActiveX control on the fly, you call a `Create()` member function for a class that represents the control, usually in the `WM_CREATE` handler for the parent window. When you want to manipulate an ActiveX control, you call a C++ member function, just as you do for a Windows control. The window that contains a control is called a **container**.

How ActiveX Controls Are Different from Ordinary Controls: Properties and Methods

The most prominent ActiveX Controls features are **properties** and **methods**. Those C++ member functions that you call to manipulate a control instance all revolve around properties and methods. Properties have symbolic names that are matched to integer indexes. For each property, the control designer assigns a **property name**, such as `BackColor or GridCellEffect`, and a **property type**, such as string, integer, or double. There's even a picture type for bitmaps and icons. The client program can set an individual ActiveX control property by specifying the property's integer index and its value. The client can get a property by specifying the index and accepting the appropriate return value. In certain cases, ClassWizard lets you define data members in your client window class that are associated with the properties of the controls the client class contains. The generated **Dialog Data Exchange** (DDX) code exchanges data between the control properties and the client class data members.
ActiveX Controls methods are like functions. A method has a symbolic name, a set of parameters, and a return value. You call a method by calling a C++ member function of the class that represents the control. A control designer can define any needed methods, such as `PreviousYear()`, `LowerControlRods()`, and so forth.
An ActiveX control doesn't send `WM_` notification messages to its container the way ordinary controls do; instead, it "fires events." An **event** has a symbolic name and can have an arbitrary sequence of parameters; it's really a container function that the control calls. Like ordinary control notification messages, events don't return a value to the ActiveX control. Examples of events are **Click**, **KeyDown**, and **NewMonth**. Events are mapped in your client class just as control notification messages are.
In the MFC world, ActiveX controls act just like child windows, but there's a significant layer of code between the container window and the control window. In fact, the control might not even have a window. When you call `Create()`, the control's window isn't created directly; instead, the control code is loaded and given the command for "in-place activation." The ActiveX control then creates its own window, which MFC lets you access through a `CWnd` pointer. It's not a good idea for the client to use the control's `hWnd` directly, however.
A DLL is used to store one or more ActiveX controls, but the DLL often has an **OCX filename extension** instead of a DLL extension. Your container program loads the DLLs when it needs them, using sophisticated COM techniques that rely on the Windows Registry. For the time being, simply accept the fact that once you specify an ActiveX control at design time, it will be loaded for you at runtime. Obviously, when you ship a program that requires special ActiveX controls, you'll have to include the OCX files and an appropriate setup program.

Installing ActiveX Controls

Let's assume you've found a nifty ActiveX control that you want to use in your project. Your first step is to **copy the control's DLL to your hard disk**. You could put it anywhere, but it's easier to track your ActiveX controls if you put them in one place, such as in the system directory (typically \Windows\System for Microsoft Windows 95, Xp or \Winnt\System32 for Microsoft Windows NT, 2000). Copy associated files such as **help** (HLP) or **license** (LIC) files to the same directory. Your next step is to **register the control in the Windows Registry**. Actually, the ActiveX control registers itself when a client program calls a special exported function. The Windows utility **Regsvr32** is a client that accepts the control name on the command line. **Regsvr32** is suitable for installation scripts, but another program, **RegComp**. Some controls have licensing requirements, which might involve **extra entries to the Registry**. Licensed controls usually come with setup programs that take care of those details. After you register your ActiveX control, you must **install it in each project that uses it**. That doesn't mean that the OCX file gets copied. It means that ClassWizard generates a copy of a C++ class that's specific to the control, and it means that the control shows up in the dialog editor control palette for that project.

To install an ActiveX control in a project, choose **Add To Project** from the **Project** menu and then choose **Components And Controls**. Select **Registered ActiveX Controls**, as shown in the following illustration.

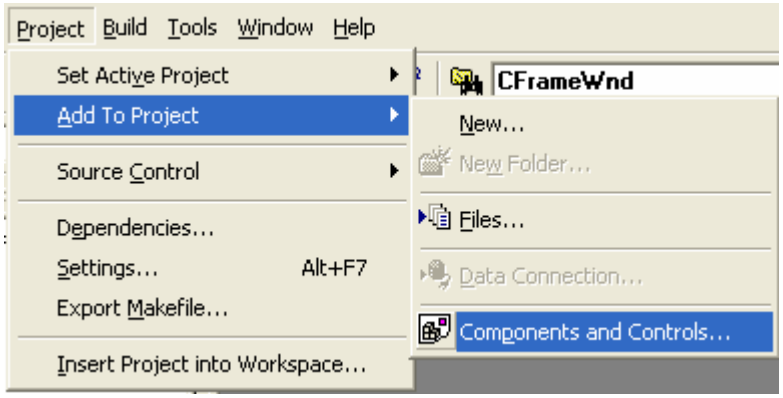

Figure 1: Adding ActiveX controls to a project.

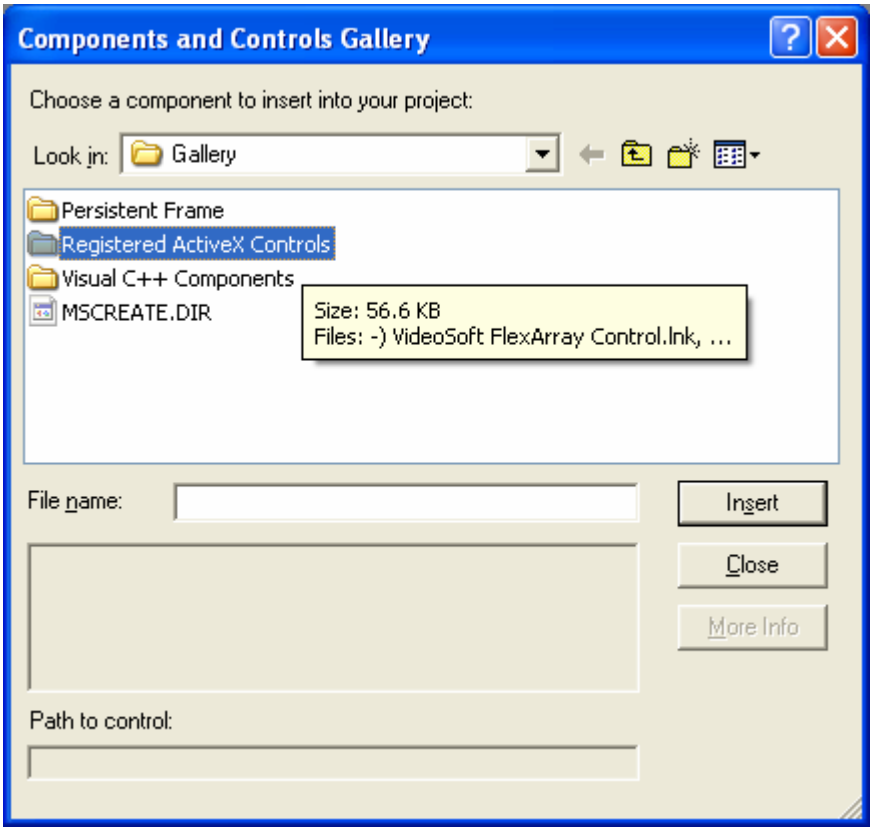

Figure 2: Browsing registered ActiveX controls.

This gets you the list of all the ActiveX controls currently registered on your system. A typical list is shown here.

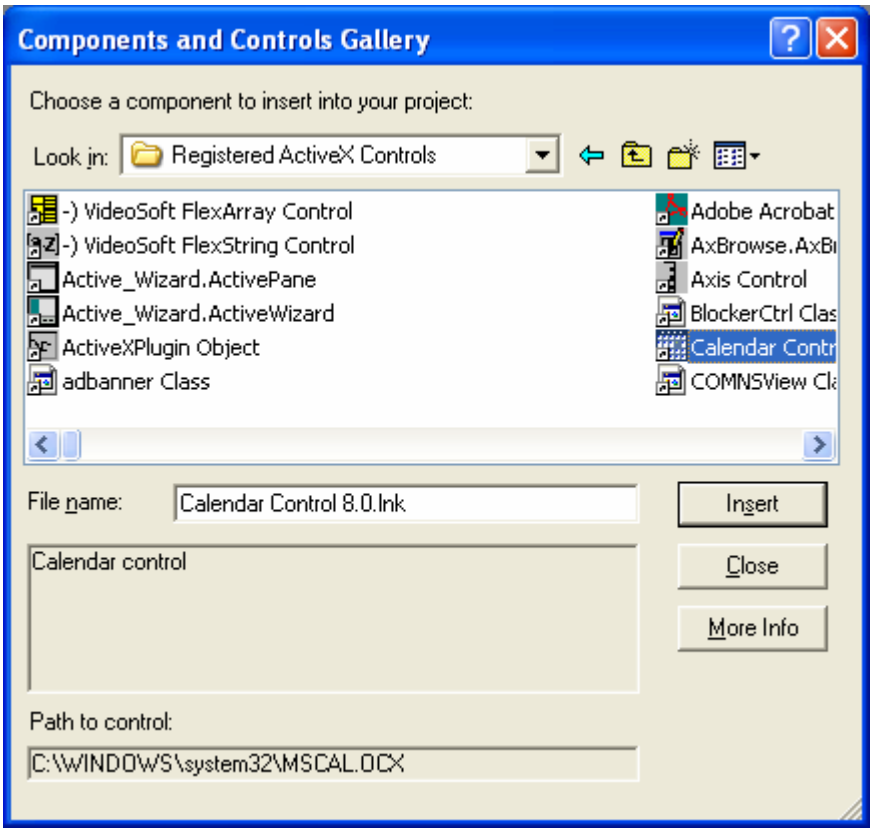

Figure 3: A list of all the ActiveX controls currently registered on the system.

The Calendar Control

The **MSCal.ocx** control is a popular **Microsoft ActiveX Calendar** control that's probably already installed and registered on your computer. Figure 4 shows the **Calendar** control inside a modal dialog.

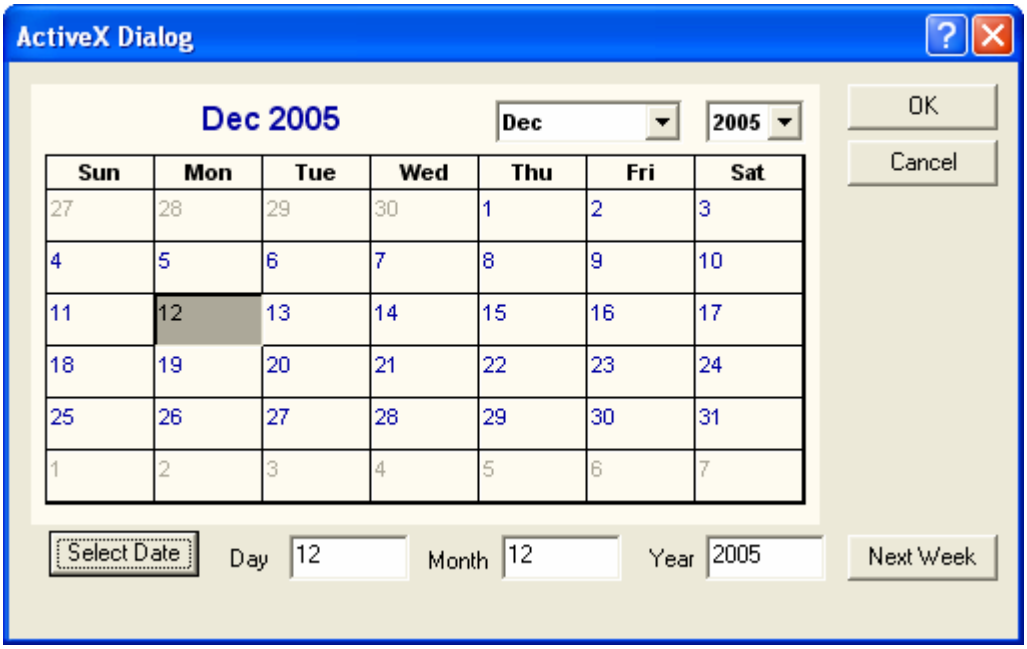

Figure 4: The **Calendar** control in use.

The **Calendar** control comes with a help file that lists the control's **properties**, **methods**, and **events** shown here.

Properties	Methods	Events
BackColor	AboutBox	AfterUpdate
Day	NextDay	BeforeUpdate
DayFont	NextMonth	Click
DayFontColor	NextWeek	DblClick
DayLength	NextYear	KeyDown
FirstDay	PreviousDay	KeyPress
GridCellEffect	PreviousMonth	KeyUp
GridFont	PreviousWeek	NewMonth
GridFontColor	PreviousYear	NewYear
GridLinesColor	Refresh	
Month	Today	
MonthLength		
ShowDateSelectors		
ShowDays		
ShowHorizontalGridlines		
ShowTitle		
ShowVerticalGridlines		
TitleFont		
TitleFontColor		
Value		
ValueIsNull		
Year		

Table 1: ActiveX Calendar's properties, methods and events.

You'll be using the **BackColor**, **Day**, **Month**, **Year**, and **Value** properties in the MYMFC24 example later in this module. **BackColor** is an unsigned long, but it is used as an `OLE_COLOR`, which is almost the same as a `COLORREF`. **Day**, **Month**, and **Year** are short integers. **Value**'s type is the special type `VARIANT`, which holds the entire date as a 64-bit value. Each of the properties, methods, and events listed above has a corresponding integer identifier. Information about the names, types, parameter sequences, and integer IDs is stored inside the control and is accessible to ClassWizard at container design time.

ActiveX Control Container Programming

MFC and ClassWizard support ActiveX controls both in dialogs and as "child windows." To use ActiveX controls, you must understand how a control grants access to properties, and you must understand the interactions between your DDX code and those property values.

Property Access

The ActiveX control developer designates certain properties for access at design time. Those properties are specified in the property pages that the control displays in the dialog editor when you right-click on a control and choose **Properties**. The **Calendar** control's main property page looks like the one shown next.

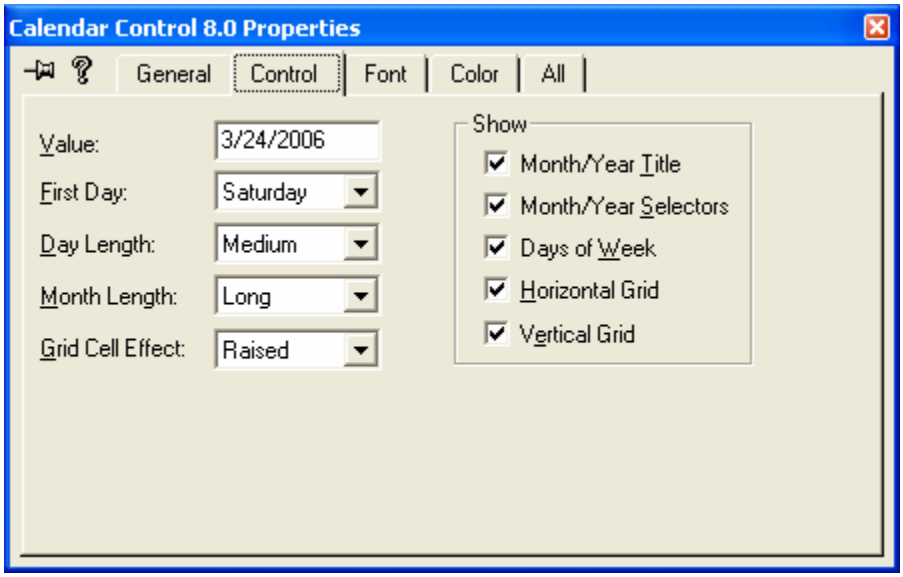

Figure 5: Calendar Control properties.

When you click on the **All** tab, you will see a list of all the **design- time**-accessible properties, which might include a few properties not found on the **Control** tab. The **Calendar** control's **All** page looks like this.

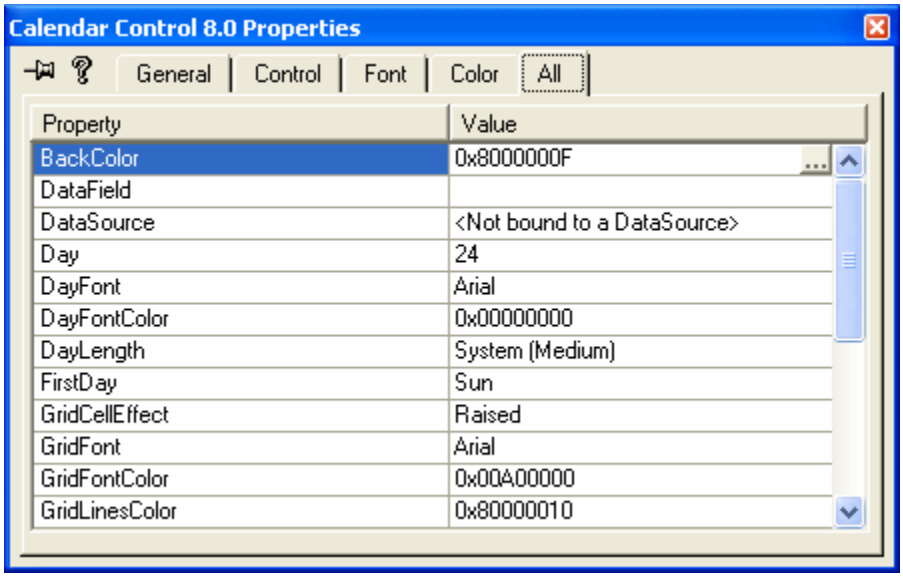

Figure 6: A list of all the **design- time**-accessible Calendar control properties.

All the control's properties, including the design-time properties, are accessible at runtime. Some properties, however, might be designated as read-only.

ClassWizard's C++ Wrapper Classes for ActiveX Controls

When you insert an ActiveX control into a project, ClassWizard generates a **C++ wrapper class**, derived from `CWnd`, which is tailored to your control's methods and properties. The class has member functions for all properties and methods, and it has constructors that you can use to dynamically create an instance of the control. ClassWizard also generates wrapper classes for objects used by the control. Following are a few typical member functions from the file **Calendar.cpp** that ClassWizard generates for the **Calendar** control:

```
unsigned long CCalendar::GetBackColor()
{
    unsigned long result;
    InvokeHelper(DISPID_BACKCOLOR, DISPATCH_PROPERTYGET, VT_I4, (void*)&result,
NULL);
    return result;
}

void CCalendar::SetBackColor(unsigned long newValue)
{
    static BYTE parms[] = VTS_I4;
    InvokeHelper(DISPID_BACKCOLOR, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
newValue);
}

short CCalendar::GetDay()
{
    short result;
    InvokeHelper(0x11, DISPATCH_PROPERTYGET, VT_I2,
                 (void*)&result, NULL);
    return result;
}

void CCalendar::SetDay(short nNewValue)
{
    static BYTE parms[] = VTS_I2;
    InvokeHelper(0x11, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms, nNewValue);
}
```

```
COleFont CCalendar::GetDayFont()
{
    LPDISPATCH pDispatch;
    InvokeHelper(0x1, DISPATCH_PROPERTYGET, VT_DISPATCH, (void*)&pDispatch, NULL);
    return COleFont(pDispatch);
}

void CCalendar::SetDayFont(LPDISPATCH newValue)
{
    static BYTE parms[] = VTS_DISPATCH;
    InvokeHelper(0x1, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms, newValue);
}

VARIANT CCalendar::GetValue()
{
    VARIANT result;
    InvokeHelper(0xc, DISPATCH_PROPERTYGET, VT_VARIANT, (void*)&result, NULL);
    return result;
}

void CCalendar::SetValue(const VARIANT& newValue)
{
    static BYTE parms[] = VTS_VARIANT;
    InvokeHelper(0xc, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms, &newValue);
}

void CCalendar::NextDay()
{
    InvokeHelper(0x16, DISPATCH_METHOD, VT_EMPTY, NULL, NULL);
}

void CCalendar::NextMonth()
{
    InvokeHelper(0x17, DISPATCH_METHOD, VT_EMPTY, NULL, NULL);
}
```

You don't have to concern yourself too much with the code inside these functions, but you can match up the first parameter of each `InvokeHelper()` function with the **dispatch ID** for the corresponding property or method in the **Calendar** control property list. As you can see, properties always have separate `Set()` and `Get()` functions. To call a method, you simply call the corresponding function. For example, to call the `NextDay()` method from a dialog class function, you write code such as this:

```
m_calendar.NextDay();
```

In this case, `m_calendar` is an object of class `CCalendar`, the wrapper class for the **Calendar** control.

AppWizard Support for ActiveX Controls

When the AppWizard ActiveX Controls option is checked (the default), AppWizard inserts the following line in your application class `InitInstance()` member function:

```
AfxEnableControlContainer();
```

It also inserts the following line in the project's **StdAfx.h** file:

```
#include <afxdisp.h>
```

If you decide to add ActiveX controls to an existing project that doesn't include the two lines above, you can simply add the lines.

ClassWizard and the Container Dialog

Once you've used the dialog editor to generate a dialog template, you already know that you can use ClassWizard to generate a C++ class for the dialog window. If your template contains one or more ActiveX controls, you can use ClassWizard to add data members and event handler functions.

Dialog Class Data Members vs. Wrapper Class Usage

What kind of data members can you add to the dialog for an ActiveX control? If you want to set a control property before you call `DoModal()` for the dialog, you can add a dialog data member for that property. If you want to change properties inside the dialog member functions, you must take another approach: you add a data member that is an object of the wrapper class for the ActiveX control.
Now is a good time to review the MFC DDX logic. Look back at the dialog in Module 5. The `CDialog::OnInitDialog` function calls `CWnd::UpdateData(FALSE)` to read the dialog class data members, and the `CDialog::OnOK` function calls `UpdateData(TRUE)` to write the members. Suppose you added a data member for each ActiveX control property and you needed to get the **Value** property value in a button handler. If you called `UpdateData(FALSE)` in the button handler, it would read all the property values from all the dialog's controls, clearly a waste of time. It's more effective to avoid using a data member and to call the wrapper class `Get()` function instead. To call that function, you must first tell ClassWizard to add a wrapper class object data member. Suppose you have a **Calendar** wrapper class `CCalendar` and you have an `m_calendar` data member in your dialog class. If you want to get the **Value** property, you do it like this:

```
COleVariant var = m_calendar.GetValue();
```

Now consider another case: you want to set the day to the 5th of the month before the control is displayed. To do this by hand, add a dialog class data member `m_sCalDay` that corresponds to the control's short integer **Day** property. Then add the following line to the `DoDataExchange()` function:

```
DDX_OCShort(pDX, ID_CALENDAR1, 0x11, m_sCalDay);
```

The third parameter is the **Day** property's integer index (its `DispID`), which you can find in the `GetDay()` and `SetDay()` functions generated by ClassWizard for the control. Here's how you construct and display the dialog:

```
CMyDialog dlg;
dlg.m_sCalDay = 5;
dlg.DoModal();
```

The DDX code takes care of setting the property value from the data member before the control is displayed. No other programming is needed. As you would expect, the DDX code sets the data member from the property value when the user clicks the OK button.
Even when ClassWizard correctly detects a control's properties, it can't always generate data members for all of them. In particular, no DDX functions exist for `VARIANT` properties like the **Calendar**'s **Value** property. You'll have to use the wrapper class for these properties.

Mapping ActiveX Control Events

ClassWizard lets you map **ActiveX control events** the same way you map **Windows messages and command messages** from controls. If a dialog class contains one or more ActiveX controls, ClassWizard adds and maintains an event sink map that connects mapped events to their handler functions. It works something like a message map. You can see the code in Listing 4. ActiveX controls have the annoying habit of **firing events before your program is ready for them**. If your event handler uses windows or pointers to C++ objects, it should verify the validity of those entities prior to using them.

Locking ActiveX Controls in Memory

Normally, an ActiveX control remains mapped in your process as long as its parent dialog is active. That means it must be reloaded each time the user opens a modal dialog. The reloads are usually quicker than the initial load because of disk caching, but you can lock the control into memory for better performance. To do so, add the following line in the overridden `OnInitDialog()` function after the base class call:

```
AfxOleLockControl(m_calendar.GetClsid());
```

The ActiveX control remains mapped until your program exits or until you call the `AfxOleUnlockControl()` function.

The MYMFC24 Example: An ActiveX Control Dialog Container

Now it's time to build an application that uses a **Calendar** control in a dialog. Here are the steps to create the MYMFC24 example:

Run AppWizard to produce \mfcproject\mymfc24. Accept all of the default settings but two: select **Single Document** and deselect **Printing And Print Preview**. In the AppWizard Step 3 dialog, make sure the **ActiveX Controls** option is selected, as shown below.

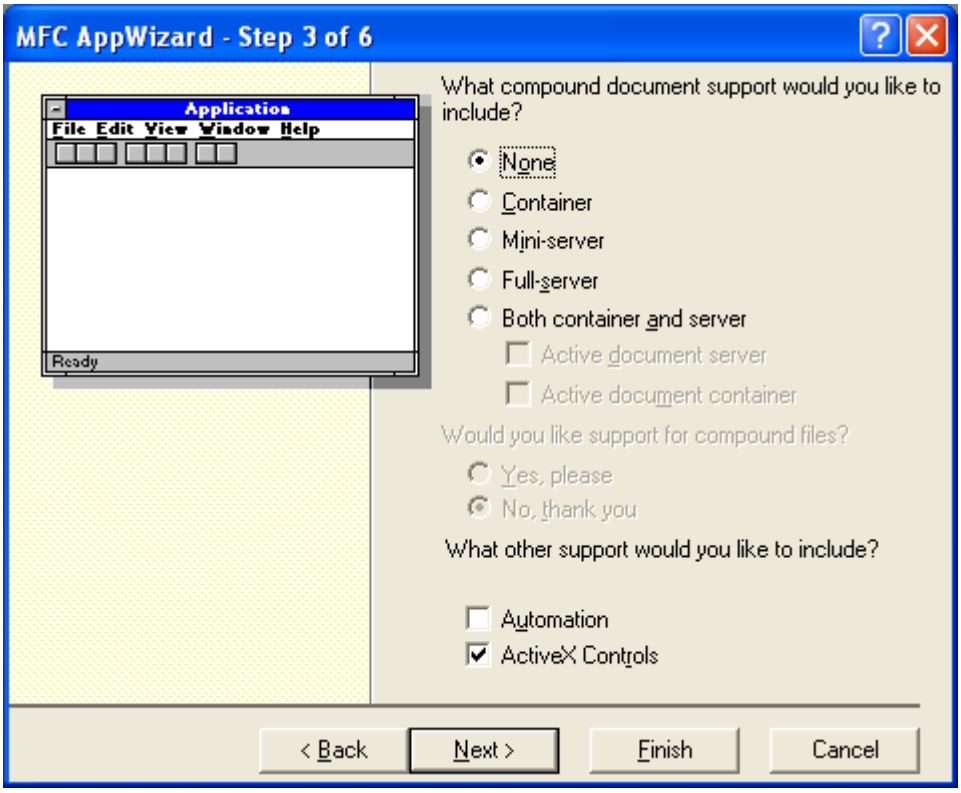

Figure 7: AppWizard step 3 of 6, enabling ActiveX control option.

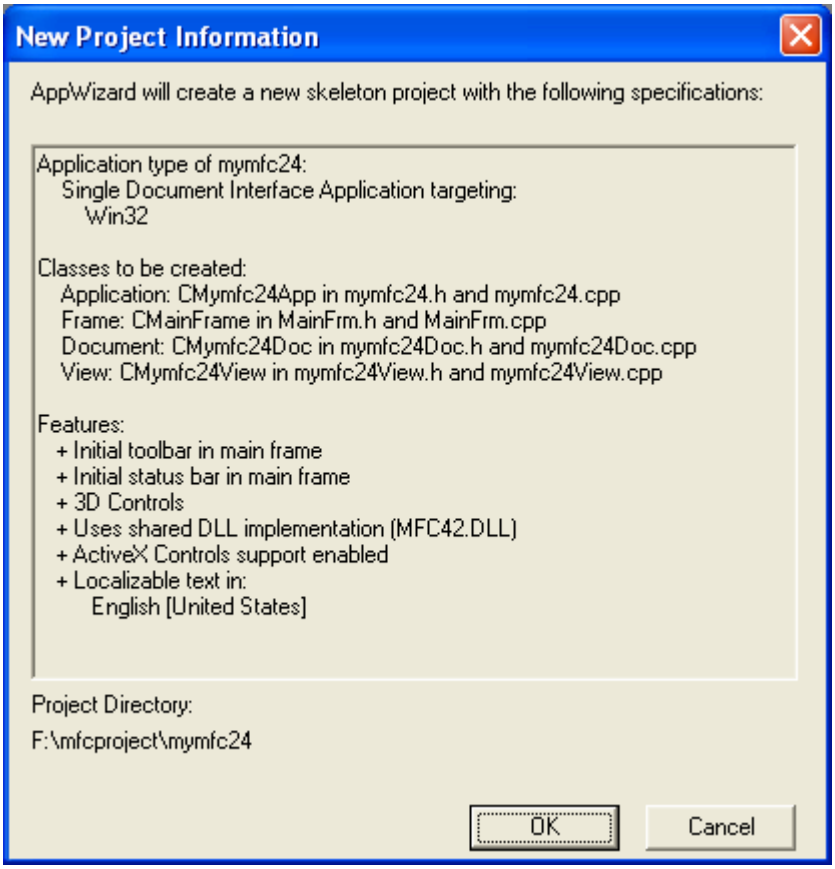

Figure 8: MYMFC24 ActiveX control project summary.

Verify that the **Calendar** control is registered. If the control does not appear in the Visual C++ **Gallery's Registered ActiveX Controls** page, copy the files **MSCal.ocx**, **MSCal.hlp**, and **MSCal.cnt** to your system directory and register the control by running the **REGCOMP** program.

Install the **Calendar** control in the MYMFC24 project. Choose **Add To Project** from Visual C++'s **Project** menu, and then choose **Components And Controls**. Choose **Registered ActiveX Controls**, and then choose **Calendar Control 8.0**.

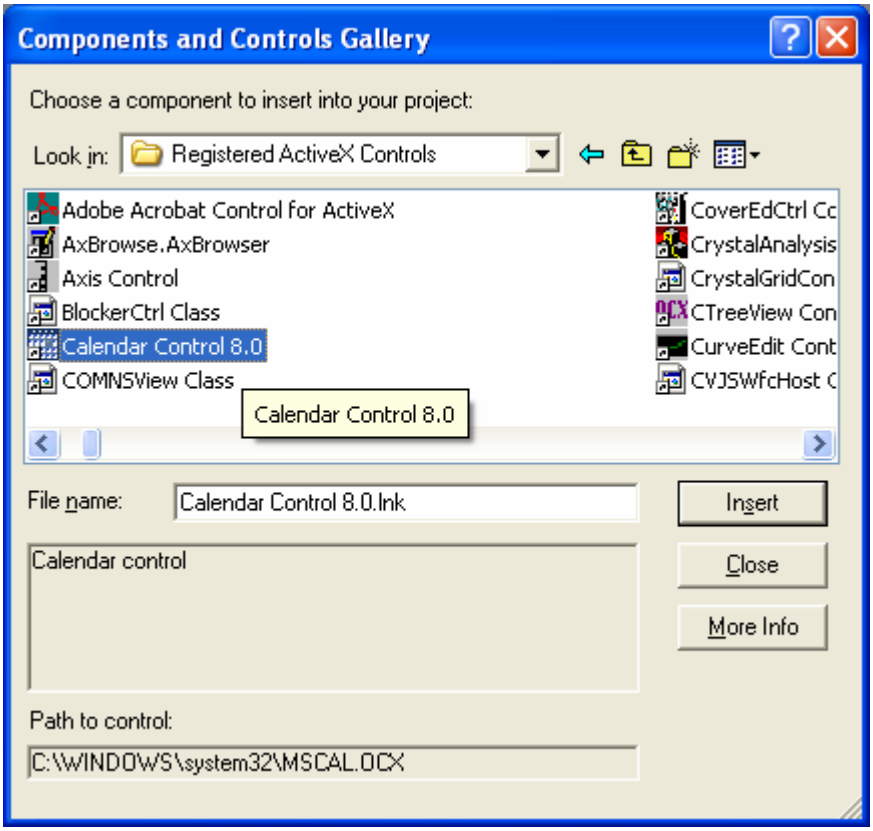

Figure 9: Installing the **Calendar** control in the MYMFC24 project.

ClassWizard generates two classes in the MYMFC24 directory, as shown here or through the ClassView.

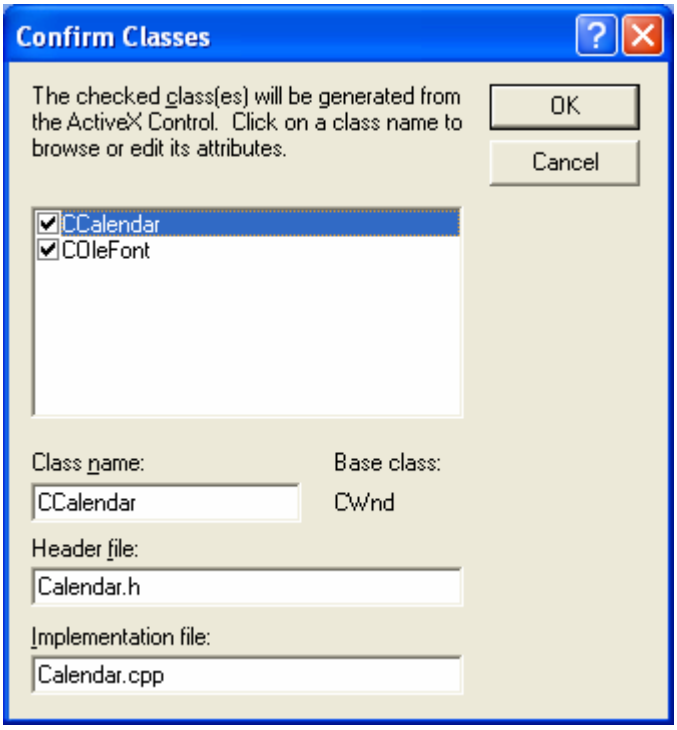

Figure 10: Two classes, `CCalendar` and `COleFont` generated in the MYMFC24 directory.

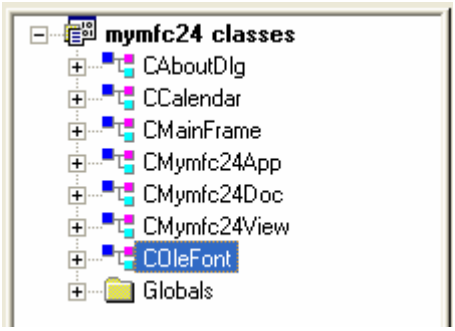

Figure 11: The generated classes viewed in ClassView.

Edit the **Calendar** control class to handle help messages. Add **Calendar.cpp** to the following message map code:

```
BEGIN_MESSAGE_MAP(CCalendar, CWnd)
    ON_WM_HELPINFO()
END_MESSAGE_MAP()
```

```
// CCalendar

IMPLEMENT_DYNCREATE(CCalendar, CWnd)

BEGIN_MESSAGE_MAP(CCalendar, CWnd)
    ON_WM_HELPINFO()
END_MESSAGE_MAP()
```

Listing 1.

In the same file, add the `OnHelpInfo()` function:

```
BOOL CCalendar::OnHelpInfo(HELPINFO* pHelpInfo)
{
    // Edit the following string for your system
    ::WinHelp(GetSafeHwnd(), "c:\\winnt\\system32\\mscal.hlp", HELP_FINDER, 0);
    return FALSE;
}
```

```
BOOL CCalendar::OnHelpInfo(HELPINFO* pHelpInfo)
{
    // Edit the following string for your system
    ::WinHelp(GetSafeHwnd(), "c:\\winnt\\system32\\mscal.hlp", HELP_FINDER, 0);
    return FALSE;
}
```

Listing 2.

In **Calendar.h**, add the function prototype and declare the message map:

```
protected:
    afx_msg BOOL OnHelpInfo(HELPINFO* pHelpInfo);
    DECLARE_MESSAGE_MAP()
```

```
protected:
    DECLARE_DYNCREATE(CCalendar)

protected:
    afx_msg BOOL OnHelpInfo(HELPINFO* pHelpInfo);
    DECLARE_MESSAGE_MAP()
```

Listing 3.

The `OnHelpInfo()` function is called if the user presses the **F1** key when the **Calendar** control has the input focus. We have to add the message map code by hand because ClassWizard doesn't modify generated ActiveX classes.

The `ON_WM_HELPINFO` macro maps the `WM_HELP` message, which is new to Microsoft Windows 95 and Microsoft Windows NT 4.0. You can use `ON_WM_HELPINFO` in any view or dialog class and then code the handler to activate any help system. Module 15 describes the MFC context-sensitive help system, some of which predates the `WM_HELP` message.

Use the dialog editor to create a new dialog resource. Choose **Resource** from Visual C++'s **Insert** menu, and then choose **Dialog**.

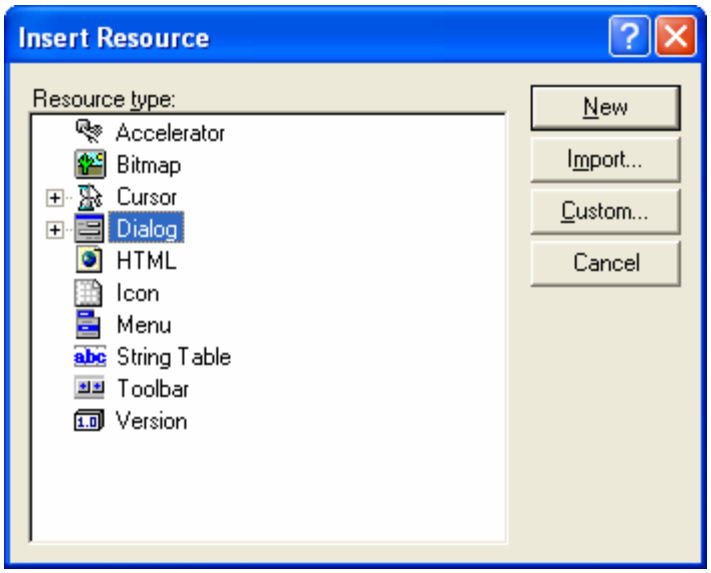

Figure 12: Inserting a new dialog to MYMFC24 project.

The dialog editor assigns the ID `IDD_DIALOG1` to the new dialog. Next change the ID to `IDD_ACTIVEXDIALOG`, change the dialog caption to **ActiveX Dialog**, and set the dialog's **Context Help** property on the **More Styles** page.

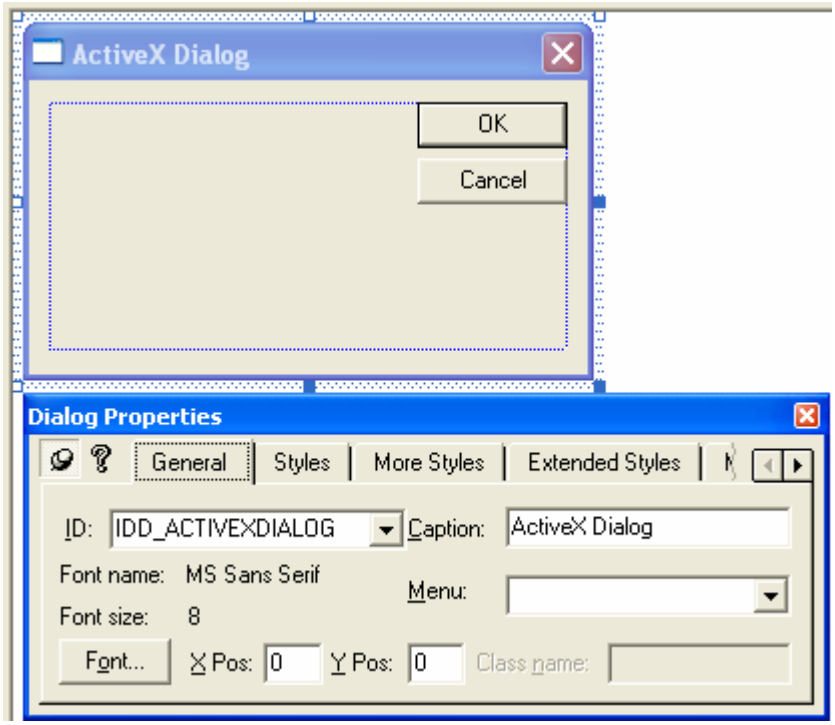

Figure 13: Modifying dialog properties.

Accept the default **OK** and **Cancel** buttons with the IDs `IDOK` and `IDCANCEL`, and then add the other controls as shown in Figure 4. Drag the **Calendar** control from the control palette. Assign control IDs as shown in the following table.

Control	ID
Calendar control	`IDC_CALENDAR1`
Select Date button	`IDC_SELECTDATE`
Edit control	`IDC_DAY`
Edit control	`IDC_MONTH`
Edit control	`IDC_YEAR`
Next Week button	`IDC_NEXTWEEK`

Table 2.

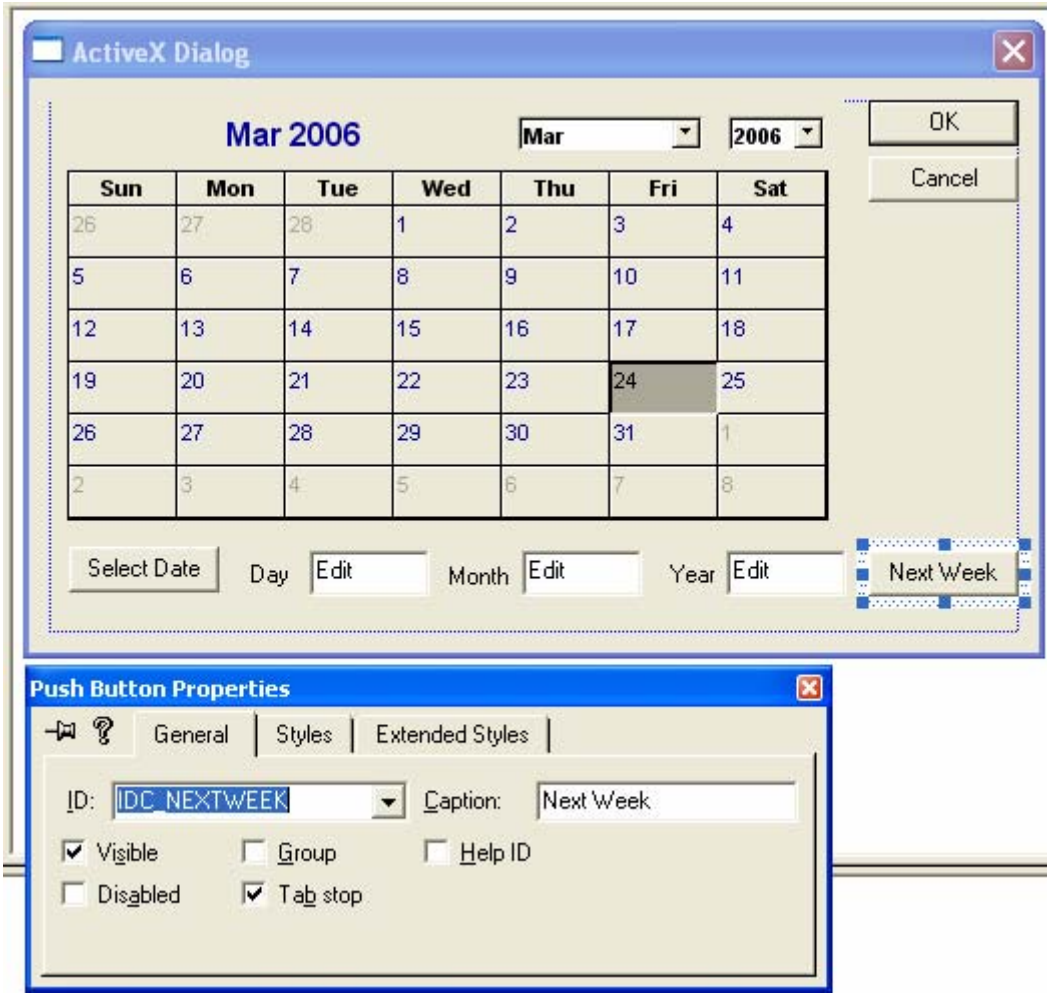

Figure 14: Adding ActiveX Calendar and other controls to the dialog.

Make the **Select Date** button the default button. Then set an appropriate tab order.

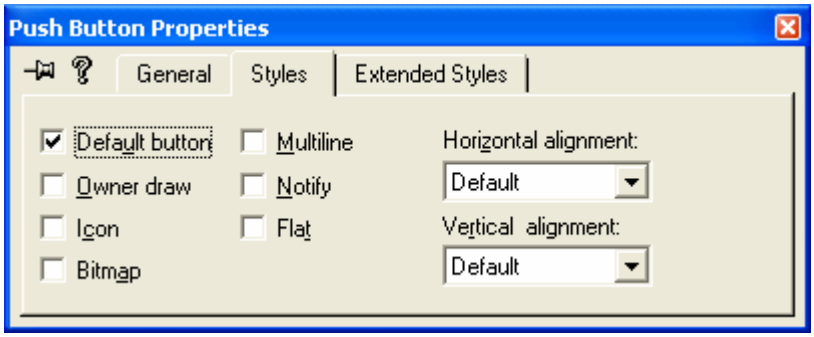

Figure 15: Modifying button properties.

Use ClassWizard to create the `CActiveXDialog` class. If you run ClassWizard directly from the dialog editor window, it will know that you want to create a `CDialog`-derived class based on the `IDD_ACTIVEXDIALOG` template. Simply accept the default options, and name the class `CActiveXDialog`.

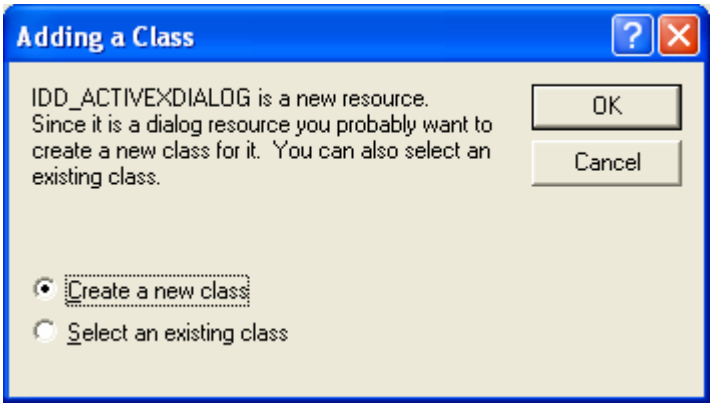

Figure 16: Creating a new class dialog prompt.

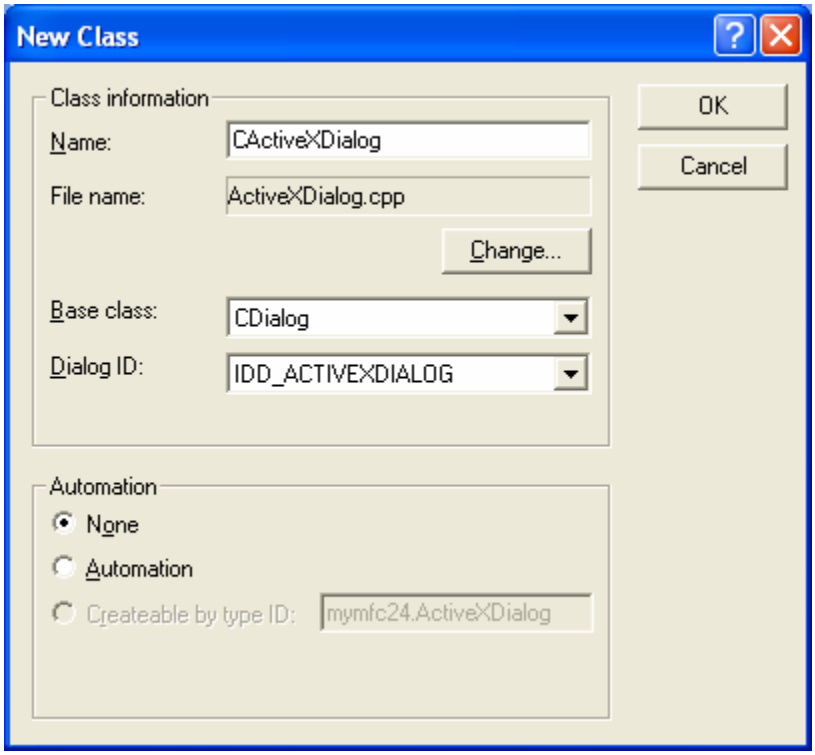

Figure 17: New class `CActiveXDialog` information.

Click on the ClassWizard **Message Maps** tab, and then add the message handler functions shown in the table below. To add a message handler function, click on an object ID, click on a message, and click the **Add Function** button. If the **Add Member Function** dialog box appears, type the function name and click the **OK** button.

Object ID	Message	Member Function
`CActiveXDialog`	`WM_INITDIALOG`	`OnInitDialog()` (virtual function)
`IDC_CALENDAR1`	`NewMonth (event)`	`OnNewMonthCalendar1()`
`IDC_SELECTDATE`	`BN_CLICKED`	`OnSelectDate()`
`IDC_NEXTWEEK`	`BN_CLICKED`	`OnNextWeek()`
`IDOK`	`BN_CLICKED`	`OnOK()` (virtual function)

Table 3

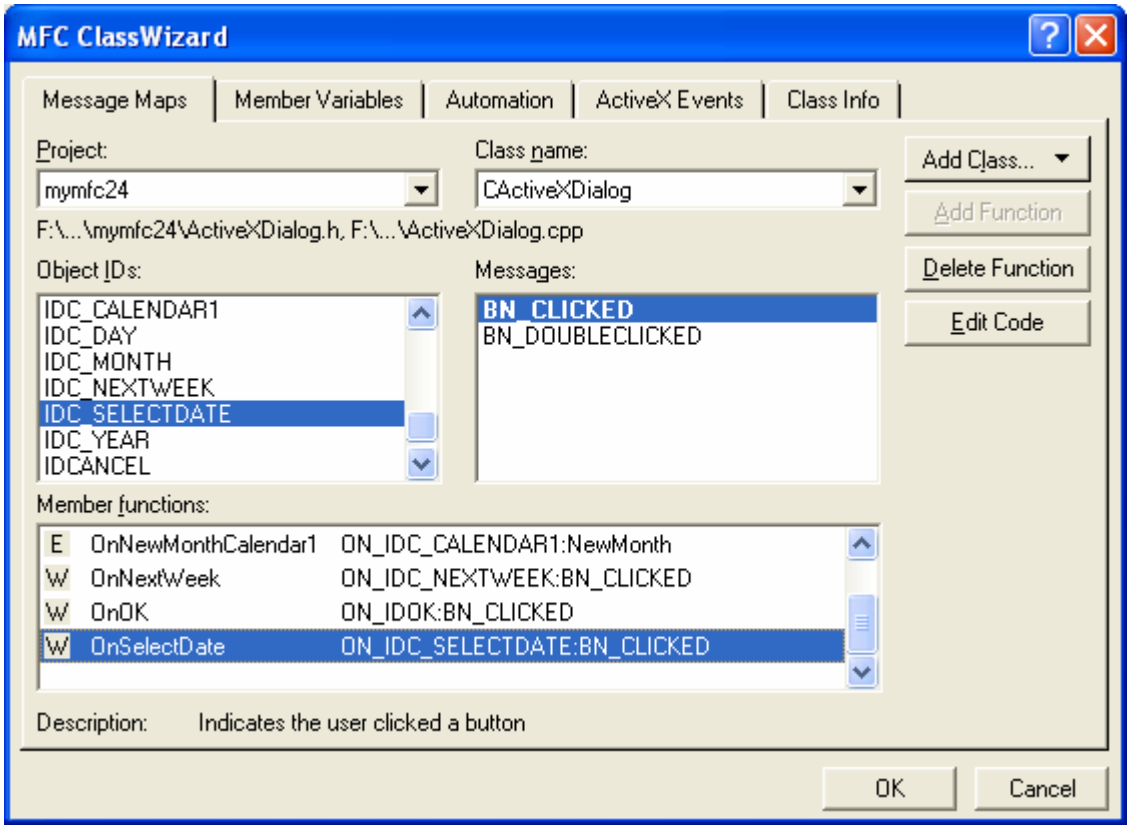

Figure 18: Adding message handler functions of the objects.

Use ClassWizard to add data members to the `CActiveXDialog` class. Click on the **Member Variables** tab, and then add the data members as shown in the illustration below.

You might think that the ClassWizard **ActiveX Events** tab is for mapping ActiveX control events in a container. That's not true: it's for ActiveX control developers who are defining events for a control.

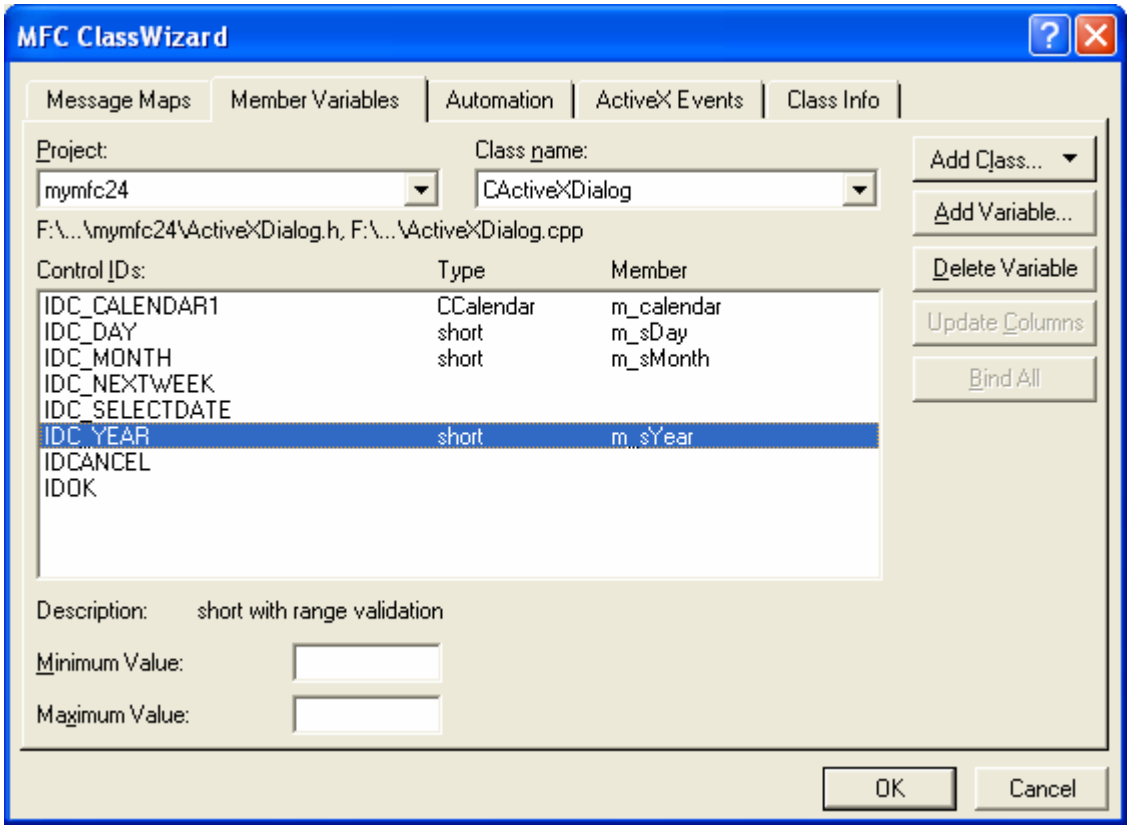

Figure 19: Adding member variables for the controls.

Edit the `CActiveXDialog` class. Add the `m_varValue` and `m_BackColor` data members and then edit the code for the five handler functions `OnInitDialog()`, `OnNewMonthCalendar1()`, `OnSelectDate()`, `OnNextWeek()`, and `OnOK()`. Listing 4 shows all the code for the dialog class, with new code added.

ACTIVEXDIALOG.H

```
//{{AFX_INCLUDES()
#include "calendar.h"
//}}AFX_INCLUDES
#if
!defined(AFX_ACTIVEXDIALOG_H__1917789D_6F24_11D0_8FD9_00C04FC2A0C2__INCLUDED_)
#define AFX_ACTIVEXDIALOG_H__1917789D_6F24_11D0_8FD9_00C04FC2A0C2__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

// ActiveXDialog.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// CActiveXDialog dialog
class CActiveXDialog : public CDialog

{
// Construction
public:
    CActiveXDialog(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
    //{{AFX_DATA(CActiveXDialog)
    enum { IDD = IDD_ACTIVEXDIALOG };
    CCalendar    m_calendar;
```

```
    short    m_sDay;
    short    m_sMonth;
    short    m_sYear;
    //}}AFX_DATA
    COleVariant m_varValue;
    unsigned long m_BackColor;

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CActiveXDialog)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV
                                                     //  support
    //}}AFX_VIRTUAL

// Implementation
protected:

    // Generated message map functions
    //{{AFX_MSG(CActiveXDialog)
    virtual BOOL OnInitDialog();
    afx_msg void OnNewMonthCalendar1();
    afx_msg void OnSelectDate();
    afx_msg void OnNextWeek();
    virtual void OnOK();
    DECLARE_EVENTSINK_MAP()
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional
//  declarations immediately before the previous line.

#endif //
!defined(AFX_ACTIVEXDIALOG_H__1917789D_6F24_11D0_8FD9_00C04FC2A0C2__INCLUDED_)
```

ACTIVEXDIALOG.CPP

```
// ActiveXDialog.cpp : implementation file
//

#include "stdafx.h"
#include "mymfc24.h"
#include "ActiveXDialog.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CActiveXDialog dialog


CActiveXDialog::CActiveXDialog(CWnd* pParent /*=NULL*/) :
CDialog(CActiveXDialog::IDD, pParent)
{
    //{{AFX_DATA_INIT(CActiveXDialog)
    m_sDay = 0;
    m_sMonth = 0;
    m_sYear = 0;
    //}}AFX_DATA_INIT
    m_BackColor = 0x8000000F;
}
```

```
void CActiveXDialog::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CActiveXDialog)
    DDX_Control(pDX, IDC_CALENDAR1, m_calendar);
    DDX_Text(pDX, IDC_DAY, m_sDay);
    DDX_Text(pDX, IDC_MONTH, m_sMonth);
    DDX_Text(pDX, IDC_YEAR, m_sYear);
    //}}AFX_DATA_MAP
    DDX_OCColor(pDX, IDC_CALENDAR1, DISPID_BACKCOLOR, m_BackColor);
}

BEGIN_MESSAGE_MAP(CActiveXDialog, CDialog)
    //{{AFX_MSG_MAP(CActiveXDialog)
    ON_BN_CLICKED(IDC_SELECTDATE, OnSelectDate)
    ON_BN_CLICKED(IDC_NEXTWEEK, OnNextWeek)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CActiveXDialog message handlers

BEGIN_EVENTSINK_MAP(CActiveXDialog, CDialog)
    //{{AFX_EVENTSINK_MAP(CActiveXDialog)
    ON_EVENT(CActiveXDialog, IDC_CALENDAR1, 3 /* NewMonth */,
OnNewMonthCalendar1, VTS_NONE)
    //}}AFX_EVENTSINK_MAP
END_EVENTSINK_MAP()

BOOL CActiveXDialog::OnInitDialog()
{
    CDialog::OnInitDialog();
    m_calendar.SetValue(m_varValue); // no DDX for VARIANTs
    return TRUE;  // return TRUE unless you set the focus to a control
                  // EXCEPTION: OCX Property Pages should return FALSE

}
void CActiveXDialog::OnNewMonthCalendar1()
{
    AfxMessageBox("EVENT:  CActiveXDialog::OnNewMonthCalendar1");
}

void CActiveXDialog::OnSelectDate()
{
    CDataExchange dx(this, TRUE);
    DDX_Text(&dx, IDC_DAY, m_sDay);
    DDX_Text(&dx, IDC_MONTH, m_sMonth);
    DDX_Text(&dx, IDC_YEAR, m_sYear);
    m_calendar.SetDay(m_sDay);
    m_calendar.SetMonth(m_sMonth);
    m_calendar.SetYear(m_sYear);
}

void CActiveXDialog::OnNextWeek()
{
    m_calendar.NextWeek();
}

void CActiveXDialog::OnOK()
{
    CDialog::OnOK();
    m_varValue = m_calendar.GetValue(); // no DDX for VARIANTs
}
```

Listing 4: Code for the `CActiveXDialog` class.

The `OnSelectDate()` function is called when the user clicks the **Select Date** button. The function gets the day, month, and year values from the three edit controls and transfers them to the control's properties. ClassWizard can't add DDX code for the **BackColor** property, so you must add it by hand. In addition, there's no DDX code for `VARIANT` types, so you must add code to the `OnInitDialog()` and `OnOK()` functions to set and retrieve the date with the control's **Value** property.
Connect the dialog to the view. Use ClassWizard to map the `WM_LBUTTONDOWN` message, and then edit the handler function as follows:

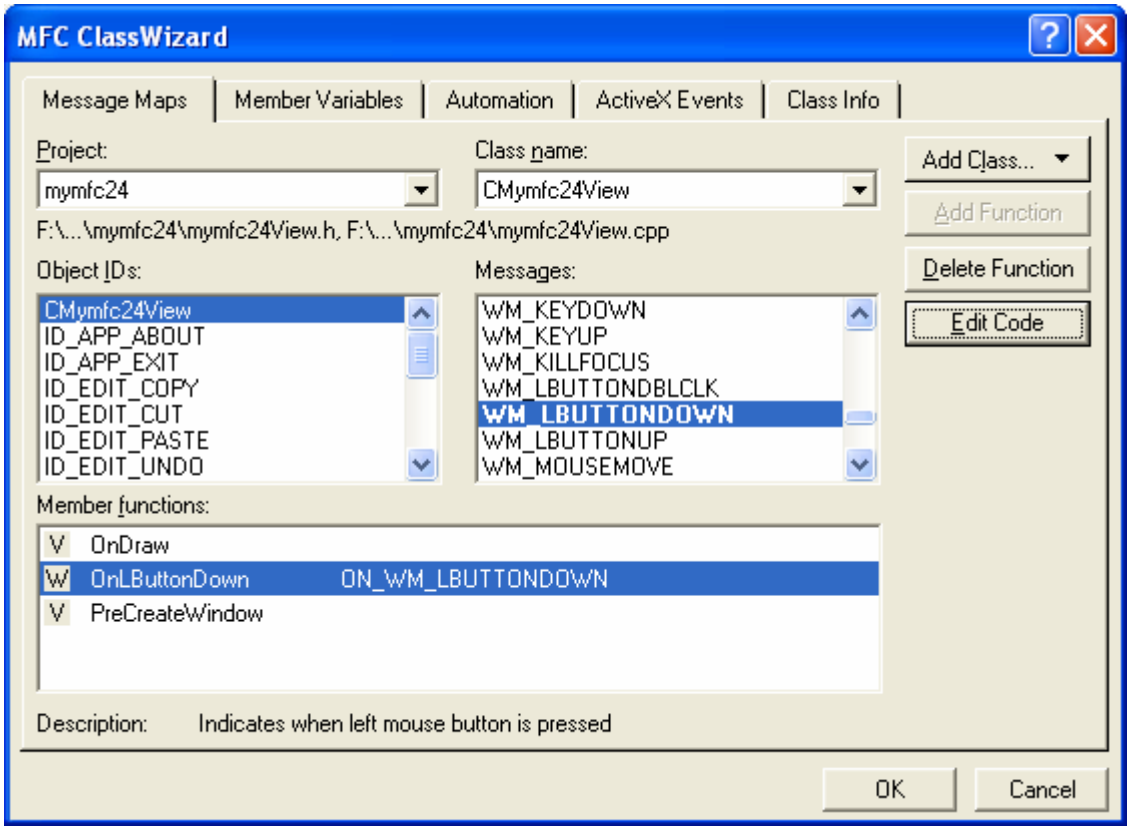

Figure 20: Mapping the `WM_LBUTTONDOWN` message.

```
void CMymfc24View::OnLButtonDown(UINT nFlags, CPoint point)
{
    CActiveXDialog dlg;
    dlg.m_BackColor = RGB(255, 251, 240); // light yellow
    COleDateTime today = COleDateTime::GetCurrentTime();
    dlg.m_varValue = COleDateTime(today.GetYear(), today.GetMonth(),
today.GetDay(), 0, 0, 0);
    if (dlg.DoModal() == IDOK) {
        COleDateTime date(dlg.m_varValue);
        AfxMessageBox(date.Format("%B %d, %Y"));
    }
}
```

```
void CMymfc24View::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    CActiveXDialog dlg;
    dlg.m_BackColor = RGB(255, 251, 240); // light yellow
    COleDateTime today = COleDateTime::GetCurrentTime();
    dlg.m_varValue = COleDateTime(today.GetYear(), today.GetMonth(),
                                  today.GetDay(), 0, 0, 0);
    if (dlg.DoModal() == IDOK) {
        COleDateTime date(dlg.m_varValue);
        AfxMessageBox(date.Format("%B %d, %Y"));
    }
}
```

Listing 5.

The code sets the background color to light yellow and the date to today's date, displays the modal dialog, and reports the date returned by the **Calendar** control. You'll need to include **ActiveXDialog.h** in **mymfc24View.cpp**.

```
#include "stdafx.h"
#include "mymfc24.h"

#include "mymfc24Doc.h"
#include "mymfc24View.h"
#include "ActiveXDialog.h"

#ifdef _DEBUG
```

Listing 6.

Edit the virtual `OnDraw()` function in the file **mymfc24View.cpp**. To prompt the user to press the left mouse button, replace the code in the view class `OnDraw()` function with this single line:

```
pDC->TextOut(0, 0, "Press the left mouse button here.");
```

```
// CMymfc24View drawing

void CMymfc24View::OnDraw(CDC* pDC)
{
    // TODO: add draw code for native data here
    pDC->TextOut(0, 0, "Press the left mouse button here.");
}
```

Listing 7.

Build and test the MYMFC24 application. Open the dialog, enter a date in the three edit controls, and then click the Select **Date** button. Click the **Next Week** button. Try moving the selected date directly to a new month, and observe the message box that is triggered by the **NewMonth** event. Watch for the final date in another message box when you click **OK**. Press the **F1** key for help on the **Calendar** control.

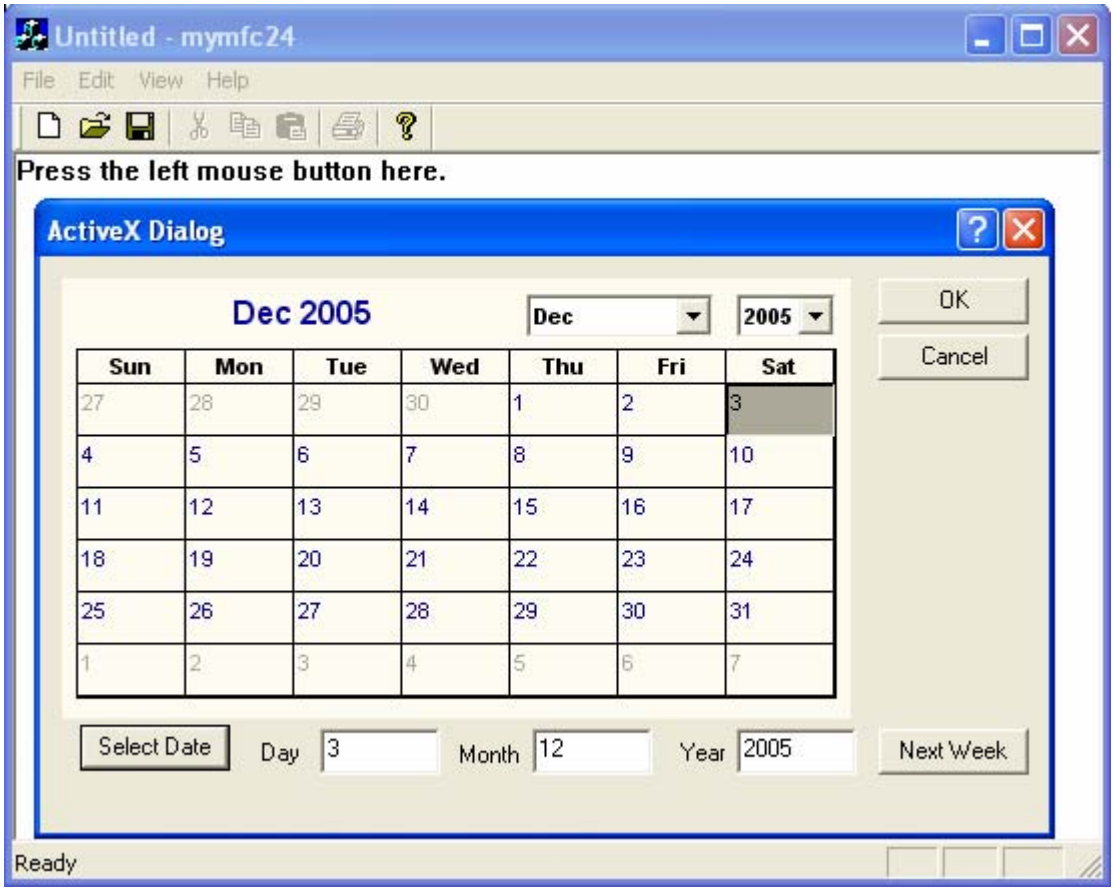

Figure 21: MYMFC24 program output, showing the ActiveX control – a calendar.

For Win32 Programmers

If you use a text editor to look inside the **mymfc24.rc** file, you might be quite mystified. Here's the entry for the **Calendar** control in the **ActiveX Dialog** template:

```
CONTROL     "",IDC_CALENDAR1,
            "{8E27C92B-1264-101C-8A2F-040224009C02}",
            WS_TABSTOP,7,7,217,113
```

There's a 32-digit number sequence where the window class name should be. What's going on? Actually, the resource template isn't the one that Windows sees. The `CDialog::DoModal` function "preprocesses" the resource template before passing it on to the dialog box procedure within Windows. It strips out all the ActiveX controls and creates the dialog window without them. Then it loads the controls (based on their 32-digit identification numbers, called **CLSID**s) and activates them in place, causing them to create their own windows in the correct places. The initial values for the properties you set in the dialog editor are stored in binary form inside the project's custom **DLGINIT** resource.
When the modal dialog runs, the MFC code coordinates the messages sent to the dialog window both by the ordinary controls and by the ActiveX controls. This allows the user to tab between all the controls in the dialog, even though the ActiveX controls are not part of the actual dialog template.
When you call the member functions for the control object, you might think you're calling functions for a child window. The control window is quite far removed, but MFC steps in to make it seem as if you're communicating with a real child window. In ActiveX terminology, the container owns a site, which is not a window. You call functions for the site, and ActiveX and MFC make the connection to the underlying window in the ActiveX control.

The container window is an object of a class derived from `CWnd`. The control site is also an object of a class derived from `CWnd`, the ActiveX control wrapper class. That means that the `CWnd` class has built-in support for both containers and sites.

What you're seeing here is MFC ActiveX control support grafted onto regular Windows. Maybe some future Windows version will have more direct support for **ActiveX Controls**. As a matter of fact, ActiveX versions of the Windows common controls already exist.

ActiveX Controls in HTML Files

You've seen the ActiveX **Calendar** control in an MFC modal dialog. You can use the same control in a Web page. The following HTML code will work (assuming the person reading the page has the **Calendar** control installed and registered on his or her machine):

```
<OBJECT
    CLASSID="clsid:8E27C92B-1264-101C-8A2F-040224009C02"
    WIDTH=300 HEIGHT=200 BORDER=1 HSPACE=5 ID=calendar>
<PARAM NAME="Day" VALUE=7>
<PARAM NAME="Month" VALUE=11>
<PARAM NAME="Year" VALUE=1998>
</OBJECT>
```

The `CLASSID` attribute (the same number that was in the MYMFC24 dialog resource) identifies the **Calendar** control in the Registry. A browser can download an ActiveX control.

Creating ActiveX Controls at Runtime

You've seen how to use the dialog editor to insert ActiveX controls at design time. If you need to create an ActiveX control at runtime without a resource template entry, here are the programming steps:

1. Insert the component into your project. ClassWizard will create the files for a wrapper class.
2. Add an embedded ActiveX control wrapper class data member to your dialog class or other C++ window class. An embedded C++ object is then constructed and destroyed along with the window object.
3. Choose **Resource Symbols** from Visual C++'s View menu. Add an ID constant for the new control.
4. If the parent window is a dialog, use ClassWizard to map the dialog's `WM_INITDIALOG` message, thus overriding `CDialog::OnInitDialog`. For other windows, use ClassWizard to map the `WM_CREATE` message. The new function should call the embedded control class's Create member function. This call indirectly displays the new control in the dialog. The control will be properly destroyed when the parent window is destroyed.
5. In the parent window class, manually add the necessary event message handlers and prototypes for your new control. Don't forget to add the event sink map macros.

ClassWizard doesn't help you with event sink maps when you add a dynamic ActiveX control to a project. Consider inserting the target control in a dialog in another temporary project. After you're finished mapping events, simply copy the event sink map code to the parent window class in your main project.

The MYMFC24B Example: The Web Browser ActiveX Control

Microsoft Internet Explorer 4.x has become a leading Web browser. I was surprised to find out that most of its functionality is contained in one big ActiveX control, **Shdocvw.dll**. When you run Internet Explorer, you launch a small shell program that loads this Web Browser control in its main window. You can find complete documentation for the **Web Browser control**'s properties, methods, and events in the Internet SDK, downloadable from microsoft.com. This documentation is in HTML form, of course. Because of this modular architecture, you can write your own custom browser program with very little effort. MYMFC24B creates a two-window browser that displays a search engine page side-by-side with the target page, as shown here.

Figure 22: Web browser ActiveX control in action.

This view window contains two **Web Browser controls** that are sized to occupy the entire client area. When the user clicks an item in the search (right-hand) control, the program intercepts the command and routes it to the target (left-hand) control.
Here are the steps for building the example:

Make sure the Web Browser control is registered. You undoubtedly have Microsoft Internet Explorer 4.x installed, since Visual C++ 6.0 requires it, so the Web Browser control should be registered. You can download Internet Explorer from microsoft.com if necessary.
Run AppWizard to produce \mfcproject\mymfc24B. Accept all the default settings but two: except select **Single Document** and deselect **Printing And Print Preview**. Make sure the **ActiveX Controls** option is checked as in MYMFC24.

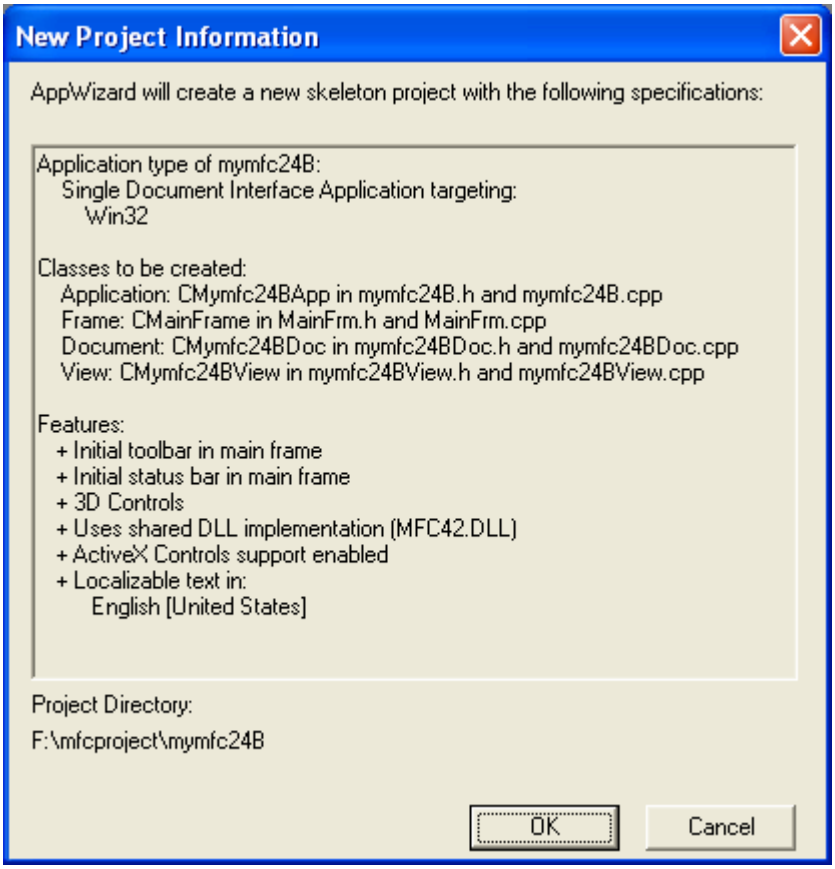

Figure 23: MYMFC24B SDI and ActiveX control project summary.

Install the Web Browser control in the MYMFC24B project. Choose **Add To Project** from Visual C++'s **Project** menu, and choose **Components And Controls** from the submenu. Select **Registered ActiveX Controls**, and then choose Microsoft Web Browser. Visual C++ will generate the wrapper class `CWebBrowser2` and add the files to your project.

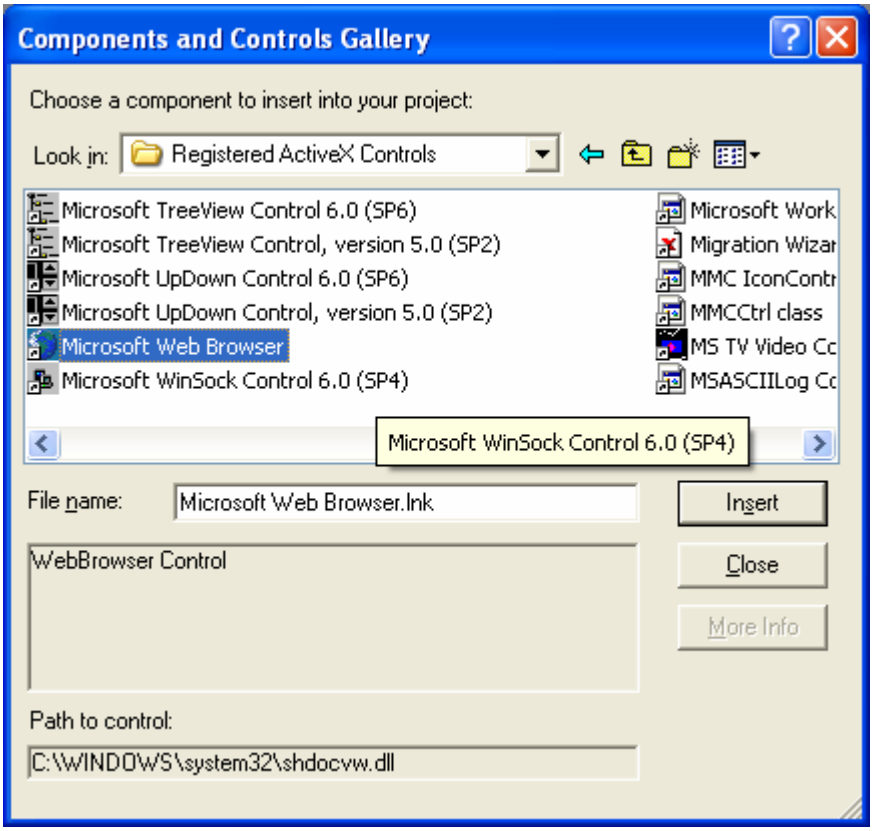

Figure 24: Installing the Web Browser control in the MYMFC24B project.

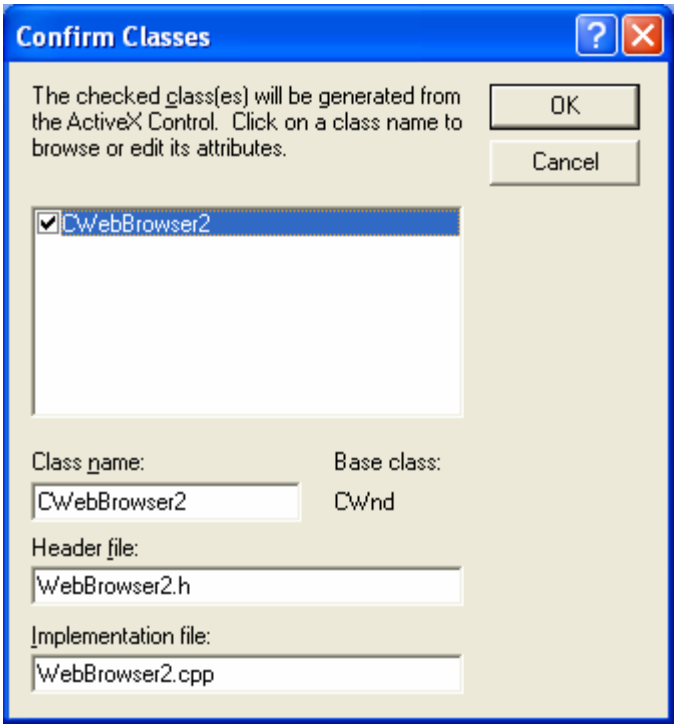

Figure 25: Generated class, `CWebBrowser2` and their files that will be added to the project.

Add two `CWebBrowser2` data members to the `CMymfc24BView` class. Click on the **ClassView** tab in the Workspace window, and then right-click the `CMymfc24BView` class. Choose **Add Member Variable**, and fill in the dialog as shown here.

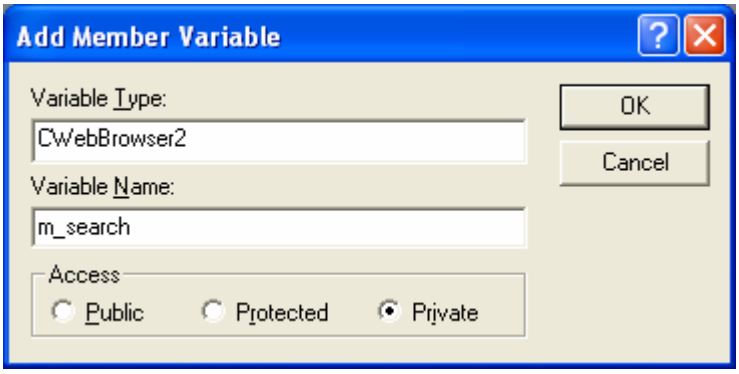

Figure 26: Adding data members/member variables with `CWebBrowser2` type.

Repeat for `m_target`. ClassWizard adds an `#include` statement for the **webbrowser2.h** file.

```
#include "webbrowser2.h"     // Added by ClassView
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
```

Listing 8.

Add the child window ID constants for the two controls. Select **Resource Symbols** from Visual C++'s **View** menu, and then add the symbols `ID_BROWSER_SEARCH` and `ID_BROWSER_TARGET`.

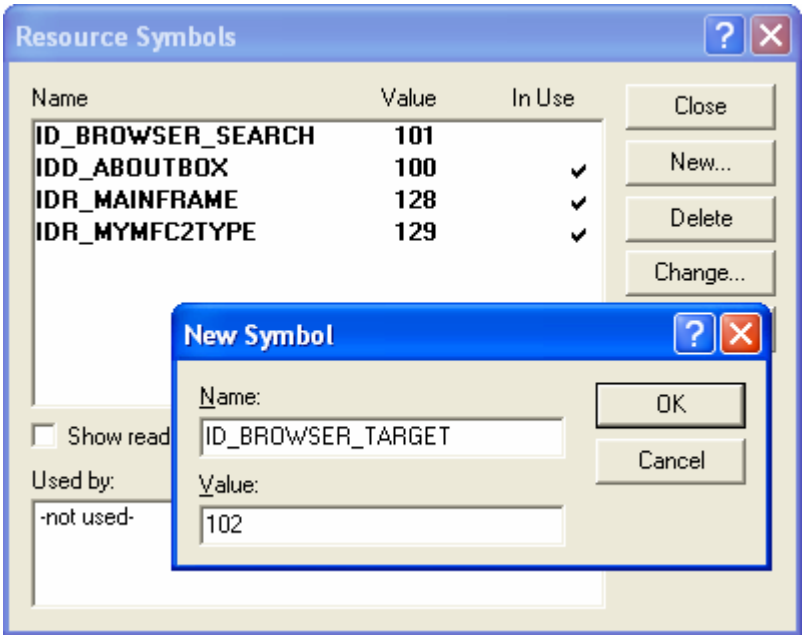

Figure 27: Adding child window ID constants for the previous two controls (browsers).

Add a static character array data member for the Yahoo URL. Add the following static data member to the class declaration in **mymfc24BView.h**:

```
private:
```

```
        static const char s_engineYahoo[];
```

```
    CWebBrowser2 m_target;
    CWebBrowser2 m_search;
private:
    static const char s_engineYahoo[];
};
```

Listing 9.

Then add the following definition in **mymfc24BView.cpp**, outside any function:

```
    const char CMymfc24BView::s_engineYahoo[] = "http://www.yahoo.com/";
```

```
END_MESSAGE_MAP()

const char CMymfc24BView::s_engineYahoo[] = "http://www.yahoo.com/";

/////////////////////////////////////////////////////////////////////////////
// CMymfc24BView construction/destruction
```

Listing 10.

Use ClassWizard to map the view's `WM_CREATE` and `WM_SIZE` messages. Edit the handler code in **mymfc24BView.cpp** as follows:

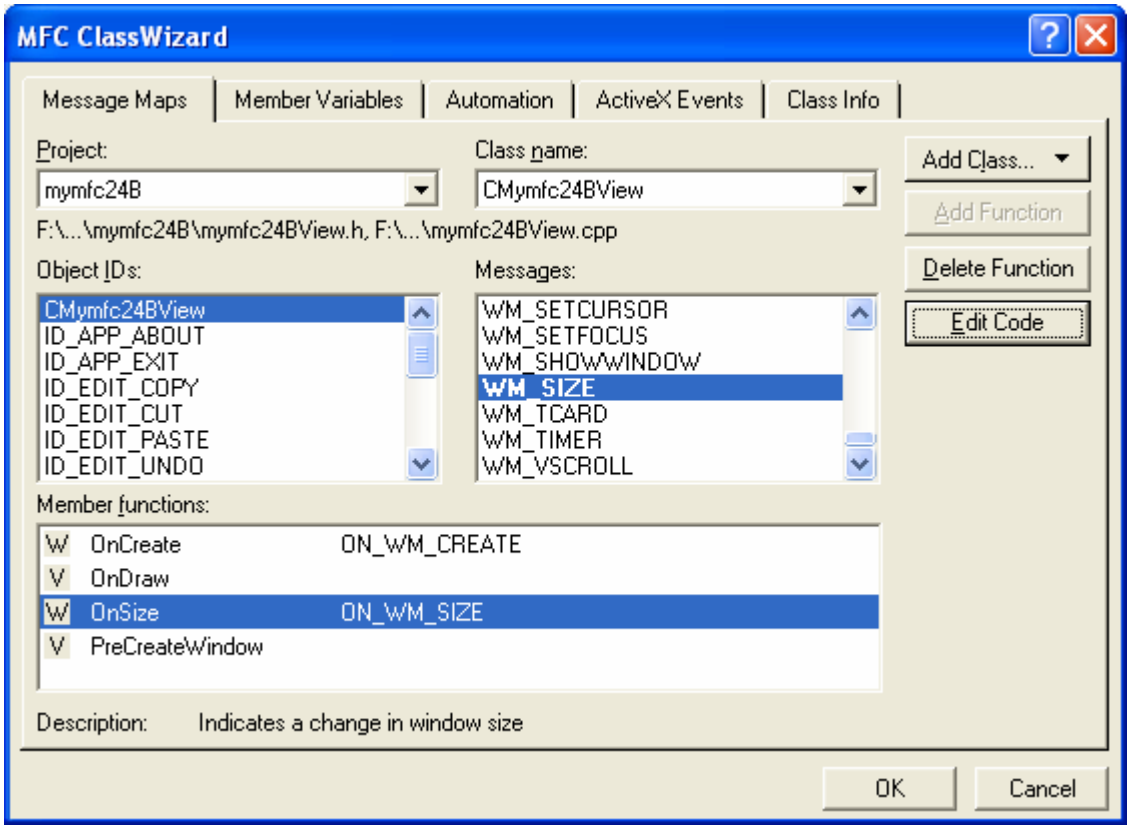

Figure 28: Using ClassWizard to map the view's `WM_CREATE` and `WM_SIZE` messages.

```
int CMymfc24BView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;
```

```
        DWORD dwStyle = WS_VISIBLE | WS_CHILD;
        if (m_search.Create(NULL, dwStyle, CRect(0, 0, 100, 100), this,
    ID_BROWSER_SEARCH) == 0) {
            AfxMessageBox("Unable to create search control!\n");
            return -1;
        }
        m_search.Navigate(s_engineYahoo, NULL, NULL, NULL, NULL);

        if (m_target.Create(NULL, dwStyle, CRect(0, 0, 100, 100), this,
    ID_BROWSER_TARGET) == 0) {
            AfxMessageBox("Unable to create target control!\n");
            return -1;
        }
        m_target.GoHome(); // as defined in Internet Explorer 4 options

        return 0;
    }
```

```
int CMymfc24BView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;

    DWORD dwStyle = WS_VISIBLE | WS_CHILD;
    if (m_search.Create(NULL, dwStyle, CRect(0, 0, 100, 100),
                        this, ID_BROWSER_SEARCH) == 0) {
        AfxMessageBox("Unable to create search control!\n");
        return -1;
    }
    m_search.Navigate(s_engineYahoo, NULL, NULL, NULL, NULL);

    if (m_target.Create(NULL, dwStyle, CRect(0, 0, 100, 100),
                        this, ID_BROWSER_TARGET) == 0) {
        AfxMessageBox("Unable to create target control!\n");
        return -1;
    }
    m_target.GoHome(); // as defined in Internet Explorer 4 options

    return 0;
}
```

Listing 11.

```
    void CMymfc24BView::OnSize(UINT nType, int cx, int cy)
    {
        CView::OnSize(nType, cx, cy);

        CRect rectClient;
        GetClientRect(rectClient);
        CRect rectBrowse(rectClient);
        rectBrowse.right = rectClient.right / 2;
        CRect rectSearch(rectClient);
        rectSearch.left = rectClient.right / 2;

        m_target.SetWidth(rectBrowse.right - rectBrowse.left);
        m_target.SetHeight(rectBrowse.bottom - rectBrowse.top);
        m_target.UpdateWindow();

        m_search.SetLeft(rectSearch.left);
        m_search.SetWidth(rectSearch.right - rectSearch.left);
        m_search.SetHeight(rectSearch.bottom - rectSearch.top);
        m_search.UpdateWindow();
    }
```

```
void CMymfc24BView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);

    CRect rectClient;
    GetClientRect(rectClient);
    CRect rectBrowse(rectClient);
    rectBrowse.right = rectClient.right / 2;
    CRect rectSearch(rectClient);
    rectSearch.left = rectClient.right / 2;

    m_target.SetWidth(rectBrowse.right - rectBrowse.left);
    m_target.SetHeight(rectBrowse.bottom - rectBrowse.top);
    m_target.UpdateWindow();

    m_search.SetLeft(rectSearch.left);
    m_search.SetWidth(rectSearch.right - rectSearch.left);
    m_search.SetHeight(rectSearch.bottom - rectSearch.top);
    m_search.UpdateWindow();
}
```

Listing 12.

The `OnCreate()` function creates two browser windows inside the view window. The right-hand browser displays the top-level Yahoo page, and the left-hand browser displays the "home" page as defined through the Internet icon in the Control Panel. The `OnSize()` function, called whenever the view window changes size, ensures that the browser windows completely cover the view window. The `CWebBrowser2` member functions `SetWidth()` and `SetHeight()` set the browser's **Width** and **Height** properties.

Add the **event sink macros** in the `CMymfc24BView` files. ClassWizard can't map events from a dynamic ActiveX control, so you must do it manually. Add the following lines inside the class declaration in the file **mymfc24BView.h**:

```
    protected:
        afx_msg void OnBeforeNavigateExplorer1(LPCTSTR URL, long Flags, LPCTSTR
    TargetFrameName, VARIANT FAR* PostData, LPCTSTR Headers, BOOL FAR* Cancel);
        afx_msg void OnTitleChangeExplorer2(LPCTSTR Text);
        DECLARE_EVENTSINK_MAP()
```

```
    DECLARE_MESSAGE_MAP()

protected:
    afx_msg void OnBeforeNavigateExplorer1(LPCTSTR URL, long Fla
    afx_msg void OnTitleChangeExplorer2(LPCTSTR Text);
    DECLARE_EVENTSINK_MAP()

private:
    CWebBrowser2 m_target;
```

Listing 13.

Then add the following code in **mymfc24BView.cpp**:

```
    BEGIN_EVENTSINK_MAP(CMymfc24BView, CView)
        ON_EVENT(CMymfc24BView, ID_BROWSER_SEARCH, 100, OnBeforeNavigateExplorer1,
    VTS_BSTR VTS_I4 VTS_BSTR VTS_PVARIANT VTS_BSTR VTS_PBOOL)
        ON_EVENT(CMymfc24BView, ID_BROWSER_TARGET, 113,  OnTitleChangeExplorer2,
    VTS_BSTR)
    END_EVENTSINK_MAP()
```

```
END_MESSAGE_MAP()

BEGIN_EVENTSINK_MAP(CMymfc24BView, CView)
    ON_EVENT(CMymfc24BView, ID_BROWSER_SEARCH, 100, OnBeforeNa
    ON_EVENT(CMymfc24BView, ID_BROWSER_TARGET, 113,  OnTitleCh
END_EVENTSINK_MAP()

const char CMymfc24BView::s_engineYahoo[] = "http://www.yahoo.

//////////////////////////////////////////////////////////////////
```

Listing 14.

Add two event handler functions. Add the following member functions in **mymfc24BView.cpp**:

```
void CMymfc24BView::OnBeforeNavigateExplorer1(LPCTSTR URL, long Flags, LPCTSTR
TargetFrameName,
    VARIANT FAR* PostData, LPCTSTR Headers, BOOL FAR* Cancel)
{
    TRACE("CMymfc24BView::OnBeforeNavigateExplorer1 -- URL = %s\n", URL);

    if (!strnicmp(URL, s_engineYahoo, strlen(s_engineYahoo))) {
        return;
    }
    m_target.Navigate(URL, NULL, NULL, PostData, NULL);
    *Cancel = TRUE;
}
```

```
void CMymfc24BView::OnBeforeNavigateExplorer1(LPCTSTR URL,
    long Flags, LPCTSTR TargetFrameName,
    VARIANT FAR* PostData, LPCTSTR Headers, BOOL FAR* Cancel)
{
    TRACE("CMymfc24BView::OnBeforeNavigateExplorer1 -- URL = %s\n", URL);

    if (!strnicmp(URL, s_engineYahoo, strlen(s_engineYahoo))) {
        return;
    }
    m_target.Navigate(URL, NULL, NULL, PostData, NULL);
    *Cancel = TRUE;
}
```

Listing 15.

```
void CMymfc24BView::OnTitleChangeExplorer2(LPCTSTR Text)
{
    // Careful!  Event could fire before we're ready.
    CWnd* pWnd = AfxGetApp()->m_pMainWnd;
    if (pWnd != NULL) {
        if (::IsWindow(pWnd->m_hWnd)) {
            pWnd->SetWindowText(Text);
        }
    }
}
```

```
void CMymfc24BView::OnTitleChangeExplorer2(LPCTSTR Text)
{
    // Careful!  Event could fire before we're ready.
    CWnd* pWnd = AfxGetApp()->m_pMainWnd;
    if (pWnd != NULL) {
        if (::IsWindow(pWnd->m_hWnd)) {
            pWnd->SetWindowText(Text);
        }
    }
}
```

Listing 16.

The `OnBeforeNavigateExplorer1()` handler is called when the user clicks on a link in the search page. The function compares the clicked URL (in the **URL string** parameter) with the search engine URL. If they match, the navigation proceeds in the search window; otherwise, the navigation is cancelled and the Navigate method is called for the target window. The `OnTitleChangeExplorer2()` handler updates the MYMFC24B window title to match the title on the target page.

Build and test the MYMFC24B application. Search for something on the Yahoo page, and then watch the information appear in the target page.

Note: This program may not work as specified or both windows may function as a separate browser for newer version of IE.

Figure 29: MYMFC24B program output.

Picture Properties

Some ActiveX controls support **picture** properties, which can accommodate bitmaps, metafiles, and icons. If an ActiveX control has at least one picture property, ClassWizard generates a `CPicture` class in your project during the control's installation. You don't need to use this `CPicture` class, but you must use the MFC class `CPictureHolder`. To access the `CPictureHolder` class declaration and code, you need the following line in **StdAfx.h**:

```
#include <afxctl.h>
```

Suppose you have an ActiveX control with a picture property named **Picture**. Here's how you set the **Picture** property to a bitmap in your program's resources:

```
CPictureHolder pict;
pict.CreateFromBitmap(IDB_MYBITMAP); // from project's resources
m_control.SetPicture(pict.GetPictureDispatch());
```

If you include the **AfxCtl.h** file, you can't statically link your program with the MFC library. If you need a stand-alone program that supports picture properties, you'll have to borrow code from the `CPictureHolder` class, located in the **\Program Files\Microsoft Visual Studio\VC98\mfc\src\ctlpict.cpp** file.

Bindable Properties: Change Notifications

If an ActiveX control has a property designated as bindable, the control will send an `OnChanged()` notification to its container when the value of the property changes inside the control. In addition, the control can send an `OnRequestEdit()` notification for a property whose value is about to change but has not yet changed. If the container returns `FALSE` from its `OnRequestEdit()` handler, the control should not change the property value. MFC fully supports property change notifications in ActiveX control containers, but as of Visual C++ version 6.0, no ClassWizard support was available. That means you must manually add entries to your container class's event sink map. Suppose you have an ActiveX control with a bindable property named **Note** with a dispatch ID of 4. You add an `ON_PROPNOTIFY` macro to the `EVENTSINK` macros in this way:

```
BEGIN_EVENTSINK_MAP(CAboutDlg, CDialog)
    //{{AFX_EVENTSINK_MAP(CAboutDlg)
    // ClassWizard places other event notification macros here
    //}}AFX_EVENTSINK_MAP
    ON_PROPNOTIFY(CAboutDlg, IDC_MYCTRL1, 4, OnNoteRequestEdit, OnNoteChanged)
END_EVENTSINK_MAP()
```

You must then code the `OnNoteRequestEdit()` and `OnNoteChanged()` functions with return types and parameter types exactly as shown here:

```
BOOL CMyDlg::OnNoteRequestEdit(BOOL* pb)
{
    TRACE("CMyDlg::OnNoteRequestEdit\n");
    *pb = TRUE; // TRUE means change request granted
    return TRUE;
}

BOOL CMyDlg::OnNoteChanged()
{
    TRACE("CMyDlg::OnNoteChanged\n");
    return TRUE;
}
```

You'll also need corresponding prototypes in the class header, as shown here:

```
afx_msg BOOL OnNoteRequestEdit(BOOL* pb);
afx_msg BOOL OnNoteChanged();
```

Other ActiveX Controls

You'll probably notice that your disk fills up with ActiveX controls, especially if you accept controls from Web sites. Most of these controls are difficult to use unless you have the documentation on hand, but you can have fun experimenting. Try the Marquee.ocx control that is distributed with Visual C++ 6.0. It works fine in both MFC programs and HTML files. The trick is to set the **szURL** property to the name of another HTML file that contains the text to display in the scrolling marquee window.
Many ActiveX controls were designed for use by Visual Basic programmers. The **SysInfo.ocx** control that comes with Visual C++, for example, lets you retrieve system parameters as property values. This isn't of much use to a C++ programmer, however, because you can make the equivalent Win32 calls anytime. Unlike the many objects provided by MFC, ActiveX controls are binary objects that are not extensible. For example, you cannot add a property or event to an ActiveX control. Nor can you use many C++ object-oriented techniques like polymorphism with ActiveX controls. Another downside of ActiveX controls is they are not compatible with many advanced MFC concepts such as the document/view architecture, which we will cover later.

Further reading and digging:

1. MSDN MFC 6.0 class library online documentation - used throughout this Tutorial.
2. MSDN MFC 7.0 class library online documentation - used in .Net framework and also backward compatible with 6.0 class library
3. MSDN Library
4. Windows data type.
5. Win32 programming Tutorial.
6. The best of C/C++, MFC, Windows and other related books.
7. Unicode and Multibyte character set: Story and program examples.