Module 21: Bitmaps

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:

Bitmaps

Without graphics images, Microsoft Windows-based applications would be pretty dull. Some applications depend on images for their usefulness, but any application can be spruced up with the addition of decorative clip art from a variety of sources. Windows bitmaps are arrays of bits mapped to display pixels. That might sound simple, but you have to learn a lot about bitmaps before you can use them to create professional applications for Windows.
This module starts with the "old" way of programming bitmaps, creating the device-dependent GDI bitmaps that work with a memory device context. You need to know these techniques because many programmers are still using them and you'll also need to use them on occasion.
Next you'll graduate to the modern way of programming bitmaps, creating **device-independent bitmaps** (DIBs). If you use DIBs, you'll have an easier time with colors and with the printer. In some cases you'll get better performance. The Win32 function `CreateDIBSection()` gives you the benefits of DIBs combined with all the features of GDI bitmaps. Finally, you'll learn how to use the MFC `CBitmapButton` class to put bitmaps on pushbuttons. Using `CBitmapButton` to put bitmaps on pushbuttons has nothing to do with DIBs, but it's a useful technique that would be difficult to master without an example.

GDI Bitmaps and Device-Independent Bitmaps

There are two kinds of Windows bitmaps: **GDI bitmaps** and **DIBs**. GDI bitmap objects are represented by the MFC Library version 6.0 `CBitmap` class. The GDI bitmap object has an associated Windows data structure, maintained inside the Windows GDI module, which is **device-dependent**. Your program can get a copy of the bitmap data, but the bit arrangement depends on the **display hardware**. GDI bitmaps can be freely transferred among programs on a single computer, but because of their device dependency, transferring bitmaps by disk or modem doesn't make sense.

In Win32, you're allowed to put a GDI bitmap handle on the clipboard for transfer to another process, but behind the scenes Windows **converts the device-dependent bitmap to a DIB** and copies the DIB to shared memory. That's a good reason to consider using DIBs from the start.
DIBs offer many programming advantages over GDI bitmaps. Because a DIB carries its own color information, color palette management is easier. DIBs also make it easy to control gray shades when printing. Any computer running Windows can process DIBs, which are usually **stored in BMP disk files** or as a resource in your program's EXE or DLL file. The wallpaper background on your monitor is read from a BMP file when you start Windows. The primary storage format for Microsoft Paint is the BMP file, and Visual C++ uses BMP files for toolbar buttons and other images. Other graphic interchange formats are available, such as TIFF, GIF, PNG and JPEG, but **only the DIB format is directly supported by the Win32 API**.

Color Bitmaps and Monochrome Bitmaps

Now might be a good time to reread the "Windows Color Mapping" section in Module 4. As you'll see in this module, Windows deals with color bitmaps a little differently from the way it deals with brush colors. Many color bitmaps are 16-color. A standard VGA board has **four contiguous color planes**, with 1 corresponding bit from each plane combining to represent a pixel. The 4-bit color values are set when the bitmap is created. With a standard VGA board, bitmap colors are limited to the standard 16 colors. Windows does not use dithered colors in bitmaps. A monochrome bitmap has only **one plane**. Each pixel is represented by a single bit that is either off (0) or on (1). The `CDC::SetTextColor` function sets the "off" display color, and `SetBkColor()` sets the "on" color. You can specify these pure colors individually with the Windows RGB macro.

Using GDI Bitmaps

A GDI bitmap is simply another GDI object, such as a pen or a font. You must somehow create a bitmap, and then you must select it into a device context. When you're finished with the object, you must deselect it and delete it. You know the drill.
There's a catch, though, because the "bitmap" of the display or printer device is effectively the display surface or the printed page itself. Therefore, you can't select a bitmap into a display device context or a printer device context. You have to create a special memory device context for your bitmaps, using the `CDC::CreateCompatibleDC` function. You must then use the `CDC` member function `StretchBlt()` or `BitBlt()` to copy the bits from the memory device context to the "real" device context. These "bit-blitting" functions are generally called in your view class's `OnDraw()` function. Of course, you mustn't forget to clean up the memory device context when you're finished.

Loading a GDI Bitmap from a Resource

The easiest way to use a bitmap is to load it from a resource. If you look in **ResourceView** in the Workspace window, you'll find a list of the project's bitmap resources. If you select a bitmap and examine its properties, you'll see a filename as shown below.

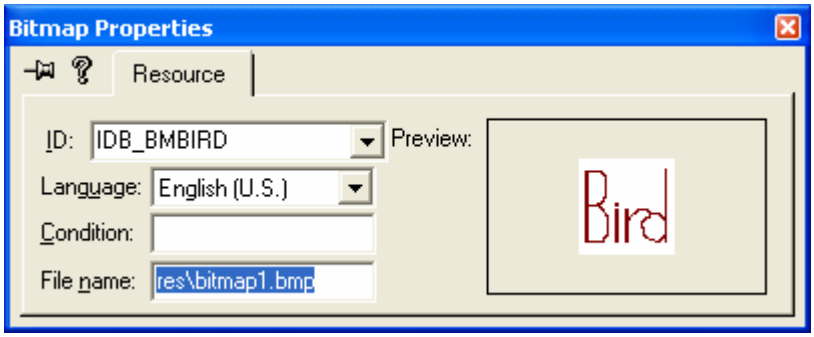

Figure 1: Bitmap properties dialog.

Here's an example entry in an RC (resource script) file, when viewed by a text editor:

```
...
// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
```

```
IDR_MAINFRAME           ICON    DISCARDABLE     "res\\mymfc25A.ico"
IDR_MYMFC2TYPE          ICON    DISCARDABLE     "res\\mymfc25ADoc.ico"
IDI_BLACK               ICON    DISCARDABLE     "res\\icon1.ico"
IDI_BLUE                ICON    DISCARDABLE     "res\\ico00001.ico"
IDI_CYAN                ICON    DISCARDABLE     "res\\ico00002.ico"
...

/////////////////////////////////////////////////////////////////////////////
//
// Bitmap
//

IDR_MAINFRAME           BITMAP  MOVEABLE PURE   "res\\Toolbar.bmp"
IDB_BMBIRD              BITMAP  DISCARDABLE     "res\\bitmap1.bmp"
IDB_BMBIRDSELECTED      BITMAP  DISCARDABLE     "res\\bmp00001.bmp"
IDB_BMDOG               BITMAP  DISCARDABLE     "res\\bmdog.bmp"
IDB_BMDOGSELECTED       BITMAP  DISCARDABLE     "res\\bmp00002.bmp"
...
```

`IDB_BMBIRD` is the resource ID, and the file is **bitmap1.bmp** in the project's **\res** subdirectory. The resource compiler reads the DIB from disk and stores it in the project's RES file. The linker copies the DIB into the program's EXE file. You know that the **bitmap1** bitmap must be in device-independent format because the EXE can be run with any display board that Windows supports.
The `CDC::LoadBitmap` function converts a resource-based DIB to a GDI bitmap. Below is the simplest possible self-contained `OnDraw()` function that displays the **bitmap1** bitmap:

```
CMyView::OnDraw(CDC* pDC)
{
    CBitmap bitmap; // Sequence is important
    CDC dcMemory;
    bitmap.LoadBitmap(IDB_BMBIRD);
    dcMemory.CreateCompatibleDC(pDC);
    dcMemory.SelectObject(&bitmap);
    pDC->BitBlt(100, 100, 54, 96, &dcMemory, 0, 0, SRCCOPY);
    // CDC destructor deletes dcMemory; bitmap is deselected
    // CBitmap destructor deletes bitmap
}
```

The `BitBlt()` function copies the **bitmap1** pixels from the memory device context to the display (or printer) device context. The bitmap is 54 bits wide by 96 bits high, and on a VGA display it occupies a rectangle of 54-by-96 logical units, offset 100 units down and to the right of the upper-left corner of the window's client area. The code above works fine for the display. The application framework calls the `OnDraw()` function for printing, in which case `pDC` points to a printer device context. The bitmap here, unfortunately, is configured specifically for the display and thus cannot be selected into the printer-compatible memory device context. If you want to print a bitmap, you should look at the `CDib` class described later in this module.

The Effect of the Display Mapping Mode

If the display mapping mode in the `bitmap1` example is `MM_TEXT`, each bitmap pixel maps to a display pixel and the bitmap fits perfectly. If the mapping mode is `MM_LOENGLISH`, the bitmap size is 0.54-by-0.96 inch, or 52-by-92 pixels for Windows 95 and the GDI must do some bit crunching to make the bitmap fit. Consequently, the bitmap might not look as good with the `MM_LOENGLISH` mapping mode. Calling `CDC::SetStretchBltMode` with a parameter value of `COLORONCOLOR` will make shrunken bitmaps look nicer.

Stretching the Bits

What if we want **bitmap1** to occupy a rectangle of exactly 54-by-96 pixels, even though the mapping mode is not `MM_TEXT`? The `StretchBlt()` function is the solution. If we replace the `BitBlt()` call with the following three statements, **bitmap1** is displayed cleanly, whatever the mapping mode:

```
CSize size(54, 96);
```

```
pDC->DPtoLP(&size);
pDC->StretchBlt(0, 0, size.cx, -size.cy, &dcMemory, 0, 0, 54, 96, SRCCOPY);
```

With either `BitBlt()` or `StretchBlt()`, the display update is slow if the GDI has to actually stretch or compress bits. If, as in the case above, the GDI determines that no conversion is necessary, the update is fast.

The MYMFC26A Example

The MYMFC26A example displays a resource-based bitmap in a scrolling view with mapping mode set to `MM_LOENGLISH`. The program uses the `StretchBlt()` logic described above, except that the memory device context and the bitmap are created in the view's `OnInitialUpdate()` member function and last for the life of the program. Also, the program reads the bitmap size through a call to the `CGdiObject` member function `GetObject()`, so it's not using hard-coded values as in the preceding examples.

Here are the steps for building the example:

Run AppWizard to produce \mfcproject\mymfc26A. Accept all the default settings but two: select **Single Document**, and in step 6, select the `CScrollView` view base class for `CMymfc26AView`. The options and the default class names are shown here.

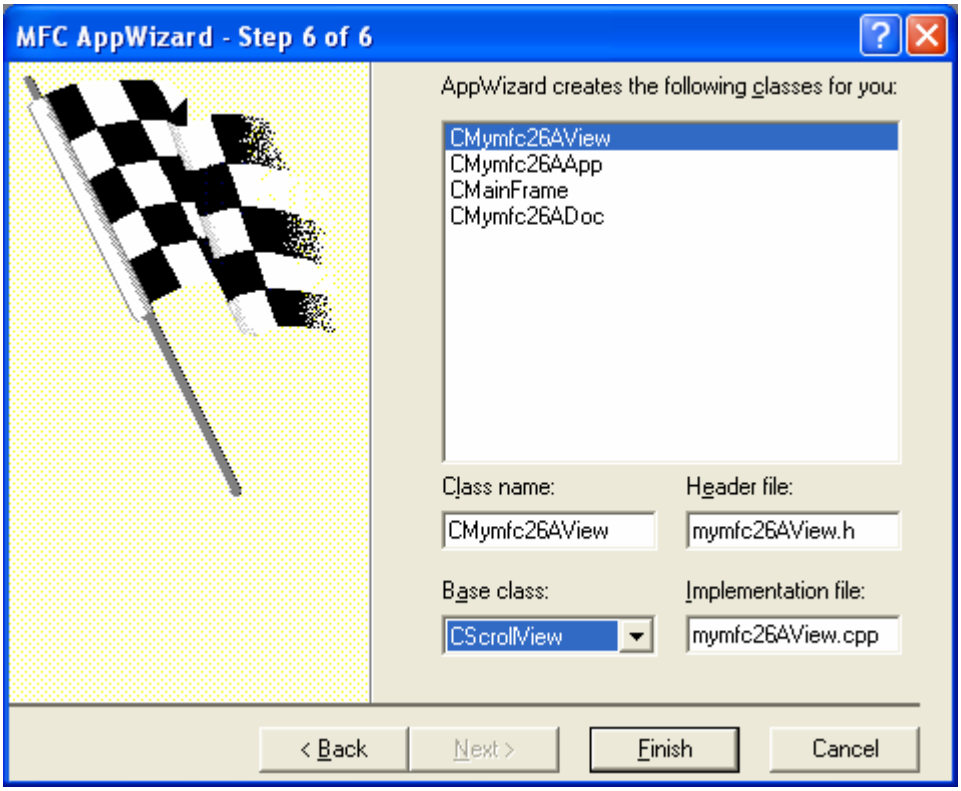

Figure 1: MYMFC26A step 6 of 6 AppWizard, using `CScrollView` as view base class.

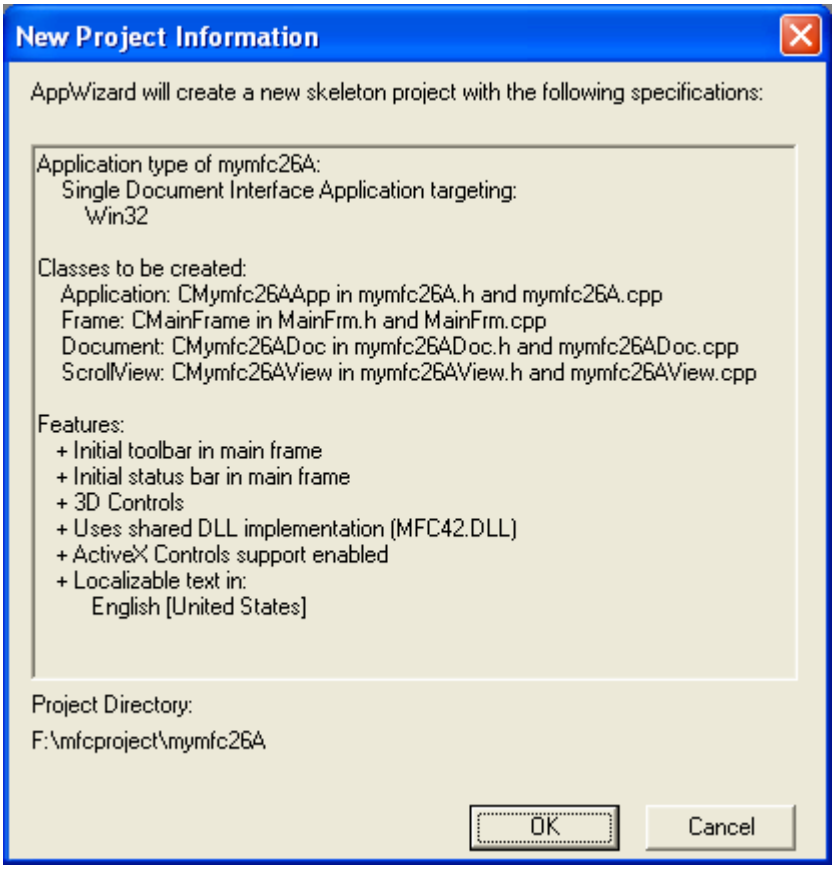

Figure 2: MYMFC26A project summary.

Import the **Soap Bubbles.bmp** bitmap. Choose **Resource** from Visual C++'s Insert menu. Import the bitmap **Soap Bubbles.bmp** from the **\WINDOWS** directory (or other bmp file from the **\WINDOWS** directory or your own bmp).

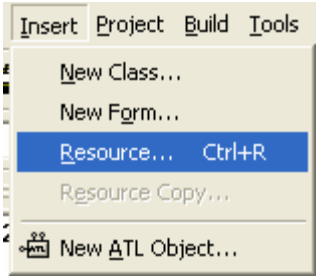

Figure 3: Inserting a new resource.

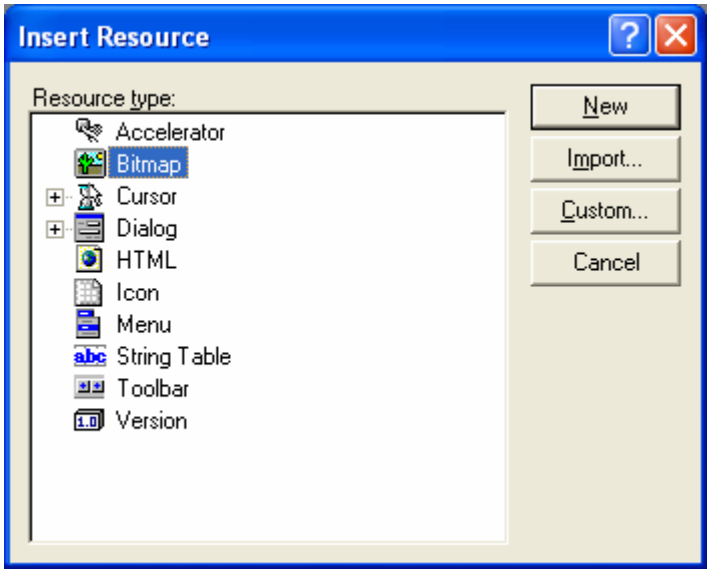

Figure 4: Importing a new bitmap into the project.

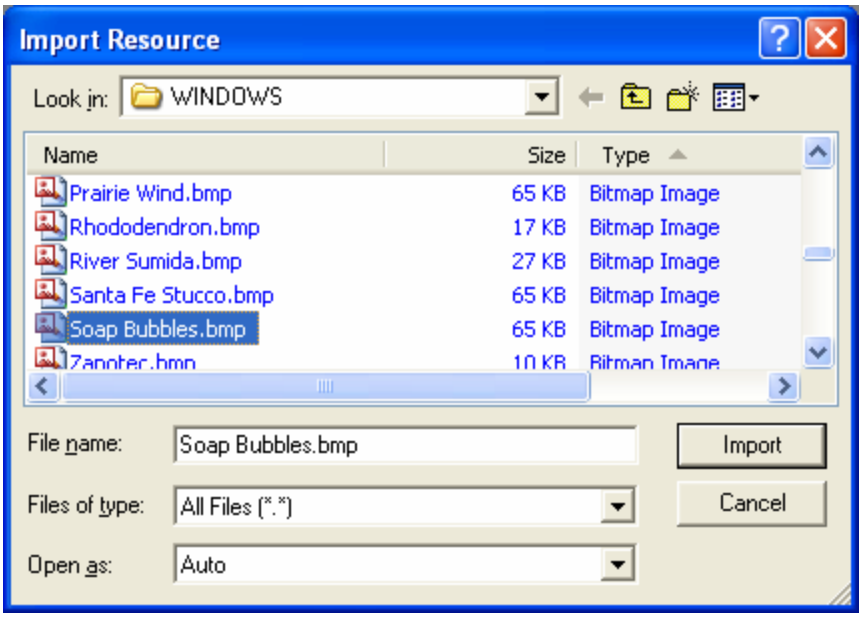

Figure 5: Selecting **Soap Bubbles.bmp** bitmap file from Windows system directory.

Visual C++ will copy this bitmap file into your project's **\res** subdirectory. Assign the ID `IDB_SOAPBUBBLE`, and save the changes.

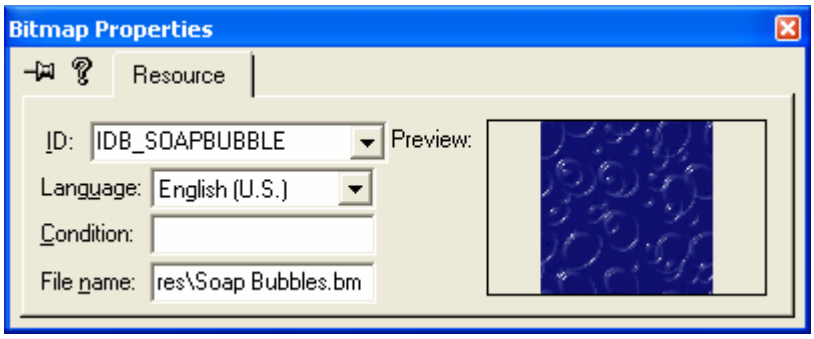

Figure 6: The imported bitmap properties dialog copied into Visual C++ **\res** directory.

Add the following private data members to the class `CMymfc26AView`. Edit the file **mymfc26AView.h** or use ClassView. The bitmap and the memory device context last for the life of the view. The `CSize` objects are the source (bitmap) dimensions and the destination (display) dimensions.

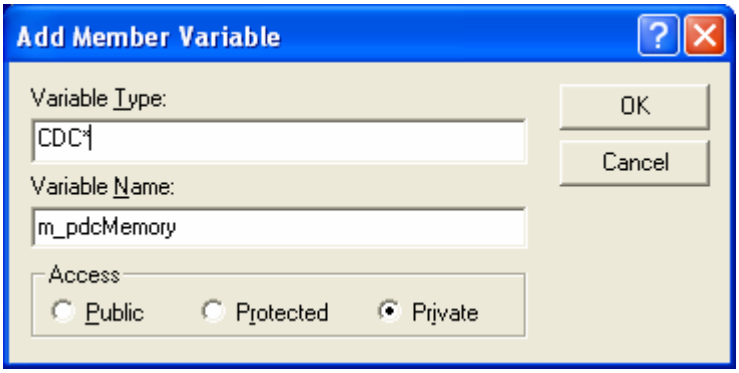

Figure 7: Using ClassView to add a private data members to the class `CMymfc26AView`..

```
    CDC*     m_pdcMemory;
    CBitmap* m_pBitmap;
    CSize    m_sizeSource, m_sizeDest;
```

```
    DECLARE_MESSAGE_MAP()
private:
    CSize m_sizeDest;
    CSize m_sizeSource;
    CBitmap* m_pBitmap;
    CDC* m_pdcMemory;
};
```

Listing 1.

Edit the following member functions in the class `CMymfc26AView`. Edit the file **mymfc26AView.cpp**. The constructor and destructor do C++ housekeeping for the embedded objects. You want to keep the constructor as simple as possible because failing constructors cause problems. The `OnInitialUpdate()` function sets up the memory device context and the bitmap, and it computes output dimensions that map each bit to a pixel. The `OnDraw()` function calls `StretchBlt()` twice, once by using the special computed dimensions and once by mapping each bit to a 0.01-by-0.01-inch square. Add the following code:

```
CMymfc26AView::CMymfc26AView()
{
    m_pdcMemory = new CDC;
    m_pBitmap = new CBitmap;
}
```

```
// CMymfc26AView construction/destruction

CMymfc26AView::CMymfc26AView()
{
    // TODO: add construction code here
    m_pdcMemory = new CDC;
    m_pBitmap = new CBitmap;
}
```

Listing 2.

```
CMymfc26AView::~CMymfc26AView()
```

```
    {
        // cleans up the memory device context and the bitmap
        delete m_pdcMemory; // deselects bitmap
        delete m_pBitmap;
    }
```

```
CMymfc26AView::~CMymfc26AView()
{
    // cleans up the memory device context and the bitmap
    delete m_pdcMemory; // deselects bitmap
    delete m_pBitmap;
}
```

Listing 3.

```
    void CMymfc26AView::OnDraw(CDC* pDC)
    {
        pDC->SetStretchBltMode(COLORONCOLOR);
        pDC->StretchBlt(20, -20, m_sizeDest.cx, -m_sizeDest.cy, m_pdcMemory, 0, 0,
            m_sizeSource.cx, m_sizeSource.cy, SRCCOPY);

        pDC->StretchBlt(350, -20, m_sizeSource.cx, -m_sizeSource.cy, m_pdcMemory, 0, 0,
            m_sizeSource.cx, m_sizeSource.cy, SRCCOPY);
    }
```

```
// CMymfc26AView drawing

void CMymfc26AView::OnDraw(CDC* pDC)
{
    pDC->SetStretchBltMode(COLORONCOLOR);
    pDC->StretchBlt(20, -20, m_sizeDest.cx, -m_sizeDest.cy,
        m_pdcMemory, 0, 0,
        m_sizeSource.cx, m_sizeSource.cy, SRCCOPY);

    pDC->StretchBlt(350, -20, m_sizeSource.cx, -m_sizeSource.cy,
        m_pdcMemory, 0, 0,
        m_sizeSource.cx, m_sizeSource.cy, SRCCOPY);
}
```

Listing 4.

```
    void CMymfc26AView::OnInitialUpdate()
    {
        CScrollView::OnInitialUpdate();
        CSize sizeTotal(800, 1050); // 8-by-10.5 inches
        CSize sizeLine = CSize(sizeTotal.cx / 100, sizeTotal.cy / 100);
        SetScrollSizes(MM_LOENGLISH, sizeTotal, sizeTotal, sizeLine);

        BITMAP bm; // Windows BITMAP data structure; see Win32 help
        if (m_pdcMemory->GetSafeHdc() == NULL) {
            CClientDC dc(this);
            OnPrepareDC(&dc); // necessary
            //Change the ID to your own bitmap if any
            m_pBitmap->LoadBitmap(IDB_SOAPBUBBLE);
            m_pdcMemory->CreateCompatibleDC(&dc);
            m_pdcMemory->SelectObject(m_pBitmap);
            m_pBitmap->GetObject(sizeof(bm), &bm);
            m_sizeSource.cx = bm.bmWidth;
            m_sizeSource.cy = bm.bmHeight;
            m_sizeDest = m_sizeSource;
            dc.DPtoLP(&m_sizeDest);
        }
    }
```

```
void CMymfc26AView::OnInitialUpdate()
{
    CScrollView::OnInitialUpdate();
    CSize sizeTotal(800, 1050); // 8-by-10.5 inches
    CSize sizeLine = CSize(sizeTotal.cx / 100, sizeTotal.cy / 100);
    SetScrollSizes(MM_LOENGLISH, sizeTotal, sizeTotal, sizeLine);

    BITMAP bm; // Windows BITMAP data structure; see Win32 help
    if (m_pdcMemory->GetSafeHdc() == NULL) {
        CClientDC dc(this);
        OnPrepareDC(&dc); // necessary
        //Change the ID to your own bitmap if any
        m_pBitmap->LoadBitmap(IDB_SOAPBUBBLE);
        m_pdcMemory->CreateCompatibleDC(&dc);
        m_pdcMemory->SelectObject(m_pBitmap);
        m_pBitmap->GetObject(sizeof(bm), &bm);
        m_sizeSource.cx = bm.bmWidth;
        m_sizeSource.cy = bm.bmHeight;
        m_sizeDest = m_sizeSource;
        dc.DPtoLP(&m_sizeDest);
    }
}
```

Listing 5.

Build and test the MYMFC26A application. Your screen should look something like this.

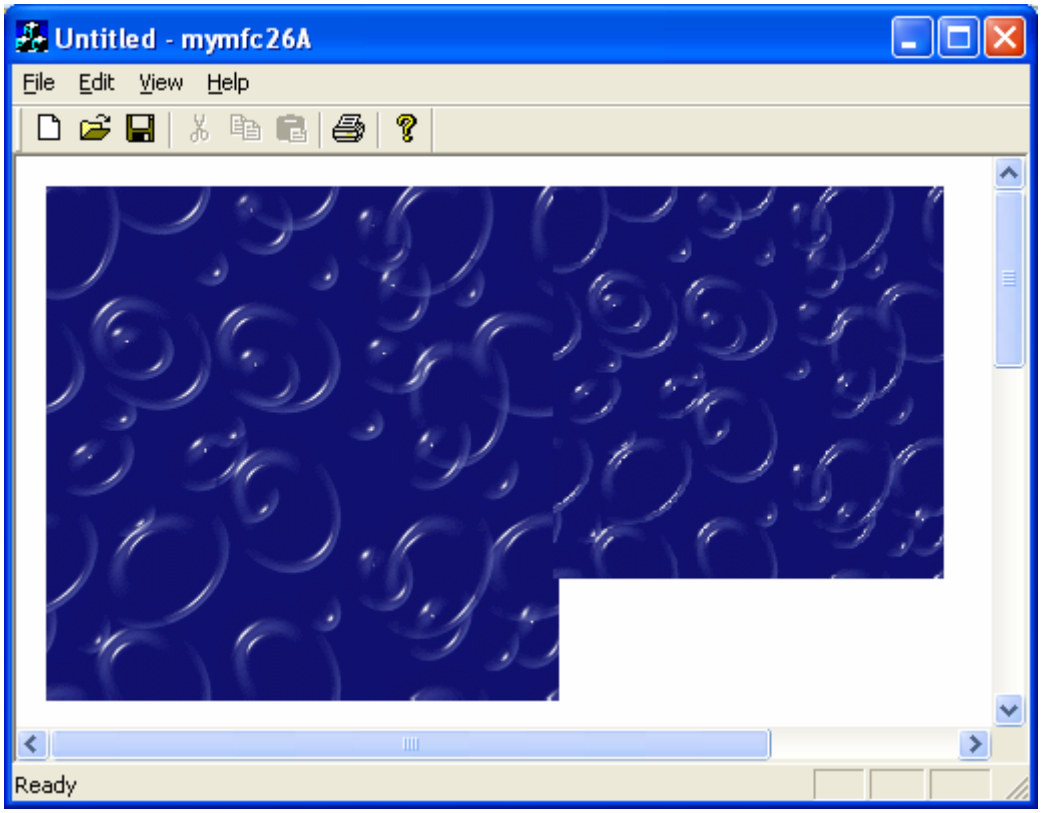

Figure 8: MYMFC26A program output.

Try the **Print Preview** and **Print** features.

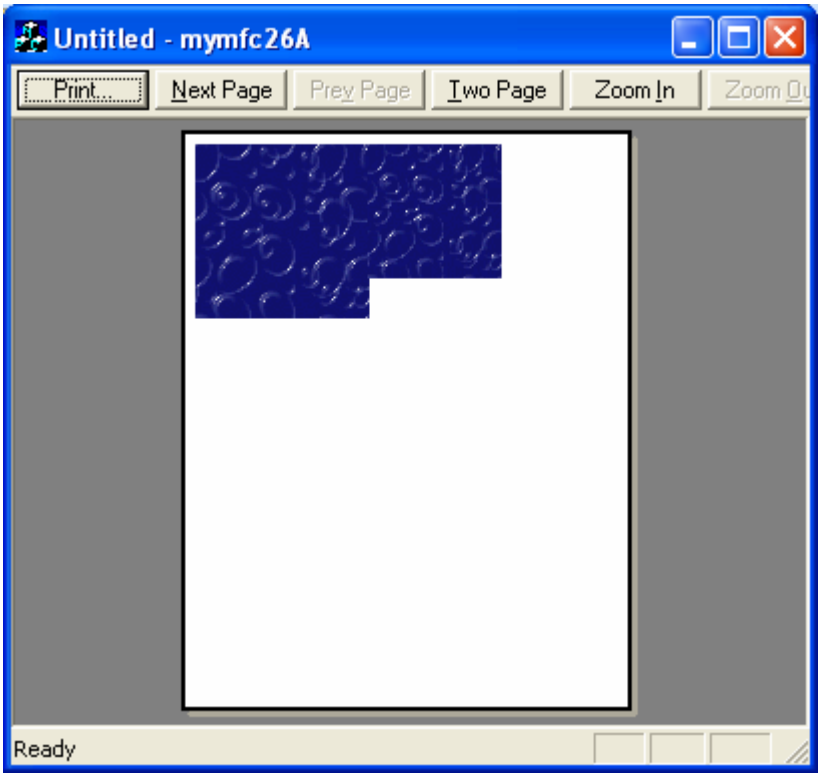

Figure 9: MYMFC26A **Print Preview** view.

The bitmap prints to scale because the application framework applies the `MM_LOENGLISH` mapping mode to the printer device context just as it does to the display device context. The output looks great in **Print Preview** mode, but (depending on your print drivers) the printed output will probably be either blank or microscopic! We'll fix that soon.

Using Bitmaps to Improve the Screen Display

You've seen an example program that displays a bitmap that originated outside the program. Now you'll see an example program that generates its own bitmap to support smooth motion on the screen. The principle is simple: you draw on a memory device context with a bitmap selected, and then you zap the bitmap onto the screen.

The MYMFC26B Example

In the MYMFC6 example in Module 4, the user dragged a circle with the mouse. As the circle moved, the display flickered because the circle was erased and redrawn on every mouse-move message. MYMFC26B uses a GDI bitmap to correct this problem. The MYMFC6 custom code for mouse message processing carries over almost intact; most of the new code is in the `OnPaint()` and `OnInitialUpdate()` functions.
In summary, the MYMFC26B `OnInitialUpdate()` function creates a memory device context and a bitmap that are compatible with the display. The `OnPaint()` function prepares the memory device context for drawing, passes `OnDraw()` a handle to the memory device context, and copies the resulting bitmap from the memory device context to the display.

Here are the steps to build MYMFC26B from scratch:

Run AppWizard to produce \mfcproject\mymfc26B. Accept all the default settings but two: select **Single Document** and select `CScrollView` view as the base class for `CMymfc26BView` in step 6. The options and the default class names are shown here.

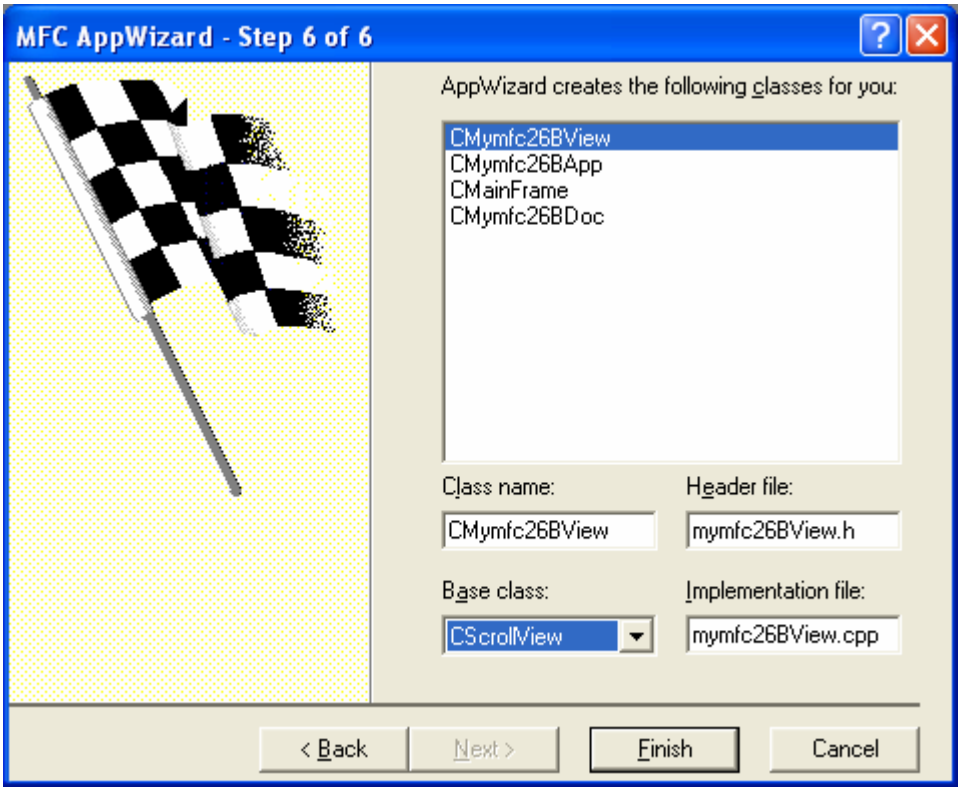

Figure 10: MYMFC26B step 6 of 6 AppWizard using `CScrollView` as a view based class.

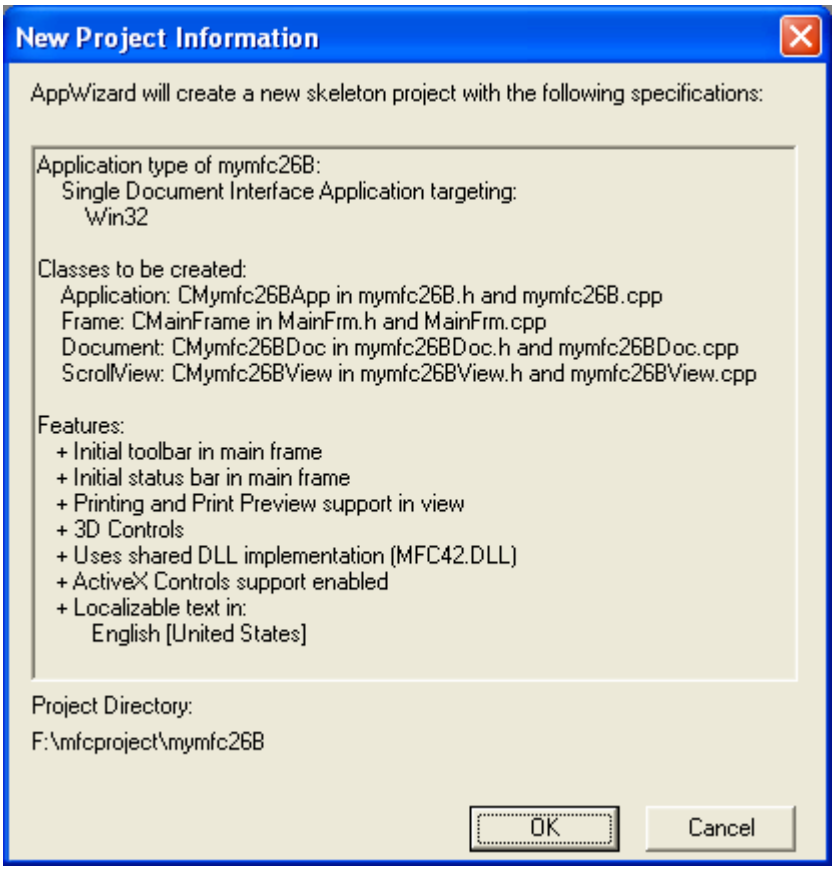

Figure 11: MYMFC26B project summary.

Use ClassWizard to add `CMymfc26BView` message handlers. Add message handlers for the following messages:

- `WM_LBUTTONDOWN`
- `WM_LBUTTONUP`
- `WM_MOUSEMOVE`
- `WM_PAINT`

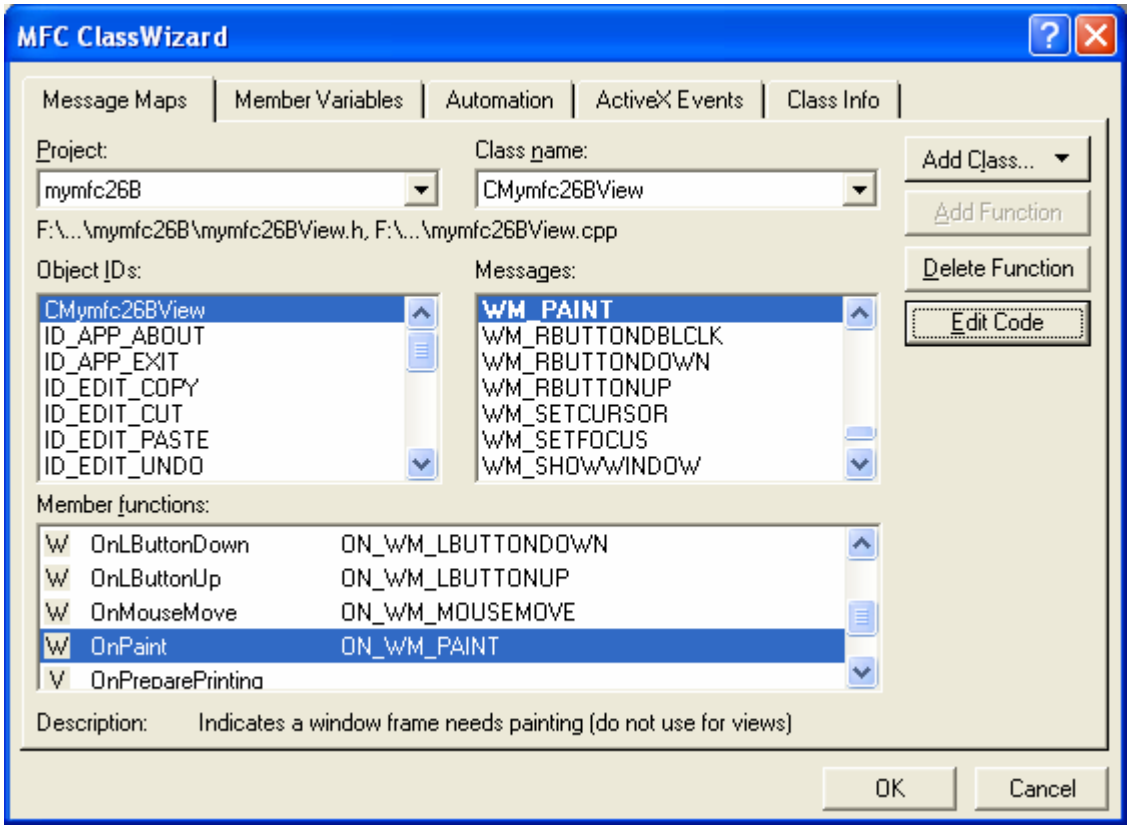

Figure 12: Using ClassWizard to add `CMymfc26BView` message handlers.

Edit the **mymfc26BView.h** header file. Add the private data members shown here to the `CMymfc26BView` class:

```
private:
    const CSize m_sizeEllipse;
    CPoint   m_pointTopLeft;
    BOOL     m_bCaptured;
    CSize    m_sizeOffset;
    CDC*     m_pdcMemory;
    CBitmap* m_pBitmap;
```

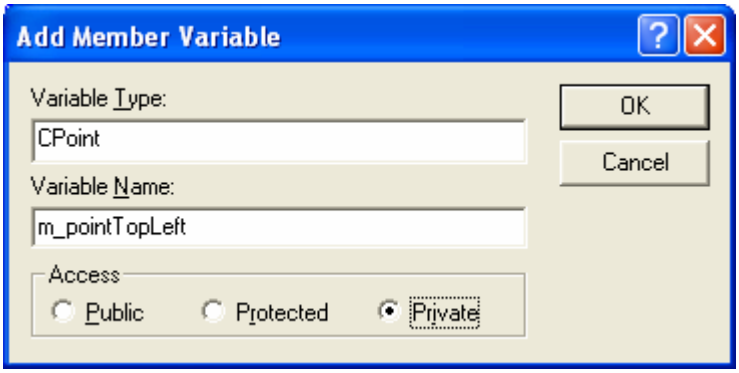

Figure 13: Adding private data members to the `CMymfc26BView` class.

```
    DECLARE_MESSAGE_MAP()
private:
    BOOL     m_bCaptured;
    CSize    m_sizeOffset;
    CDC*     m_pdcMemory;
    CBitmap* m_pBitmap;
    CPoint m_pointTopLeft;
    const CSize m_sizeEllipse;
};
```

Listing 6.

Code the `CMymfc26BView` constructor and destructor in **mymfc26BView.cpp**. You need a memory device context object and a bitmap GDI object. These are constructed in the view's constructor and destroyed in the view's destructor. Add the following code:

```
CMymfc26BView::CMymfc26BView() : m_sizeEllipse(100, -100),
                                 m_pointTopLeft(10, -10),
                                 m_sizeOffset(0, 0)
{
    m_bCaptured = FALSE;
    m_pdcMemory = new CDC;
    m_pBitmap   = new CBitmap;
}
```

```
// CMymfc26BView construction/destruction

CMymfc26BView::CMymfc26BView() : m_sizeEllipse(100, -100),
                                 m_pointTopLeft(10, -10),
                                 m_sizeOffset(0, 0)
{
    // TODO: add construction code here
    m_bCaptured = FALSE;
    m_pdcMemory = new CDC;
    m_pBitmap   = new CBitmap;
}
```

Listing 7.

```
CMymfc26BView::~CMymfc26BView()
{
    delete m_pBitmap; // already deselected
    delete m_pdcMemory;
}
```

```
CMymfc26BView::~CMymfc26BView()
{
    delete m_pBitmap; // already deselected
    delete m_pdcMemory;
}
```

Listing 8.

Add code for the `OnInitialUpdate()` function in **mymfc26BView.cpp**. The C++ memory device context and bitmap objects are already constructed. This function creates the corresponding Windows objects. Both the device context and the bitmap are compatible with the display context dc, but you must explicitly set the memory device context's mapping mode to match the display context. You could create the bitmap in the `OnPaint()` function, but the program runs faster if you create it once here. Add the code shown here:

```
void CMymfc26BView::OnInitialUpdate()
{
    CScrollView::OnInitialUpdate();
    CSize sizeTotal(800, 1050); // 8-by-10.5 inches
    CSize sizePage(sizeTotal.cx / 2, sizeTotal.cy / 2);
    CSize sizeLine(sizeTotal.cx / 50, sizeTotal.cy / 50);
    SetScrollSizes(MM_LOENGLISH, sizeTotal, sizePage, sizeLine);
    // creates the memory device context and the bitmap
    if (m_pdcMemory->GetSafeHdc() == NULL) {
        CClientDC dc(this);
        OnPrepareDC(&dc);
        CRect rectMax(0, 0, sizeTotal.cx, -sizeTotal.cy);
        dc.LPtoDP(rectMax);
        m_pdcMemory->CreateCompatibleDC(&dc);
        // makes bitmap same size as display window
        m_pBitmap->CreateCompatibleBitmap(&dc, rectMax.right, rectMax.bottom);
        m_pdcMemory->SetMapMode(MM_LOENGLISH);
    }
}
```

```
void CMymfc26BView::OnInitialUpdate()
{
        CScrollView::OnInitialUpdate();
    CSize sizeTotal(800, 1050); // 8-by-10.5 inches
    CSize sizePage(sizeTotal.cx / 2, sizeTotal.cy / 2);
    CSize sizeLine(sizeTotal.cx / 50, sizeTotal.cy / 50);
    SetScrollSizes(MM_LOENGLISH, sizeTotal, sizePage, sizeLine);
    // creates the memory device context and the bitmap
    if (m_pdcMemory->GetSafeHdc() == NULL) {
        CClientDC dc(this);
        OnPrepareDC(&dc);
        CRect rectMax(0, 0, sizeTotal.cx, -sizeTotal.cy);
        dc.LPtoDP(rectMax);
        m_pdcMemory->CreateCompatibleDC(&dc);
        // makes bitmap same size as display window
        m_pBitmap->CreateCompatibleBit                    int __thiscall CDC::CreateCompatibleDC(CDC *)Max.bottom);
        m_pdcMemory->SetMapMode(MM_LOENGLISH);
    }
}
```

Listing 9.

Add code for the `OnPaint()` function in **mymfc26BView.cpp**. Normally it isn't necessary to map the `WM_PAINT` message in your derived view class. The `CView` version of `OnPaint()` contains the following code:

```
CPaintDC dc(this);
OnPrepareDC(&dc);
OnDraw(&dc);
```

In this example, you will be using the `OnPaint()` function to reduce screen flicker through the use of a memory device context. `OnDraw()` is passed this memory device context for the display, and it is passed the printer device context for printing. Thus, `OnDraw()` can perform tasks common to the display and to the printer. You don't need to use the bitmap with the printer because the printer has no speed constraint. The `OnPaint()` function must perform, in order, the following three steps to prepare the memory device context for drawing:

- Select the bitmap into the memory device context.
- Transfer the invalid rectangle (as calculated by `OnMouseMove()`) from the display context to the memory device context. There is no `SetClipRect()` function, but the `CDC::IntersectClipRect` function, when called after the `CDC::SelectClipRgn` function (with a `NULL` parameter), has the same effect. If you don't set the clipping rectangle to the minimum size, the program runs more slowly.
- Initialize the bitmap to the current window background color. The `CDC::PatBlt` function fills the specified rectangle with a pattern. In this case, the pattern is the brush pattern for the current window background. That brush must first be constructed and selected into the memory device context.

After the memory device context is prepared, `OnPaint()` can call `OnDraw()` with a memory device context parameter. Then the `CDC::BitBlt` function copies the updated rectangle from the memory device context to the display device context. Add the following code:

```
void CMymfc26BView::OnPaint()
{
    CPaintDC dc(this); // device context for painting
    OnPrepareDC(&dc);
    CRect rectUpdate;
    dc.GetClipBox(&rectUpdate);
    CBitmap* pOldBitmap = m_pdcMemory->SelectObject(m_pBitmap);
    m_pdcMemory->SelectClipRgn(NULL);
    m_pdcMemory->IntersectClipRect(&rectUpdate);
    CBrush backgroundBrush((COLORREF) ::GetSysColor(COLOR_WINDOW));
    CBrush* pOldBrush = m_pdcMemory->SelectObject(&backgroundBrush);
    m_pdcMemory->PatBlt(rectUpdate.left, rectUpdate.top, rectUpdate.Width(),
rectUpdate.Height(), PATCOPY);
    OnDraw(m_pdcMemory);
    dc.BitBlt(rectUpdate.left, rectUpdate.top, rectUpdate.Width(),
rectUpdate.Height(), m_pdcMemory, rectUpdate.left, rectUpdate.top, SRCCOPY);
    m_pdcMemory->SelectObject(pOldBitmap);
    m_pdcMemory->SelectObject(pOldBrush);
}
```

```
void CMymfc26BView::OnPaint()
{
    // TODO: Add your message handler code here
    CPaintDC dc(this);
    OnPrepareDC(&dc);
    CRect rectUpdate;
    dc.GetClipBox(&rectUpdate);

    CBitmap* pOldBitmap = m_pdcMemory->SelectObject(m_pBitmap);
    m_pdcMemory->SelectClipRgn(NULL);
    m_pdcMemory->IntersectClipRect(&rectUpdate);
    CBrush backgroundBrush((COLORREF) ::GetSysColor(COLOR_WINDOW));
    CBrush* pOldBrush = m_pdcMemory->SelectObject(&backgroundBrush);
    m_pdcMemory->PatBlt(rectUpdate.left, rectUpdate.top,
                        rectUpdate.Width(), rectUpdate.Height(), PATCOPY);
    OnDraw(m_pdcMemory);
    dc.BitBlt(rectUpdate.left, rectUpdate.top,
              rectUpdate.Width(), rectUpdate.Height(),
              m_pdcMemory, rectUpdate.left, rectUpdate.top, SRCCOPY);
    m_pdcMemory->SelectObject(pOldBitmap);
    m_pdcMemory->SelectObject(pOldBrush);
    // Do not call CScrollView::OnPaint() for painting messages
}
```

Listing 10.

Code the `OnDraw()` function in **mymfc26BView.cpp**. Copy the code from **mymfc6View.cpp** as shown below. In MYMFC26B, `OnDraw()` is passed a pointer to a memory device context by the `OnPaint()` function. For printing, `OnDraw()` is passed a pointer to the printer device context.

```
void CMymfc26BView::OnDraw(CDC* pDC)
{
    // TODO: add draw code for native data here
    CBrush brushHatch(HS_DIAGCROSS, RGB(255, 0, 0));
    CPoint point(0, 0);                  // logical (0, 0)

    pDC->LPtoDP(&point);                 // In device coordinates,
    pDC->SetBrushOrg(point);             //  align the brush with
                                         //  the window origin
    pDC->SelectObject(&brushHatch);
    pDC->Ellipse(CRect(m_pointTopLeft, m_sizeEllipse));
    pDC->SelectStockObject(BLACK_BRUSH); // Deselect brushHatch
    pDC->Rectangle(CRect(100, -100, 200, -200)); // Test invalid rect
}
```

```
void CMymfc26BView::OnDraw(CDC* pDC)
{
    // TODO: add draw code for native data here
    CBrush brushHatch(HS_DIAGCROSS, RGB(255, 0, 0));
    CPoint point(0, 0);                  // logical (0, 0)

    pDC->LPtoDP(&point);                 // In device coordinates,
    pDC->SetBrushOrg(point);             //  align the brush with
                                         //  the window origin
    pDC->SelectObject(&brushHatch);
    pDC->Ellipse(CRect(m_pointTopLeft, m_sizeEllipse));
    pDC->SelectStockObject(BLACK_BRUSH); // Deselect brushHatch
    pDC->Rectangle(CRect(100, -100, 200, -200)); // Test invalid rect
}
```

Listing 11.

Copy the mouse message-handling code from **mymfc6View.cpp**. Copy the functions shown below from **mymfc6View.cpp** to **mymfc26BView.cpp**. Be sure to change the functions' class names from `CMymfc6View` to `CMymfc26BView`.

- `OnLButtonDown()`
- `OnLButtonUp()`
- `OnMouseMove()`

```
void CMymfc26BView::OnLButtonDown(UINT nFlags, CPoint point)
{
       // TODO: Add your message handler code here and/or call default
       CRect rectEllipse(m_pointTopLeft, m_sizeEllipse); // still logical
       CRgn  circle;

    CClientDC dc(this);
    OnPrepareDC(&dc);
    dc.LPtoDP(rectEllipse); // Now it's in device coordinates
    circle.CreateEllipticRgnIndirect(rectEllipse);
    if (circle.PtInRegion(point)) {
        // Capturing the mouse ensures subsequent LButtonUp message
        SetCapture();
        m_bCaptured = TRUE;
        CPoint pointTopLeft(m_pointTopLeft);
        dc.LPtoDP(&pointTopLeft);
        m_sizeOffset = point - pointTopLeft; // device coordinates
```

```
            // New mouse cursor is active while mouse is captured
            ::SetCursor(::LoadCursor(NULL, IDC_CROSS));
        }
    }
```

```
// CMymfc26BView message handlers

void CMymfc26BView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    CRect rectEllipse(m_pointTopLeft, m_sizeEllipse); // still logical
       CRgn  circle;

    CClientDC dc(this);
    OnPrepareDC(&dc);
    dc.LPtoDP(rectEllipse); // Now it's in device coordinates
    circle.CreateEllipticRgnIndirect(rectEllipse);
    if (circle.PtInRegion(point)) {
        // Capturing the mouse ensures subsequent LButtonUp message
        SetCapture();
        m_bCaptured = TRUE;
        CPoint pointTopLeft(m_pointTopLeft);
        dc.LPtoDP(&pointTopLeft);
        m_sizeOffset = point - pointTopLeft; // device coordinates
        // New mouse cursor is active while mouse is captured
        ::SetCursor(::LoadCursor(NULL, IDC_CROSS));
    }
}
```

Listing 12.

```
void CMymfc26BView::OnLButtonUp(UINT nFlags, CPoint point)
{
       // TODO: Add your message handler code here and/or call default
       if (m_bCaptured)
    {
        ::ReleaseCapture();
        m_bCaptured = FALSE;
    }
}
```

```
void CMymfc26BView::OnLButtonUp(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    if (m_bCaptured)
    {
        ::ReleaseCapture();
        m_bCaptured = FALSE;
    }
}
```

Listing 13.

```
void CMymfc26BView::OnMouseMove(UINT nFlags, CPoint point)
{
     // TODO: Add your message handler code here and/or call default
    if (m_bCaptured)
    {
        CClientDC dc(this);
        OnPrepareDC(&dc);
        CRect rectOld(m_pointTopLeft, m_sizeEllipse);
        dc.LPtoDP(rectOld);
        InvalidateRect(rectOld, TRUE);
```

```
        m_pointTopLeft = point - m_sizeOffset;
        dc.DPtoLP(&m_pointTopLeft);
        CRect rectNew(m_pointTopLeft, m_sizeEllipse);
        dc.LPtoDP(rectNew);
        InvalidateRect(rectNew, TRUE);
    }
}
```

```
void CMymfc26BView::OnMouseMove(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
if (m_bCaptured)
    {
        CClientDC dc(this);
        OnPrepareDC(&dc);
        CRect rectOld(m_pointTopLeft, m_sizeEllipse);
        dc.LPtoDP(rectOld);
        InvalidateRect(rectOld, TRUE);
        m_pointTopLeft = point - m_sizeOffset;
        dc.DPtoLP(&m_pointTopLeft);
        CRect rectNew(m_pointTopLeft, m_sizeEllipse);
        dc.LPtoDP(rectNew);
        InvalidateRect(rectNew, TRUE);
    }
}
```

Listing 14.

Change two lines in the `OnMouseMove()` function in **mymfc26BView.cpp**. Change the following two lines:

```
InvalidateRect(rectOld, TRUE);
...
InvalidateRect(rectNew, TRUE);
```

to

```
InvalidateRect(rectOld, FALSE);
...
InvalidateRect(rectNew, FALSE);
```

If the second `CWnd::InvalidateRect` parameter is `TRUE` (the default), Windows erases the background before repainting the invalid rectangle. That's what you needed in MYMFC6, but the background erasure is what causes the flicker. Because the entire invalid rectangle is being copied from the bitmap, you no longer need to erase the background. The `FALSE` parameter prevents this erasure.

Build and run the application. Here is the MYMFC26B program output.

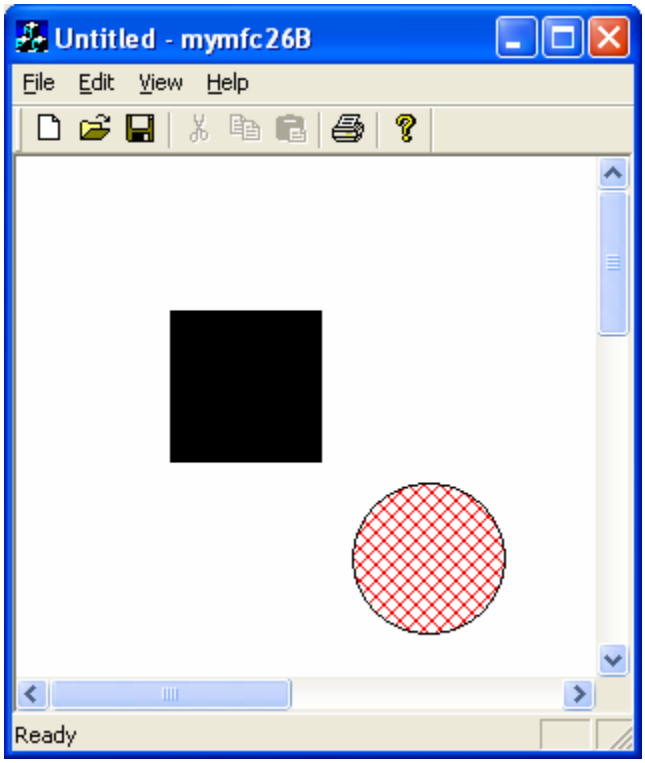

Figure 14: MYMFC26A program output, with smooth movement of the circle bitmap.

Is the circle's movement smoother now? The problem is that the bitmap is only 8-by-10.5 inches, and if the scrolling window is big enough, the circle goes off the edge. One solution to this problem is to make the bitmap as big as the largest display.

Windows Animation

MYMFC26B is a crude attempt at Windows animation. What if you wanted to move an angelfish instead of a circle? Win32 doesn't have an Angelfish function (yet), so you'd have to keep your angelfish in its own bitmap and use the `StretchBlt()` mask ROP codes to merge the angelfish with the background. You'd probably keep the background in its own bitmap, too. These techniques are outside the scope of this book. If you are interested in learning more about Windows Animation, run out and get Nigel Thompson's Animation Techniques in Win32 (Microsoft Press, 1995). After you read it, you can get rich writing video games for Windows!

DIBs and the `CDib` Class

There's an MFC class for GDI bitmaps (`CBitmap`), but there's no MFC class for DIBs. Don't worry, I'm giving you one here. It's a complete rewrite of the `CDib` class from the early editions of this book (prior to the fourth edition), and it takes advantage of Win32 features such as memory-mapped files, improved memory management, and DIB sections. It also includes palette support. Before you examine the `CDib` class, however, you need a little background on DIBs.

A Few Words About Palette Programming

Windows palette programming is quite complex, but you've got to deal with it if you expect your users to run their displays in the 8-bpp (bits per pixel) mode and many users will if they have video cards with 1 MB or less of memory. Suppose you're displaying a single DIB in a window. First you must create a logical palette, a GDI object that contains the colors in the DIB. Then you must "realize" this logical palette into the hardware system palette, a table of the 256 colors the video card can display at that instant. If your program is the foreground program, the realization process tries to copy all your colors into the system palette, but it doesn't touch the 20 standard Windows colors. For the most part, your DIB looks just like you want it to look.

But what if another program is the foreground program, and what if that program has a forest scene DIB with 236 shades of green? Your program still realizes its palette, but something different happens this time. Now the system palette won't change, but Windows sets up a new mapping between your logical palette and the system palette. If your DIB contains a neon pink color, for example, Windows maps it to the standard red color. If your program forgot to realize its palette, your neon pink stuff would turn green when the other program went active.
The forest scene example is extreme because we assumed that the other program grabbed 236 colors. If instead the other program realized a logical palette with only 200 colors, Windows would let your program load 36 of its own colors, including, one hopes, neon pink.
So when is a program supposed to realize its palette? The Windows message `WM_PALETTECHANGED` is sent to your program's main window whenever a program, including yours, realizes its palette. Another message, `WM_QUERYNEWPALETTE`, is sent whenever one of the windows in your program gets the input focus. Your program should realize its palette in response to both these messages (unless your program generated the message). These palette messages are not sent to your view window, however. You must map them in your application's main frame window and then notify the view.
You call the Win32 `RealizePalette()` function to perform the realization, but first you must call `SelectPalette()` to select your DIB's logical palette into the device context. `SelectPalette()` has a flag parameter that you normally set to `FALSE` in your `WM_PALETTECHANGED` and `WM_QUERYNEWPALETTE` handlers. This flag ensures that your palette is realized as a foreground palette if your application is indeed running in the foreground. If you use a `TRUE` flag parameter here, you can force Windows to realize the palette as though the application were in the background.
You must also call `SelectPalette()` for each DIB that you display in your `OnDraw()` function. This time you call it with a `TRUE` flag parameter. Things do get complicated if you're displaying several DIBs, each with its own palette. Basically, you've got to choose a palette for one of the DIBs and realize it (by selecting it with the `FALSE` parameter) in the palette message handlers. The chosen DIB will end up looking better than the other DIBs. There are ways of merging palettes, but it might be easier to go out and buy more video memory.

DIBs, Pixels, and Color Tables

A DIB contains a two-dimensional array of elements called pixels. In many cases, each DIB pixel will be mapped to a display pixel, but the DIB pixel might be mapped to some logical area on the display, depending on the mapping mode and the display function stretch parameters.
A pixel consists of 1, 4, 8, 16, 24, or 32 contiguous bits, depending on the color resolution of the DIB. For 16-bpp, 24-bpp, and 32-bpp DIBs, each pixel represents an RGB color. A pixel in a 16-bpp DIB typically contains 5 bits each for red, green, and blue values; a pixel in a 24-bpp DIB has 8 bits for each color value. The 16-bpp and 24-bpp DIBs are optimized for video cards that can display 65,536 or 16.7 million simultaneous colors.
A **1-bpp DIB is a monochrome DIB**, but these DIBs don't have to be black and white, they can contain any two colors chosen from the color table that is built into each DIB. A monochrome bitmap has two 32-bit color table entries, each containing 8 bits for red, green, and blue values plus another 8 bits for flags. Zero (0) pixels use the first entry, and one (1) pixel uses the second. Whether you have a 65,536-color video card or a 16.7-million-color card, Windows can display the two colors directly. (Windows truncates 8-bits-per-color values to 5 bits for 65,536-color displays.) If your video card is running in 256-color palletized mode, your program can adjust the system palette to load the two specified colors.
Eight-bpp DIBs are quite common. Like a monochrome DIB, an 8-bpp DIB has a color table, but the color table has 256 (or fewer) 32-bit entries. Each pixel is an index into this color table. If you have a palletized video card, your program can create a logical palette from the 256 entries. If another program (running in the foreground) has control of the system palette, Windows does its best to match your logical palette colors to the system palette.
What if you're trying to display a 24-bpp DIB with a 256-color palletized video card? If the DIB author was nice, he or she included a color table containing the most important colors in the DIB. Your program can build a logical palette from that table, and the DIB will look fine. If the DIB has no color table, use the palette returned by the Win32 `CreateHalftonePalette()` function; it's better than the 20 standard colors you'd get with no palette at all. Another option is to analyze the DIB to identify the most important colors, but you can buy a utility to do that.

The Structure of a DIB Within a BMP File

You know that the DIB is the **standard Windows bitmap format** and that a BMP file contains a DIB. So let's look inside a BMP file to see what's there. Figure 15 shows a layout for a BMP file.

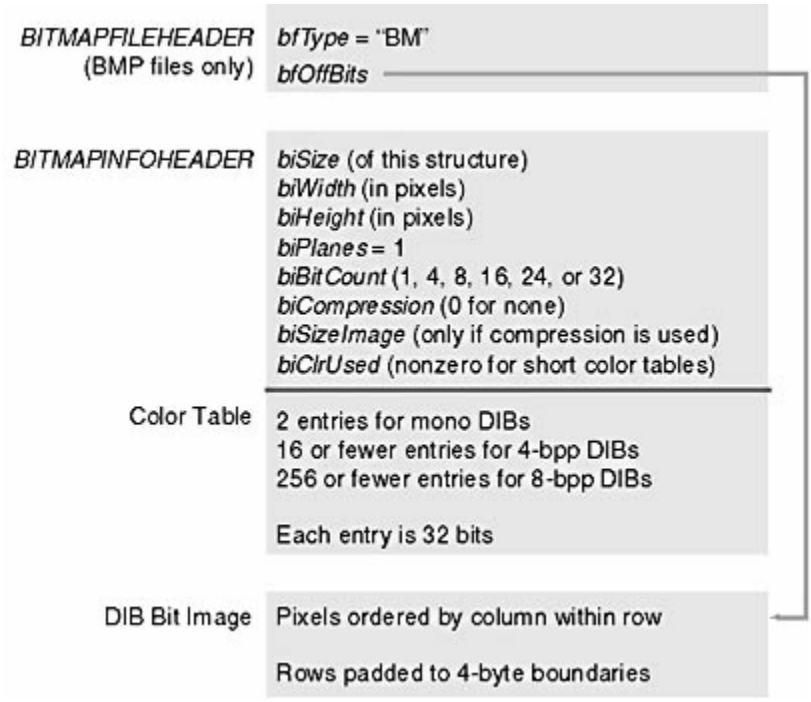

Figure 15: The layout for a BMP file.

The `BITMAPFILEHEADER` structure contains the offset to the image bits, which you can use to compute the combined size of the `BITMAPINFOHEADER` structure and the color table that follows. The `BITMAPFILEHEADER` structure contains a file size member, but you can't depend on it because you don't know whether the size is measured in bytes, words, or double words.
The `BITMAPINFOHEADER` structure contains the bitmap dimensions, the bits per pixel, compression information for both 4-bpp and 8-bpp bitmaps, and the number of color table entries. If the DIB is compressed, this header contains the size of the pixel array; otherwise, you can compute the size from the dimensions and the bits per pixel. Immediately following the header is the color table (if the DIB has a color table). The DIB image comes after that. The DIB image consists of pixels arranged by column within rows, starting with the bottom row. Each row is padded to a 4-byte boundary. The only place you'll find a `BITMAPFILEHEADER` structure, however, is in a BMP file. If you get a DIB from the clipboard, for example, there will not be a file header. You can always count on the color table to follow the `BITMAPINFOHEADER` structure, but you can't count on the image to follow the color table. If you're using the `CreateDIBSection()` function, for example, you must allocate the bitmap info header and color table and then let Windows allocate the image somewhere else. This module and all the associated code are specific to Windows DIBs. There's also a well-documented variation of the DIB format for OS/2. If you need to process these OS/2 DIBs, you'll have to modify the `CDib` class.

DIB Access Functions

Windows supplies some important DIB access functions. None of these functions is wrapped by MFC, so you'll need to refer to the online Win32 documentation for details. Here's a summary:

- **`SetDIBitsToDevice()`**: This function displays a DIB directly on the display or printer. No scaling occurs; one bitmap bit corresponds to one display pixel or one printer dot. This scaling restriction limits the function's usefulness. The function doesn't work like `BitBlt()` because `BitBlt()` uses logical coordinates.
- **`StretchDIBits()`**: This function displays a DIB directly on the display or printer in a manner similar to that of `StretchBlt()`.
- **`GetDIBits()`**: This function constructs a DIB from a GDI bitmap, using memory that you allocate. You have some control over the format of the DIB because you can specify the number of color bits per pixel and the compression. If you are using compression, you have to call `GetDIBits()` twice, once to calculate the memory needed and again to generate the DIB data.

- **`CreateDIBitmap()`**: This function creates a GDI bitmap from a DIB. As for all these DIB functions, you must supply a device context pointer as a parameter. A display device context will do; you don't need a memory device context.
- **`CreateDIBSection()`**: This Win32 function creates a special kind of DIB known as a DIB section. It then returns a GDI bitmap handle. This function gives you the best features of DIBs and GDI bitmaps. You have direct access to the DIB's memory, and with the bitmap handle and a memory device context, you can call GDI functions to draw into the DIB.

The `CDib` Class

If DIBs look intimidating, don't worry. The `CDib` class makes DIB programming easy. The best way to get to know the `CDib` class is to look at the public member functions and data members. Listing 15 shows the `CDib` header and implementation files.

CDIB.H

```
#ifndef _INSIDE_VISUAL_CPP_CDIB
#define _INSIDE_VISUAL_CPP_CDIB

class CDib : public CObject
{
    enum Alloc {noAlloc, crtAlloc,
                heapAlloc}; // applies to BITMAPINFOHEADER
    DECLARE_SERIAL(CDib)
public:
    LPVOID  m_lpvColorTable;
    HBITMAP m_hBitmap;
    LPBYTE  m_lpImage;  // starting address of DIB bits
    LPBITMAPINFOHEADER m_lpBMIH; // buffer containing the
                                 //  BITMAPINFOHEADER
private:
    HGLOBAL m_hGlobal; // for external windows we need to free;
                       //  could be allocated by this class or
                       //  allocated externally
    Alloc m_nBmihAlloc;
    Alloc m_nImageAlloc;
    DWORD m_dwSizeImage; // of bits—not BITMAPINFOHEADER
                         // or BITMAPFILEHEADER
    int m_nColorTableEntries;

    HANDLE m_hFile;
    HANDLE m_hMap;
    LPVOID m_lpvFile;
    HPALETTE m_hPalette;
public:
    CDib();
    CDib(CSize size, int nBitCount);  // builds BITMAPINFOHEADER
    ~CDib();
    int GetSizeImage() {return m_dwSizeImage;}
    int GetSizeHeader()
        {return sizeof(BITMAPINFOHEADER) + sizeof(RGBQUAD) * m_nColorTableEntries;}
    CSize GetDimensions();
    BOOL AttachMapFile(const char* strPathname, BOOL bShare = FALSE);
    BOOL CopyToMapFile(const char* strPathname);
    BOOL AttachMemory(LPVOID lpvMem, BOOL bMustDelete = FALSE, HGLOBAL hGlobal =
NULL);
    BOOL Draw(CDC* pDC, CPoint origin,
        CSize size); // until we implement CreateDibSection
    HBITMAP CreateSection(CDC* pDC = NULL);
    UINT UsePalette(CDC* pDC, BOOL bBackground = FALSE);
    BOOL MakePalette();
    BOOL SetSystemPalette(CDC* pDC);
    BOOL Compress(CDC* pDC, BOOL bCompress = TRUE); // FALSE means decompress
    HBITMAP CreateBitmap(CDC* pDC);
    BOOL Read(CFile* pFile);
```

```
    BOOL ReadSection(CFile* pFile, CDC* pDC = NULL);
    BOOL Write(CFile* pFile);
    void Serialize(CArchive& ar);
    void Empty();
private:
    void DetachMapFile();
    void ComputePaletteSize(int nBitCount);
    void ComputeMetrics();
};
#endif // _INSIDE_VISUAL_CPP_CDIB
```

CDIB.CPP

```
// cdib.cpp
// new version for WIN32
#include "stdafx.h"
#include "cdib.h"

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

IMPLEMENT_SERIAL(CDib, CObject, 0);

CDib::CDib()
{
       m_hFile = NULL;
       m_hBitmap = NULL;
       m_hPalette = NULL;
       m_nBmihAlloc = m_nImageAlloc = noAlloc;
       Empty();
}

CDib::CDib(CSize size, int nBitCount)
{
       m_hFile = NULL;
       m_hBitmap = NULL;
       m_hPalette = NULL;
       m_nBmihAlloc = m_nImageAlloc = noAlloc;
       Empty();
       ComputePaletteSize(nBitCount);
       m_lpBMIH = (LPBITMAPINFOHEADER) new
              char[sizeof(BITMAPINFOHEADER) + sizeof(RGBQUAD) * m_nColorTableEntries];
       m_nBmihAlloc = crtAlloc;
       m_lpBMIH->biSize = sizeof(BITMAPINFOHEADER);
       m_lpBMIH->biWidth = size.cx;
       m_lpBMIH->biHeight = size.cy;
       m_lpBMIH->biPlanes = 1;
       m_lpBMIH->biBitCount = nBitCount;
       m_lpBMIH->biCompression = BI_RGB;
       m_lpBMIH->biSizeImage = 0;
       m_lpBMIH->biXPelsPerMeter = 0;
       m_lpBMIH->biYPelsPerMeter = 0;
       m_lpBMIH->biClrUsed = m_nColorTableEntries;
       m_lpBMIH->biClrImportant = m_nColorTableEntries;
       ComputeMetrics();
       memset(m_lpvColorTable, 0, sizeof(RGBQUAD) * m_nColorTableEntries);
       m_lpImage = NULL;  // no data yet
}

CDib::~CDib()
{
       Empty();
}
```

```
CSize CDib::GetDimensions()
{
	if(m_lpBMIH == NULL) return CSize(0, 0);
	return CSize((int) m_lpBMIH->biWidth, (int) m_lpBMIH->biHeight);
}

BOOL CDib::AttachMapFile(const char* strPathname, BOOL bShare) // for reading
{
	// if we open the same file twice, Windows treats it as 2 separate files
	// doesn't work with rare BMP files where # palette entries > biClrUsed
	HANDLE hFile = ::CreateFile(strPathname, GENERIC_WRITE | GENERIC_READ,
		bShare ? FILE_SHARE_READ : 0,
		NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
	ASSERT(hFile != INVALID_HANDLE_VALUE);
	DWORD dwFileSize = ::GetFileSize(hFile, NULL);
	HANDLE hMap = ::CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0, 0, NULL);
	DWORD dwErr = ::GetLastError();
	if(hMap == NULL) {
		AfxMessageBox("Empty bitmap file");
		return FALSE;
	}
	LPVOID lpvFile = ::MapViewOfFile(hMap, FILE_MAP_WRITE, 0, 0, 0); // map whole
file
	ASSERT(lpvFile != NULL);
	if(((LPBITMAPFILEHEADER) lpvFile)->bfType != 0x4d42) {
		AfxMessageBox("Invalid bitmap file");
		DetachMapFile();
		return FALSE;
	}
	AttachMemory((LPBYTE) lpvFile + sizeof(BITMAPFILEHEADER));
	m_lpvFile = lpvFile;
	m_hFile = hFile;
	m_hMap = hMap;
	return TRUE;
}

BOOL CDib::CopyToMapFile(const char* strPathname)
{
	// copies DIB to a new file, releases prior pointers
	// if you previously used CreateSection, the HBITMAP will be NULL (and
unusable)
	BITMAPFILEHEADER bmfh;
	bmfh.bfType = 0x4d42;  // 'BM'
	bmfh.bfSize = m_dwSizeImage + sizeof(BITMAPINFOHEADER) +
			sizeof(RGBQUAD) * m_nColorTableEntries +
sizeof(BITMAPFILEHEADER);
	// meaning of bfSize open to interpretation
	bmfh.bfReserved1 = bmfh.bfReserved2 = 0;
	bmfh.bfOffBits = sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFOHEADER) +
			sizeof(RGBQUAD) * m_nColorTableEntries;
	HANDLE hFile = ::CreateFile(strPathname, GENERIC_WRITE | GENERIC_READ, 0, NULL,
		CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
	ASSERT(hFile != INVALID_HANDLE_VALUE);
	int nSize =  sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFOHEADER) +
				sizeof(RGBQUAD) * m_nColorTableEntries +  m_dwSizeImage;
	HANDLE hMap = ::CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0, nSize, NULL);
	DWORD dwErr = ::GetLastError();
	ASSERT(hMap != NULL);
	LPVOID lpvFile = ::MapViewOfFile(hMap, FILE_MAP_WRITE, 0, 0, 0); // map whole
file
	ASSERT(lpvFile != NULL);
	LPBYTE lpbCurrent = (LPBYTE) lpvFile;
	memcpy(lpbCurrent, &bmfh, sizeof(BITMAPFILEHEADER)); // file header
	lpbCurrent += sizeof(BITMAPFILEHEADER);
	LPBITMAPINFOHEADER lpBMIH = (LPBITMAPINFOHEADER) lpbCurrent;
	memcpy(lpbCurrent, m_lpBMIH,
```

```
                sizeof(BITMAPINFOHEADER) + sizeof(RGBQUAD) * m_nColorTableEntries); //
info
        lpbCurrent += sizeof(BITMAPINFOHEADER) + sizeof(RGBQUAD) *
m_nColorTableEntries;
        memcpy(lpbCurrent, m_lpImage, m_dwSizeImage); // bit image
        DWORD dwSizeImage = m_dwSizeImage;
        Empty();
        m_dwSizeImage = dwSizeImage;
        m_nBmihAlloc = m_nImageAlloc = noAlloc;
        m_lpBMIH = lpBMIH;
        m_lpImage = lpbCurrent;
        m_hFile = hFile;
        m_hMap = hMap;
        m_lpvFile = lpvFile;
        ComputePaletteSize(m_lpBMIH->biBitCount);
        ComputeMetrics();
        MakePalette();
        return TRUE;
}

BOOL CDib::AttachMemory(LPVOID lpvMem, BOOL bMustDelete, HGLOBAL hGlobal)
{
        // assumes contiguous BITMAPINFOHEADER, color table, image
        // color table could be zero length
        Empty();
        m_hGlobal = hGlobal;
        if(bMustDelete == FALSE) {
                m_nBmihAlloc = noAlloc;
        }
        else {
                m_nBmihAlloc = ((hGlobal == NULL) ? crtAlloc : heapAlloc);
        }
        try {
                m_lpBMIH = (LPBITMAPINFOHEADER) lpvMem;
                ComputeMetrics();
                ComputePaletteSize(m_lpBMIH->biBitCount);
                m_lpImage = (LPBYTE) m_lpvColorTable + sizeof(RGBQUAD) *
m_nColorTableEntries;
                MakePalette();
        }
        catch(CException* pe) {
                AfxMessageBox("AttachMemory error");
                pe->Delete();
                return FALSE;
        }
        return TRUE;
}

UINT CDib::UsePalette(CDC* pDC, BOOL bBackground /* = FALSE */)
{
        if(m_hPalette == NULL) return 0;
        HDC hdc = pDC->GetSafeHdc();
        ::SelectPalette(hdc, m_hPalette, bBackground);
        return ::RealizePalette(hdc);
}

BOOL CDib::Draw(CDC* pDC, CPoint origin, CSize size)
{
        if(m_lpBMIH == NULL) return FALSE;
        if(m_hPalette != NULL) {
                ::SelectPalette(pDC->GetSafeHdc(), m_hPalette, TRUE);
        }
        pDC->SetStretchBltMode(COLORONCOLOR);
        ::StretchDIBits(pDC->GetSafeHdc(), origin.x, origin.y, size.cx, size.cy,
                0, 0, m_lpBMIH->biWidth, m_lpBMIH->biHeight,
                m_lpImage, (LPBITMAPINFO) m_lpBMIH, DIB_RGB_COLORS, SRCCOPY);
```

```
	return TRUE;
}

HBITMAP CDib::CreateSection(CDC* pDC /* = NULL */)
{
	if(m_lpBMIH == NULL) return NULL;
	if(m_lpImage != NULL) return NULL; // can only do this if image doesn't exist
	m_hBitmap = ::CreateDIBSection(pDC->GetSafeHdc(), (LPBITMAPINFO) m_lpBMIH,
		DIB_RGB_COLORS,	(LPVOID*) &m_lpImage, NULL, 0);
	ASSERT(m_lpImage != NULL);
	return m_hBitmap;
}

BOOL CDib::MakePalette()
{
	// makes a logical palette (m_hPalette) from the DIB's color table
	// this palette will be selected and realized prior to drawing the DIB
	if(m_nColorTableEntries == 0) return FALSE;
	if(m_hPalette != NULL) ::DeleteObject(m_hPalette);
	TRACE("CDib::MakePalette -- m_nColorTableEntries = %d\n",
m_nColorTableEntries);
	LPLOGPALETTE pLogPal = (LPLOGPALETTE) new char[2 * sizeof(WORD) +
		m_nColorTableEntries * sizeof(PALETTEENTRY)];
	pLogPal->palVersion = 0x300;
	pLogPal->palNumEntries = m_nColorTableEntries;
	LPRGBQUAD pDibQuad = (LPRGBQUAD) m_lpvColorTable;
	for(int i = 0; i < m_nColorTableEntries; i++) {
		pLogPal->palPalEntry[i].peRed = pDibQuad->rgbRed;
		pLogPal->palPalEntry[i].peGreen = pDibQuad->rgbGreen;
		pLogPal->palPalEntry[i].peBlue = pDibQuad->rgbBlue;
		pLogPal->palPalEntry[i].peFlags = 0;
		pDibQuad++;
	}
	m_hPalette = ::CreatePalette(pLogPal);
	delete pLogPal;
	return TRUE;
}

BOOL CDib::SetSystemPalette(CDC* pDC)
{
	// if the DIB doesn't have a color table, we can use the system's halftone
palette
	if(m_nColorTableEntries != 0) return FALSE;
	m_hPalette = ::CreateHalftonePalette(pDC->GetSafeHdc());
	return TRUE;
}

HBITMAP CDib::CreateBitmap(CDC* pDC)
{
    if (m_dwSizeImage == 0) return NULL;
    HBITMAP hBitmap = ::CreateDIBitmap(pDC->GetSafeHdc(), m_lpBMIH,
            CBM_INIT, m_lpImage, (LPBITMAPINFO) m_lpBMIH, DIB_RGB_COLORS);
    ASSERT(hBitmap != NULL);
    return hBitmap;
}

BOOL CDib::Compress(CDC* pDC, BOOL bCompress /* = TRUE */)
{
	// 1. makes GDI bitmap from existing DIB
	// 2. makes a new DIB from GDI bitmap with compression
	// 3. cleans up the original DIB
	// 4. puts the new DIB in the object
	if((m_lpBMIH->biBitCount != 4) && (m_lpBMIH->biBitCount != 8)) return FALSE;
		// compression supported only for 4 bpp and 8 bpp DIBs
	if(m_hBitmap) return FALSE; // can't compress a DIB Section!
	TRACE("Compress: original palette size = %d\n", m_nColorTableEntries);
```

```
        HDC hdc = pDC->GetSafeHdc();
        HPALETTE hOldPalette = ::SelectPalette(hdc, m_hPalette, FALSE);
        HBITMAP hBitmap;  // temporary
        if((hBitmap = CreateBitmap(pDC)) == NULL) return FALSE;
        int nSize = sizeof(BITMAPINFOHEADER) + sizeof(RGBQUAD) * m_nColorTableEntries;
        LPBITMAPINFOHEADER lpBMIH = (LPBITMAPINFOHEADER) new char[nSize];
        memcpy(lpBMIH, m_lpBMIH, nSize);  // new header
        if(bCompress) {
                switch (lpBMIH->biBitCount) {
                case 4:
                        lpBMIH->biCompression = BI_RLE4;
                        break;
                case 8:
                        lpBMIH->biCompression = BI_RLE8;
                        break;
                default:
                        ASSERT(FALSE);
                }
                // calls GetDIBits with null data pointer to get size of compressed DIB
                if(!::GetDIBits(pDC->GetSafeHdc(), hBitmap, 0, (UINT) lpBMIH->biHeight,
                                        NULL, (LPBITMAPINFO) lpBMIH, DIB_RGB_COLORS))
{
                        AfxMessageBox("Unable to compress this DIB");
                        // probably a problem with the color table
                        ::DeleteObject(hBitmap);
                        delete [] lpBMIH;
                        ::SelectPalette(hdc, hOldPalette, FALSE);
                        return FALSE;
                }
                if (lpBMIH->biSizeImage == 0) {
                        AfxMessageBox("Driver can't do compression");
                        ::DeleteObject(hBitmap);
                        delete [] lpBMIH;
                        ::SelectPalette(hdc, hOldPalette, FALSE);
                        return FALSE;
                }
                else {
                        m_dwSizeImage = lpBMIH->biSizeImage;
                }
        }
        else {
                lpBMIH->biCompression = BI_RGB; // decompress
                // figure the image size from the bitmap width and height
                DWORD dwBytes = ((DWORD) lpBMIH->biWidth * lpBMIH->biBitCount) / 32;
                if(((DWORD) lpBMIH->biWidth * lpBMIH->biBitCount) % 32) {
                        dwBytes++;
                }
                dwBytes *= 4;
                m_dwSizeImage = dwBytes * lpBMIH->biHeight; // no compression
                lpBMIH->biSizeImage = m_dwSizeImage;
        }
        // second GetDIBits call to make DIB
        LPBYTE lpImage = (LPBYTE) new char[m_dwSizeImage];
        VERIFY(::GetDIBits(pDC->GetSafeHdc(), hBitmap, 0, (UINT) lpBMIH->biHeight,
                lpImage, (LPBITMAPINFO) lpBMIH, DIB_RGB_COLORS));
    TRACE("dib successfully created - height = %d\n", lpBMIH->biHeight);
        ::DeleteObject(hBitmap);
        Empty();
        m_nBmihAlloc = m_nImageAlloc = crtAlloc;
        m_lpBMIH = lpBMIH;
        m_lpImage = lpImage;
        ComputeMetrics();
        ComputePaletteSize(m_lpBMIH->biBitCount);
        MakePalette();
        ::SelectPalette(hdc, hOldPalette, FALSE);
        TRACE("Compress: new palette size = %d\n", m_nColorTableEntries);
```

```
	return TRUE;
}

BOOL CDib::Read(CFile* pFile)
{
	// 1. read file header to get size of info hdr + color table
	// 2. read info hdr (to get image size) and color table
	// 3. read image
	// can't use bfSize in file header
	Empty();
	int nCount, nSize;
	BITMAPFILEHEADER bmfh;
	try {
		nCount = pFile->Read((LPVOID) &bmfh, sizeof(BITMAPFILEHEADER));
		if(nCount != sizeof(BITMAPFILEHEADER)) {
			throw new CException;
		}
		if(bmfh.bfType != 0x4d42) {
			throw new CException;
		}
		nSize = bmfh.bfOffBits - sizeof(BITMAPFILEHEADER);
		m_lpBMIH = (LPBITMAPINFOHEADER) new char[nSize];
		m_nBmihAlloc = m_nImageAlloc = crtAlloc;
		nCount = pFile->Read(m_lpBMIH, nSize); // info hdr & color table
		ComputeMetrics();
		ComputePaletteSize(m_lpBMIH->biBitCount);
		MakePalette();
		m_lpImage = (LPBYTE) new char[m_dwSizeImage];
		nCount = pFile->Read(m_lpImage, m_dwSizeImage); // image only
	}
	catch(CException* pe) {
		AfxMessageBox("Read error");
		pe->Delete();
		return FALSE;
	}
	return TRUE;
}

BOOL CDib::ReadSection(CFile* pFile, CDC* pDC /* = NULL */)
{
	// new function reads BMP from disk and creates a DIB section
	//    allows modification of bitmaps from disk
	// 1. read file header to get size of info hdr + color table
	// 2. read info hdr (to get image size) and color table
	// 3. create DIB section based on header parms
	// 4. read image into memory that CreateDibSection allocates
	Empty();
	int nCount, nSize;
	BITMAPFILEHEADER bmfh;
	try {
		nCount = pFile->Read((LPVOID) &bmfh, sizeof(BITMAPFILEHEADER));
		if(nCount != sizeof(BITMAPFILEHEADER)) {
			throw new CException;
		}
		if(bmfh.bfType != 0x4d42) {
			throw new CException;
		}
		nSize = bmfh.bfOffBits - sizeof(BITMAPFILEHEADER);
		m_lpBMIH = (LPBITMAPINFOHEADER) new char[nSize];
		m_nBmihAlloc = crtAlloc;
		m_nImageAlloc = noAlloc;
		nCount = pFile->Read(m_lpBMIH, nSize); // info hdr & color table
		if(m_lpBMIH->biCompression != BI_RGB) {
			throw new CException;
		}
		ComputeMetrics();
```

```
		ComputePaletteSize(m_lpBMIH->biBitCount);
		MakePalette();
		UsePalette(pDC);
		m_hBitmap = ::CreateDIBSection(pDC->GetSafeHdc(), (LPBITMAPINFO)
m_lpBMIH,
			DIB_RGB_COLORS,	(LPVOID*) &m_lpImage, NULL, 0);
		ASSERT(m_lpImage != NULL);
		nCount = pFile->Read(m_lpImage, m_dwSizeImage); // image only
	}
	catch(CException* pe) {
		AfxMessageBox("ReadSection error");
		pe->Delete();
		return FALSE;
	}
	return TRUE;
}

BOOL CDib::Write(CFile* pFile)
{
	BITMAPFILEHEADER bmfh;
	bmfh.bfType = 0x4d42;  // 'BM'
	int nSizeHdr = sizeof(BITMAPINFOHEADER) + sizeof(RGBQUAD) *
m_nColorTableEntries;
	bmfh.bfSize = 0;
//	bmfh.bfSize = sizeof(BITMAPFILEHEADER) + nSizeHdr + m_dwSizeImage;
// meaning of bfSize open to interpretation (bytes, words, dwords?) -- we won't use it
	bmfh.bfReserved1 = bmfh.bfReserved2 = 0;
	bmfh.bfOffBits = sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFOHEADER) +
			sizeof(RGBQUAD) * m_nColorTableEntries;
	try {
		pFile->Write((LPVOID) &bmfh, sizeof(BITMAPFILEHEADER));
		pFile->Write((LPVOID) m_lpBMIH,  nSizeHdr);
		pFile->Write((LPVOID) m_lpImage, m_dwSizeImage);
	}
	catch(CException* pe) {
		pe->Delete();
		AfxMessageBox("write error");
		return FALSE;
	}
	return TRUE;
}

void CDib::Serialize(CArchive& ar)
{
	DWORD dwPos;
	dwPos = ar.GetFile()->GetPosition();
	TRACE("CDib::Serialize -- pos = %d\n", dwPos);
	ar.Flush();
	dwPos = ar.GetFile()->GetPosition();
	TRACE("CDib::Serialize -- pos = %d\n", dwPos);
	if(ar.IsStoring()) {
		Write(ar.GetFile());
	}
	else {
		Read(ar.GetFile());
	}
}

// helper functions
void CDib::ComputePaletteSize(int nBitCount)
{
	if((m_lpBMIH == NULL) || (m_lpBMIH->biClrUsed == 0)) {
		switch(nBitCount) {
			case 1:
				m_nColorTableEntries = 2;
				break;
```

```
                        case 4:
                                m_nColorTableEntries = 16;
                                break;
                        case 8:
                                m_nColorTableEntries = 256;
                                break;
                        case 16:
                        case 24:
                        case 32:
                                m_nColorTableEntries = 0;
                                break;
                        default:
                                ASSERT(FALSE);
                }
        }
        else {
                m_nColorTableEntries = m_lpBMIH->biClrUsed;
        }
        ASSERT((m_nColorTableEntries >= 0) && (m_nColorTableEntries <= 256));
}

void CDib::ComputeMetrics()
{
        if(m_lpBMIH->biSize != sizeof(BITMAPINFOHEADER)) {
                TRACE("Not a valid Windows bitmap -- probably an OS/2 bitmap\n");
                throw new CException;
        }
        m_dwSizeImage = m_lpBMIH->biSizeImage;
        if(m_dwSizeImage == 0) {
                DWORD dwBytes = ((DWORD) m_lpBMIH->biWidth * m_lpBMIH->biBitCount) / 32;
                if(((DWORD) m_lpBMIH->biWidth * m_lpBMIH->biBitCount) % 32) {
                        dwBytes++;
                }
                dwBytes *= 4;
                m_dwSizeImage = dwBytes * m_lpBMIH->biHeight; // no compression
        }
        m_lpvColorTable = (LPBYTE) m_lpBMIH + sizeof(BITMAPINFOHEADER);
}

void CDib::Empty()
{
        // this is supposed to clean up whatever is in the DIB
        DetachMapFile();
        if(m_nBmihAlloc == crtAlloc) {
                delete [] m_lpBMIH;
        }
        else if(m_nBmihAlloc == heapAlloc) {
                ::GlobalUnlock(m_hGlobal);
                ::GlobalFree(m_hGlobal);
        }
        if(m_nImageAlloc == crtAlloc) delete [] m_lpImage;
        if(m_hPalette != NULL) ::DeleteObject(m_hPalette);
        if(m_hBitmap != NULL) ::DeleteObject(m_hBitmap);
        m_nBmihAlloc = m_nImageAlloc = noAlloc;
        m_hGlobal = NULL;
        m_lpBMIH = NULL;
        m_lpImage = NULL;
        m_lpvColorTable = NULL;
        m_nColorTableEntries = 0;
        m_dwSizeImage = 0;
        m_lpvFile = NULL;
        m_hMap = NULL;
        m_hFile = NULL;
        m_hBitmap = NULL;
        m_hPalette = NULL;
}
```

```
void CDib::DetachMapFile()
{
       if(m_hFile == NULL) return;
       ::UnmapViewOfFile(m_lpvFile);
       ::CloseHandle(m_hMap);
       ::CloseHandle(m_hFile);
       m_hFile = NULL;
}
```

Listing 15: The `CDib` class declaration (header file) and implementation (source file).

Here's a rundown of the `CDib` member functions, starting with the constructors and the destructor:

- **Default constructor**: You'll use the default constructor in preparation for loading a DIB from a file or for attaching to a DIB in memory. The default constructor creates an empty DIB object.
- **DIB section constructor**: If you need a DIB section that is created by the `CreateDIBSection()` function, use this constructor. Its parameters determine DIB size and number of colors. The constructor allocates info header memory but not image memory. You can also use this constructor if you need to allocate your own image memory.

Parameter	Description
`size`	`CSize` object that contains the width and height of the DIB.
`nBitCount`	Bits per pixel; should be 1, 4, 8, 16, 24, or 32.

Table 1.

- **Destructor**: The `CDib` destructor frees all allocated DIB memory.
- **`AttachMapFile()`**: This function opens a memory-mapped file in read mode and attaches it to the `CDib` object. The return is immediate because the file isn't actually read into memory until it is used. When you access the DIB, however, a delay might occur as the file is paged in. The `AttachMapFile()` function releases existing allocated memory and closes any previously attached memory-mapped file.

Parameter	Description
`strPathname`	Pathname of the file to be mapped.
`bShare`	Flag that is `TRUE` if the file is to be opened in share mode; the default value is `FALSE`.
Return value	`TRUE` if successful.

Table 2

- **`AttachMemory()`**: This function associates an existing `CDib` object with a DIB in memory. This memory could be in the program's resources, or it could be clipboard or OLE data object memory. Memory might have been allocated from the CRT heap with the new operator, or it might have been allocated from the Windows heap with `GlobalAlloc()`.

Parameter	Description
`lpvMem`	Address of the memory to be attached.
`bMustDelete`	Flag that is `TRUE` if the `CDib` class is responsible for deleting this memory; the default value is `FALSE`.
`hGlobal`	If memory was obtained with a call to the Win32 `GlobalAlloc()` function, the `CDib` object needs to keep the handle in order to free it later, assuming that `bMustDelete` was set to `TRUE`.
Return value	`TRUE` if successful.

Table 3.

- **`Compress()`**: This function regenerates the DIB as a compressed or an uncompressed DIB. Internally, it converts the existing DIB to a GDI bitmap and then makes a new compressed or an uncompressed DIB. Compression is supported only for 4-bpp and 8-bpp DIBs. You can't compress a DIB section.

Parameter	Description
`pDC`	Pointer to the display device context.
`bCompress`	`TRUE` (default) to compress the DIB; `FALSE` to uncompress it.
Return value	`TRUE` if successful.

Table 4.

- **`CopyToMapFile()`**: This function creates a new memory-mapped file and copies the existing `CDib` data to the file's memory, releasing any previously allocated memory and closing any existing memory-mapped file. The data isn't actually written to disk until the new file is closed, but that happens when the `CDib` object is reused or destroyed.

Parameter	Description
`strPathname`	Pathname of the file to be mapped.
Return value	`TRUE` if successful.

Table 5.

- **`CreateBitmap()`**: This function creates a GDI bitmap from an existing DIB and is called by the `Compress()` function. Don't confuse this function with `CreateSection()`, which generates a DIB and stores the handle.

Parameter	Description
`pDC`	Pointer to the display or printer device context.
Return value	Handle to a GDI bitmap, `NULL` if unsuccessful. This handle is not stored as a public data member.

Table 6.

- **`CreateSection()`**: This function creates a DIB section by calling the Win32 `CreateDIBSection()` function. The image memory will be uninitialized.

Parameter	Description
`pDC`	Pointer to the display or printer device context.
Return value	Handle to a GDI bitmap, `NULL` if unsuccessful. This handle is also stored as a public data member.

Table 7.

- **`Draw()`**: This function outputs the `CDib` object to the display (or to the printer) with a call to the Win32 `StretchDIBits()` function. The bitmap will be stretched as necessary to fit the specified rectangle.

Parameter	Description
`pDC`	Pointer to the display or printer device context that will receive the DIB image.
`origin`	`CPoint` object that holds the logical coordinates at which the DIB will be displayed.
`size`	`CSize` object that represents the display rectangle's width and height in logical units.
Return value	TRUE if successful.

Table 8.

- **`Empty()`**: This function empties the DIB, freeing allocated memory and closing the map file if necessary.
- **`GetDimensions()`**: This function returns the width and height of a DIB in pixels.

Parameter	Description
Return value	`CSize` object

Table 9.

- **`GetSizeHeader()`**: This function returns the number of bytes in the info header and color table combined.

Parameter	Description
Return value	32-bit integer

Table 10.

- **`GetSizeImage()`**: This function returns the number of bytes in the DIB image (excluding the info header and the color table).

Parameter	Description
Return value	32-bit integer

Table 11.

- **`MakePalette()`**: If the color table exists, this function reads it and creates a Windows palette. The `HPALETTE` handle is stored in a data member.

Parameter	Description
Return value	`TRUE` if successful

Table 12.

- **`Read()`**: This function reads a DIB from a file into the `CDib` object. The file must have been successfully opened. If the file is a BMP file, reading starts from the beginning of the file. If the file is a document, reading starts from the current file pointer.

Parameter	Description
`pFile`	Pointer to a `CFile` object; the corresponding disk file contains the DIB.
Return value	`TRUE` if successful.

Table 13.

- **`ReadSection()`**: This function reads the info header from a BMP file, calls `CreateDIBSection()` to allocate image memory, and then reads the image bits from the file into that memory. Use this function if you want to read a DIB from disk and then edit it by calling GDI functions. You can write the DIB back to disk with Write or `CopyToMapFile()`.

Parameter	Description
`pFile`	Pointer to a `CFile` object; the corresponding disk file contains the DIB.
`pDC`	Pointer to the display or printer device context.
Return value	`TRUE` if successful.

Table 14.

- **`Serialize()`**: The `CDib::Serialize` function, which overrides the MFC `CObject::Serialize` function, calls the `Read()` and `Write()` member functions. See the Microsoft Foundation Classes and Templates section of the online help for a description of the parameters.
- **`SetSystemPalette()`**: If you have a 16-bpp, 24-bpp, or 32-bpp DIB that doesn't have a color table, you can call this function to create for your `CDib` object a logical palette that matches the palette returned by the `CreateHalftonePalette()` function. If your program is running on a 256-color palletized display and you don't call `SetSystemPalette()`, you'll have no palette at all, and only the 20 standard Windows colors will appear in your DIB.

Parameter	Description
`pDC`	Pointer to the display context.
Return value	`TRUE` if successful.

Table 15.

- **`UsePalette()`**: This function selects the `CDib` object's logical palette into the device context and then realizes the palette. The `Draw()` member function calls `UsePalette()` prior to painting the DIB.

Parameter	Description
`pDC`	Pointer to the display device context for realization.
`bBackground`	If this flag is `FALSE` (the default value) and the application is running in the foreground, Windows realizes the palette as the foreground palette (copies as many colors as possible into the system palette). If this flag is `TRUE`, Windows realizes the palette as a background palette (maps the logical palette to the system palette as best it can).
Return value	Number of entries in the logical palette mapped to the system palette. If the function fails, the return value is `GDI_ERROR`.

Table 16.

- **`Write()`**: This function writes a DIB from the `CDib` object to a file. The file must have been successfully opened or created.

Parameter	Description
`pFile`	Pointer to a `CFile` object; the DIB will be written to the corresponding disk file.
Return value	`TRUE` if successful.

Table 17.

For your convenience, four public data members give you access to the DIB memory and to the DIB section handle. These members should give you a clue about the structure of a `CDib` object. A `CDib` is just a bunch of pointers to heap memory. That memory might be owned by the DIB or by someone else. Additional private data members determine whether the `CDib` class frees the memory.

DIB Display Performance

Optimized DIB processing is now a major feature of Windows. Modern video cards have frame buffers that conform to the standard DIB image format. If you have one of these cards, your programs can take advantage of the new Windows DIB engine, which speeds up the process of drawing directly from DIBs. If you're still running in VGA mode, however, you're out of luck; your programs will still work, but not as fast.
If you're running Windows in 256-color mode, your 8-bpp bitmaps will be drawn very quickly, either with `StretchBlt()` or with `StretchDIBits()`. If, however, you are displaying 16-bpp or 24-bpp bitmaps, those drawing functions will be too slow. Your bitmaps will appear more quickly in this situation if you create a separate 8-bbp GDI bitmap and then call `StretchBlt()`. Of course, you must be careful to realize the correct palette prior to creating the bitmap and prior to drawing it.

Here's some code that you might insert just after loading your `CDib` object from a BMP file:

```
// m_hBitmap is a data member of type HBITMAP
// m_dcMem is a memory device context object of class CDC
m_pDib->UsePalette(&dc);
m_hBitmap = m_pDib->CreateBitmap(&dc); // could be slow
::SelectObject(m_dcMem.GetSafeHdc(), m_hBitmap);
Here is the code that you use in place of CDib::Draw in your view's OnDraw member
function:
m_pDib->UsePalette(pDC); // could be in palette msg handler
CSize sizeDib = m_pDib->GetDimensions();
pDC->StretchBlt(0, 0, sizeDib.cx, sizeDib.cy, &m_dcMem, 0, 0, sizeToDraw.cx,
sizeToDraw.cy, SRCCOPY);
```

Don't forget to call `DeleteObject()` for `m_hBitmap` when you're done with it.

The MYMFC26C Example

Now you'll put the `CDib` class to work in an application. The MYMFC26C program displays two DIBs, one from a resource and the other loaded from a BMP file that you select at runtime. The program manages the system palette and displays the DIBs correctly on the printer. Compare the MYMFC26C code with the GDI bitmap code in MYMFC26A. Notice that you're not dealing with a memory device context and all the GDI selection rules! Following are the steps to build MYMFC26C.

Run AppWizard to produce \mfcproject\mymfc26C. Accept all the defaults but two: select **Single Document** and select the `CScrollView` view base class for `CMymfc26CView` in step 6. The options and the default class names are shown here.

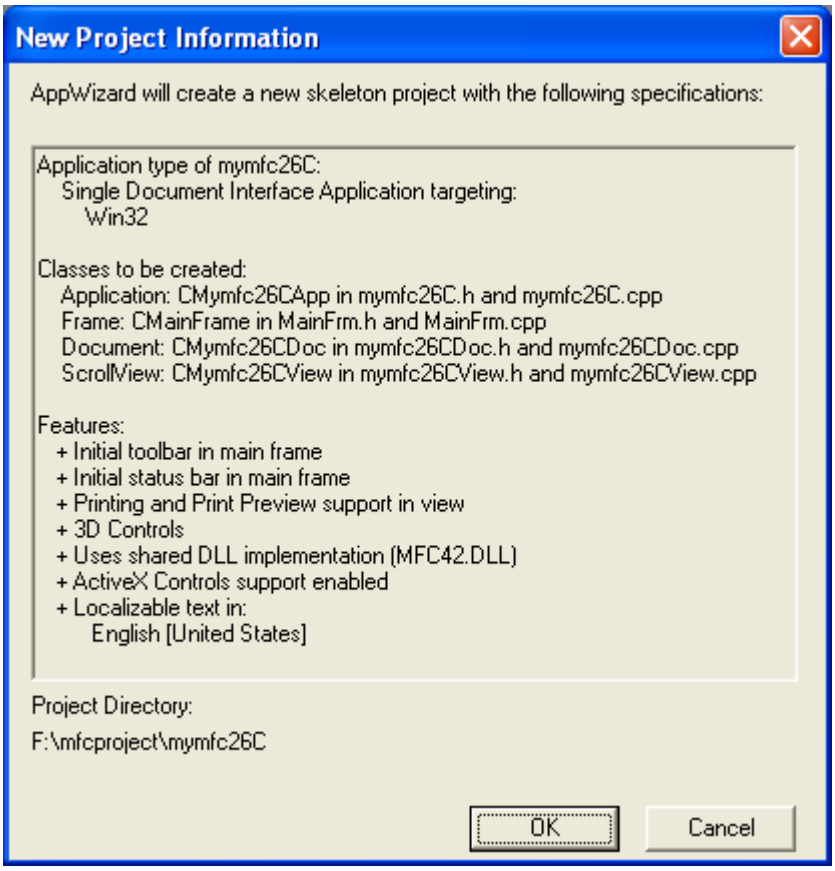

Figure 16: MYMFC26C project summary.

Import the **Soap Bubbles** bitmap. Choose **Resource** from Visual C++'s **Insert** menu. Import **Soap Bubbles.bmp** from the **\WINDOWS** directory.

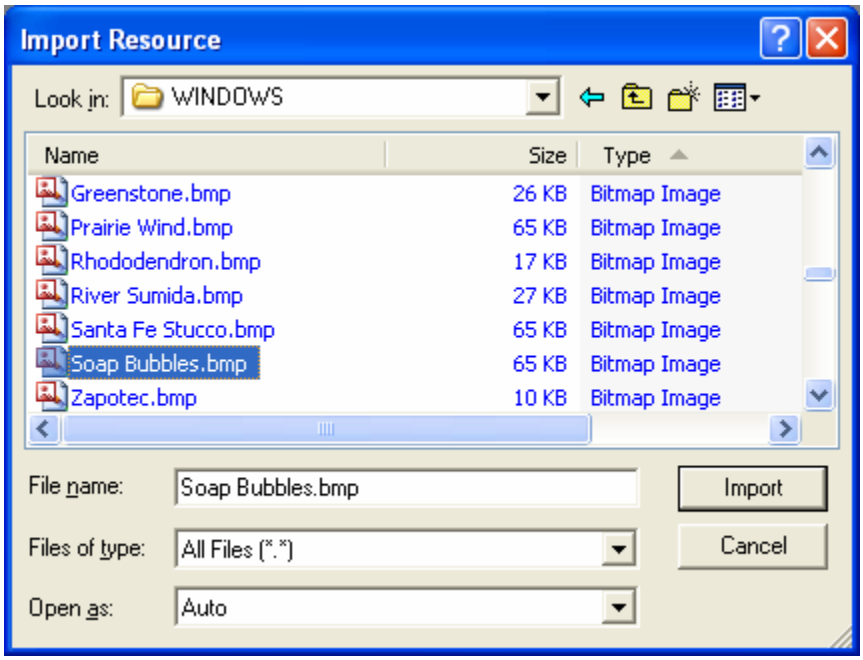

Figure 17: Importing **Soap Bubbles.bmp** into MYMFC26C project.

Visual C++ will copy this bitmap file into your project's **\res** subdirectory. Assign `IDB_SOAPBUBBLE` as the ID, and save the changes.

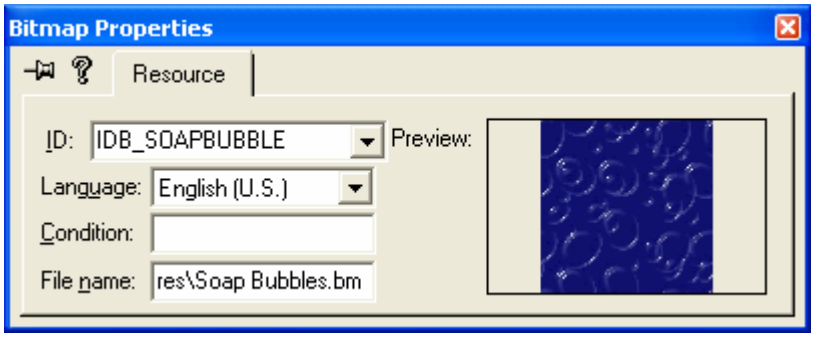

Figure 18: Modifying bitmap properties.

Integrate the `CDib` class with this project. If you've created this project from scratch, copy the **cdib.h** and **cdib.cpp** files to the **\mfcproject\mymfc26C** directory. Simply copying the files to disk isn't enough; you must also add the `CDib` files to the project. Choose `Add To Project` from Visual C++'s `Project` menu, and then choose `Files`. Select **cdib.h** and **cdib.cpp**, and click the **OK** button. If you now switch to ClassView in the Workspace window, you will see the class `CDib` and all of its member variables and functions.

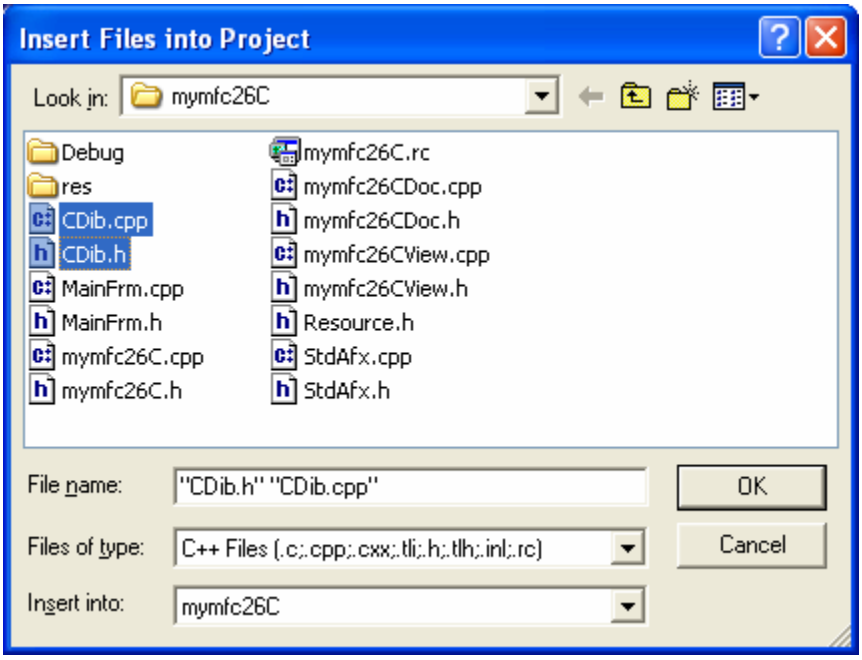

Figure 19: Adding header and source files (class) to the project.

Add two private `CDib` data members to the class `CMymfc26CView`. In the ClassView window, right-click the `CMymfc26CView` class. Choose **Add Member Variable** from the resulting pop-up menu, and then add the `m_dibResource` member as shown in the following illustration.

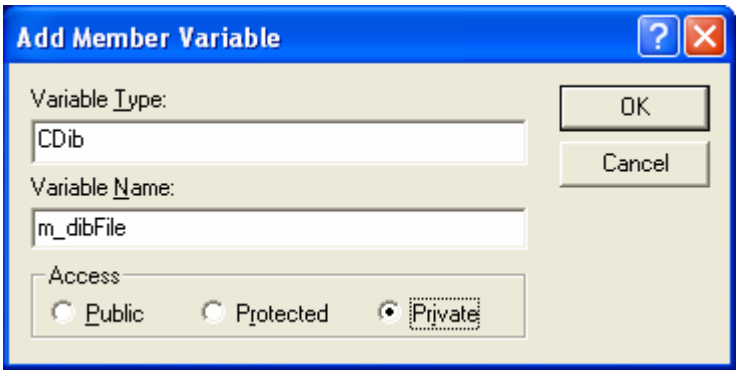

Figure 20: Adding private `CDib` type data members to the class `CMymfc26CView`.

Add `m_dibFile` in the same way. The result should be two data members at the bottom of the header file as shown below:

```
        CDib m_dibFile;
        CDib m_dibResource;
```

```
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
private:
    CDib m_dibFile;
    CDib m_dibResource;
};
```

Listing 16.

ClassView also adds the `#include` statement at the top of the **mymfc26CView.h** file:

```
#include "cdib.h"   // Added by ClassView
```

```
#if !defined(AFX_MYMFC26CVIEW_H__2ED2E5F3_
#define AFX_MYMFC26CVIEW_H__2ED2E5F3_99A5_

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

Listing 17.

Edit the `OnInitialUpdate()` member function in **mymfc26CView.cpp**. This function sets the mapping mode to `MM_HIMETRIC` and loads the `m_dibResource` object directly from the `IDB_REDBLOCKS` resource. Note that we're not calling `LoadBitmap()` to load a GDI bitmap as we did in MYMFC26A. The `CDib::AttachMemory` function connects the object to the resource in your EXE file. Add the following code:

```
void CMymfc26CView::OnInitialUpdate()
{
    CScrollView::OnInitialUpdate();
    CSize sizeTotal(30000, 40000); // 30-by-40 cm
    CSize sizeLine = CSize(sizeTotal.cx / 100, sizeTotal.cy / 100);
    SetScrollSizes(MM_HIMETRIC, sizeTotal, sizeTotal, sizeLine);

    LPVOID lpvResource = (LPVOID) ::LoadResource(NULL,
        ::FindResource(NULL, MAKEINTRESOURCE(IDB_SOAPBUBBLE), RT_BITMAP));
    m_dibResource.AttachMemory(lpvResource); // no need for
                                             //  ::LockResource
    CClientDC dc(this);
    TRACE("bits per pixel = %d\n", dc.GetDeviceCaps(BITSPIXEL));
}
```

```
void CMymfc26CView::OnInitialUpdate()
{
    CScrollView::OnInitialUpdate();
    CSize sizeTotal(30000, 40000); // 30-by-40 cm
    CSize sizeLine = CSize(sizeTotal.cx / 100, sizeTotal.cy / 100);
    SetScrollSizes(MM_HIMETRIC, sizeTotal, sizeTotal, sizeLine);

    LPVOID lpvResource = (LPVOID) ::LoadResource(NULL,
        ::FindResource(NULL, MAKEINTRESOURCE(IDB_SOAPBUBBLE), RT_BITMAP));
    m_dibResource.AttachMemory(lpvResource); // no need for
                                             //  ::LockResource
    CClientDC dc(this);
    TRACE("bits per pixel = %d\n", dc.GetDeviceCaps(BITSPIXEL));
}
```

Listing 18.

Edit the `OnDraw()` member function in the file **mymfc26CView.cpp**. This code calls `CDib::Draw` for each of the DIBs. The `UsePalette()` calls should really be made by message handlers for the `WM_QUERYNEWPALETTE` and `WM_PALETTECHANGED` messages. These messages are hard to deal with because they don't go to the view directly, so we'll take a shortcut. Add the following code:

```
void CMymfc26CView::OnDraw(CDC* pDC)
{
    BeginWaitCursor();
    m_dibResource.UsePalette(pDC); // should be in palette
    m_dibFile.UsePalette(pDC);     //  message handlers, not here
    pDC->TextOut(0, 0,
        "Press the left mouse button here to load a file.");
    CSize sizeResourceDib = m_dibResource.GetDimensions();
```

```
        sizeResourceDib.cx *= 30;
        sizeResourceDib.cy *= -30;
        m_dibResource.Draw(pDC, CPoint(0, -800), sizeResourceDib);
        CSize sizeFileDib = m_dibFile.GetDimensions();
        sizeFileDib.cx *= 30;
        sizeFileDib.cy *= -30;
        m_dibFile.Draw(pDC, CPoint(1800, -800), sizeFileDib);
        EndWaitCursor();
    }
```

```
// CMymfc26CView drawing

void CMymfc26CView::OnDraw(CDC* pDC)
{
    BeginWaitCursor();
    m_dibResource.UsePalette(pDC); // should be in palette
    m_dibFile.UsePalette(pDC);     //  message handlers, not here
    pDC->TextOut(0, 0,
        "Press the left mouse button here to load a file.");
    CSize sizeResourceDib = m_dibResource.GetDimensions();
    sizeResourceDib.cx *= 30;
    sizeResourceDib.cy *= -30;
    m_dibResource.Draw(pDC, CPoint(0, -800), sizeResourceDib);
    CSize sizeFileDib = m_dibFile.GetDimensions();
    sizeFileDib.cx *= 30;
    sizeFileDib.cy *= -30;
    m_dibFile.Draw(pDC, CPoint(1800, -800), sizeFileDib);
    EndWaitCursor();
}
```

Listing 19.

Map the `WM_LBUTTONDOWN` message in the C`Mymfc26CView` class. Edit the file **mymfc26CView.cpp**. `OnLButtonDown()` contains code to read a DIB in two different ways. If you leave the `MEMORY_MAPPED_FILES` definition intact, the `AttachMapFile()` code is activated to read a memory-mapped file. If you comment out the first line, the `Read()` call is activated. The `SetSystemPalette()` call is there for DIBs that don't have a color table.

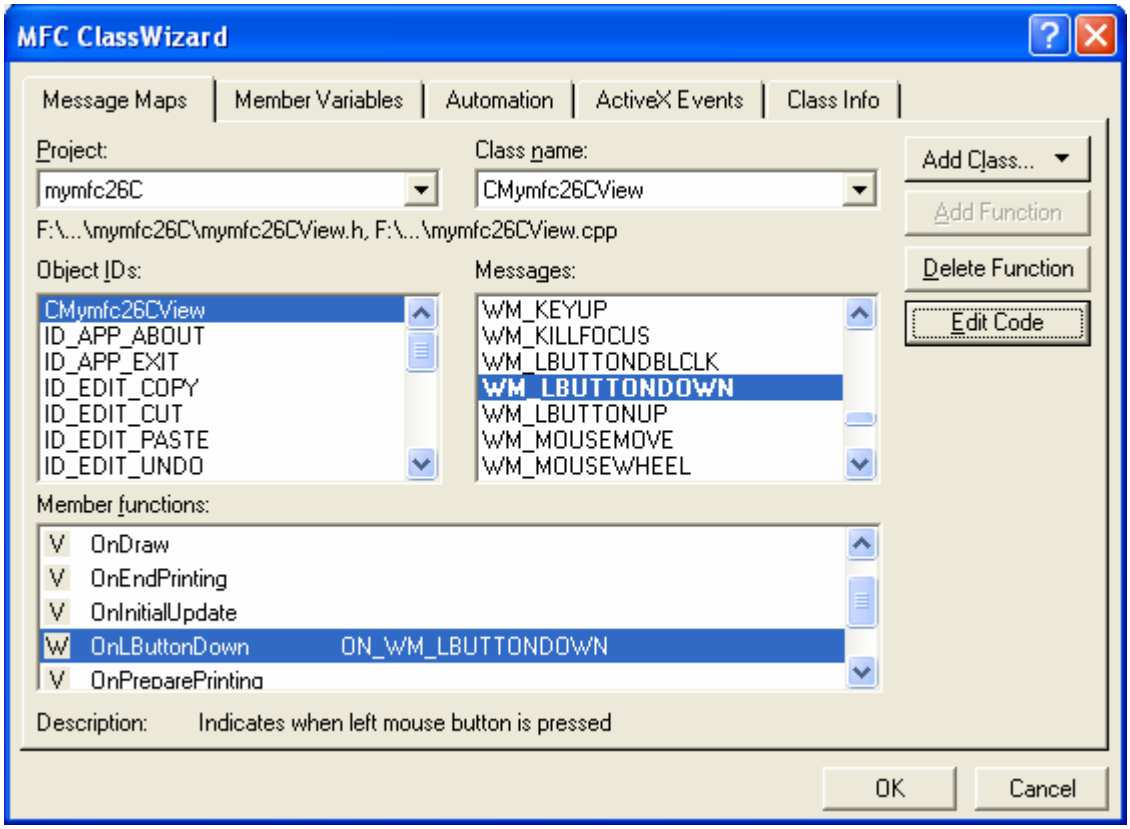

Figure 21: Mapping the `WM_LBUTTONDOWN` message in the `CMymfc26CView` class for left mouse button click.

Then, add the following code:

```
#define MEMORY_MAPPED_FILES

void CMymfc26CView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CFileDialog dlg(TRUE, "bmp", "*.bmp");
    if (dlg.DoModal() != IDOK) {
        return;
    }

#ifdef MEMORY_MAPPED_FILES
    if (m_dibFile.AttachMapFile(dlg.GetPathName(),
            TRUE) == TRUE) { // share
        Invalidate();
    }
 #else
    CFile file;
    file.Open(dlg.GetPathName(), CFile::modeRead);
    if (m_dibFile.Read(&file) == TRUE) {
        Invalidate();
    }
#endif // MEMORY_MAPPED_FILES
    CClientDC dc(this);
    m_dibFile.SetSystemPalette(&dc);
}
```

```
// CMymfc26CView message handlers

#define MEMORY_MAPPED_FILES

void CMymfc26CView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CFileDialog dlg(TRUE, "bmp", "*.bmp");
    if (dlg.DoModal() != IDOK) {
        return;
    }

#ifdef MEMORY_MAPPED_FILES
    if (m_dibFile.AttachMapFile(dlg.GetPathName(),
            TRUE) == TRUE) { // share
        Invalidate();
    }
 #else
    CFile file;
    file.Open(dlg.GetPathName(), CFile::modeRead);
    if (m_dibFile.Read(&file) == TRUE) {
        Invalidate();
    }
#endif // MEMORY_MAPPED_FILES
    CClientDC dc(this);
    m_dibFile.SetSystemPalette(&dc);
}
```

Listing 20.

Build and run the application. Try some other BMP files if you have them. Note that **Soap Bubbles** is a 16-color DIB that uses standard colors, which are always included in the system palette.

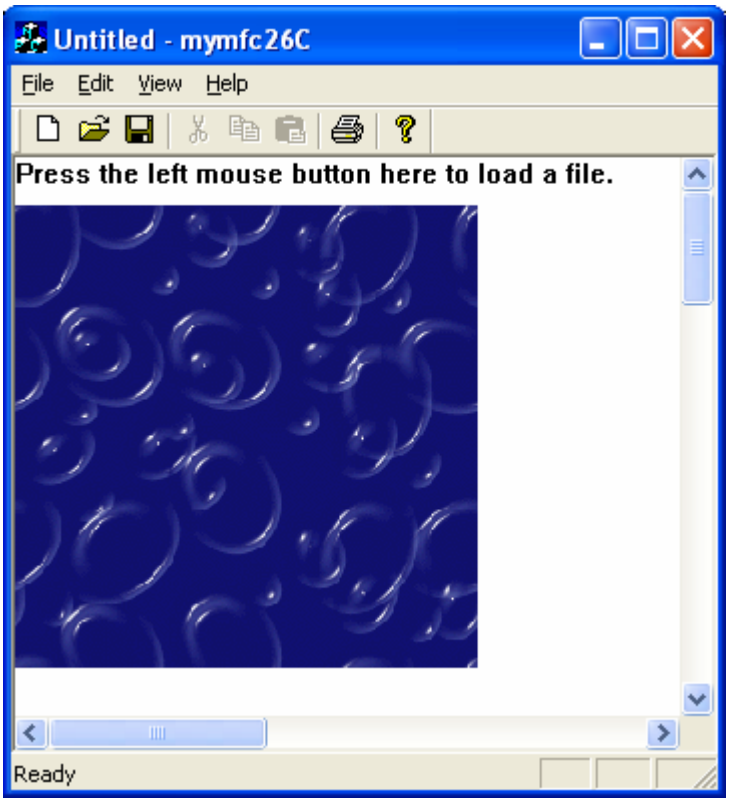

Figure 22: MYMFC26C program output.

Going Further with DIBs

Each new version of Windows offers more DIB programming choices. Both Windows 95 and Microsoft Windows NT 4.0 provide the `LoadImage()` and `DrawDibDraw()` functions, which are useful alternatives to the DIB functions already described. Experiment with these functions to see if they work well in your applications.

The `LoadImage()` Function

The `LoadImage()` function can read a bitmap directly from a disk file, returning a DIB section handle. It can even process OS/2 (already discontinued OS) format DIBs. Suppose you wanted to add an `ImageLoad()` member function to `CDib` that would work like `ReadSection()`. This is the code you would add to **cdib.cpp**:

```
BOOL CDib::ImageLoad(const char* lpszPathName, CDC* pDC)
{
    Empty();
    m_hBitmap = (HBITMAP) ::LoadImage(NULL, lpszPathName, IMAGE_BITMAP, 0, 0,
        LR_LOADFROMFILE | LR_CREATEDIBSECTION | LR_DEFAULTSIZE);
    DIBSECTION ds;
    VERIFY(::GetObject(m_hBitmap, sizeof(ds), &ds) == sizeof(ds));
    // Allocate memory for BITMAPINFOHEADER
    //  and biggest possible color table
    m_lpBMIH = (LPBITMAPINFOHEADER) new
        char[sizeof(BITMAPINFOHEADER) + 256 * sizeof(RGBQUAD)];
    memcpy(m_lpBMIH, &ds.dsBmih, sizeof(BITMAPINFOHEADER));
    TRACE("CDib::LoadImage, biClrUsed = %d, biClrImportant = %d\n",
        m_lpBMIH->biClrUsed, m_lpBMIH->biClrImportant);
    ComputeMetrics(); // sets m_lpvColorTable
    m_nBmihAlloc = crtAlloc;
    m_lpImage = (LPBYTE) ds.dsBm.bmBits;
    m_nImageAlloc = noAlloc;
    // Retrieve the DIB section's color table
    //  and make a palette from it
    CDC memdc;
    memdc.CreateCompatibleDC(pDC);
    ::SelectObject(memdc.GetSafeHdc(), m_hBitmap);
    UINT nColors = ::GetDIBColorTable(memdc.GetSafeHdc(), 0, 256, (RGBQUAD*)
m_lpvColorTable);
    if (nColors != 0) {
        ComputePaletteSize(m_lpBMIH->biBitCount);
        MakePalette();
    }
    // memdc deleted and bitmap deselected
    return TRUE;
}
```

Note that this function extracts and copies the `BITMAPINFOHEADER` structure and sets the values of the `CDib` pointer data members. You must do some work to extract the palette from the DIB section, but the Win32 `GetDIBColorTable()` function gets you started. It's interesting that `GetDIBColorTable()` can't tell you how many palette entries a particular DIB uses. If the DIB uses only 60 entries, for example, `GetDIBColorTable()` generates a 256-entry color table with the last 196 entries set to 0.

The `DrawDibDraw()` Function

Windows includes the Video for Windows (VFW) component, which is supported by Visual C++. The VFW `DrawDibDraw()` function is an alternative to `StretchDIBits()`. One advantage of `DrawDibDraw()` is its ability to use dithered colors. Another is its increased speed in drawing a DIB with a bpp value that does not match the current video mode. The main disadvantage is the need to link the VFW code into your process at runtime. Shown below is a `DrawDib()` member function for the `CDib` class that calls `DrawDibDraw()`:

```
BOOL CDib::DrawDib(CDC* pDC, CPoint origin, CSize size)
{
    if (m_lpBMIH == NULL) return FALSE;
    if (m_hPalette != NULL) {
```

```
            ::SelectPalette(pDC->GetSafeHdc(), m_hPalette, TRUE);
        }
        HDRAWDIB hdd = ::DrawDibOpen();
        CRect rect(origin, size);
        pDC->LPtoDP(rect); // Convert DIB's rectangle
                           //  to MM_TEXT coordinates
        rect -= pDC->GetViewportOrg();
        int nMapModeOld = pDC->SetMapMode(MM_TEXT);
        ::DrawDibDraw(hdd, pDC->GetSafeHdc(), rect.left, rect.top, rect.Width(),
    rect.Height(), m_lpBMIH, m_lpImage, 0, 0,
            m_lpBMIH->biWidth, m_lpBMIH->biHeight, 0);
        pDC->SetMapMode(nMapModeOld);
        VERIFY(::DrawDibClose(hdd));
        return TRUE;
    }
```

Note that `DrawDibDraw()` needs `MM_TEXT` coordinates and the `MM_TEXT` mapping mode. Thus, logical coordinates must be converted not to device coordinates but to pixels with the origin at the top left of the scrolling window. To use `DrawDibDraw()`, your program needs an `#include <vfw.h>` statement, and you must add **vfw32.lib** to the list of linker input files. `DrawDibDraw()` might assume the bitmap it draws is in read/write memory, a fact to keep in mind if you map the memory to the BMP file.

Putting Bitmaps on Pushbuttons

The MFC library makes it easy to display a bitmap (instead of text) on a pushbutton. If you were to program this from scratch, you would set the **Owner Draw** property for your button and then write a message handler in your dialog class that would paint a bitmap on the button control's window. If you use the MFC `CBitmapButton` class instead, you end up doing a lot less work, but you have to follow a kind of "cookbook" procedure. Don't worry too much about how it all works (but be glad that you don't have to write much code!).

To make a long story short, you lay out your dialog resource as usual with unique text captions for the buttons you designate for bitmaps. Next you add some bitmap resources to your project, and you identify those resources by name rather than by numeric ID. Finally you add some `CBitmapButton` data members to your dialog class, and you call the `AutoLoad()` member function for each one, which matches a bitmap name to a button caption. If the button caption is "`Copy`", you add two bitmaps: "`COPYU`" for the up state and "`COPYD`" for the down state. By the way, you must still set the button's **Owner Draw** property. This will all make more sense when you write a program.

If you look at the MFC source code for the `CBitmapButton` class, you'll see that the bitmap is an ordinary GDI bitmap painted with a `BitBlt()` call. Thus, you can't expect any palette support. That's not often a problem because bitmaps for buttons are usually 16-color bitmaps that depend on standard VGA colors.

The MYMFC26D Example

Here are the steps for building MYMFC26D:

Run AppWizard to produce \mfcproject\mymfc26D. Accept all the defaults but three: select **Single Document**, deselect **Printing And Print Preview**, and select **Context-Sensitive Help**.

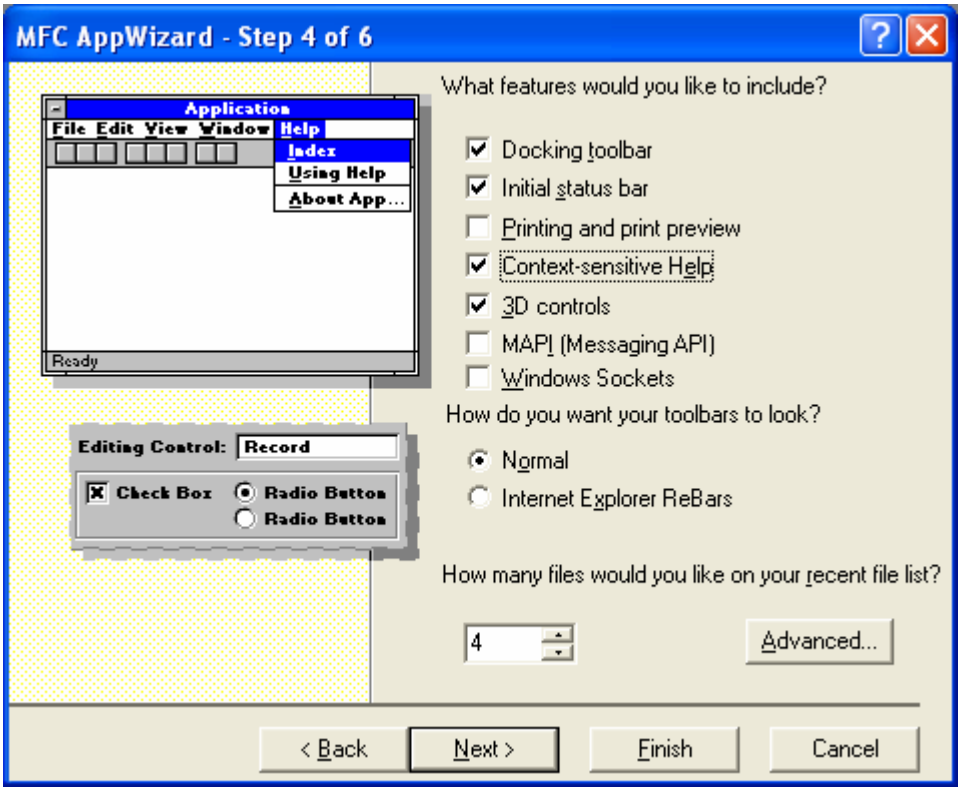

Figure 23: Step 4 of 6 AppWizard for MYMFC26D project, selecting the **Context-sensitive Help**.

The options and the default class names are shown in the illustration below.

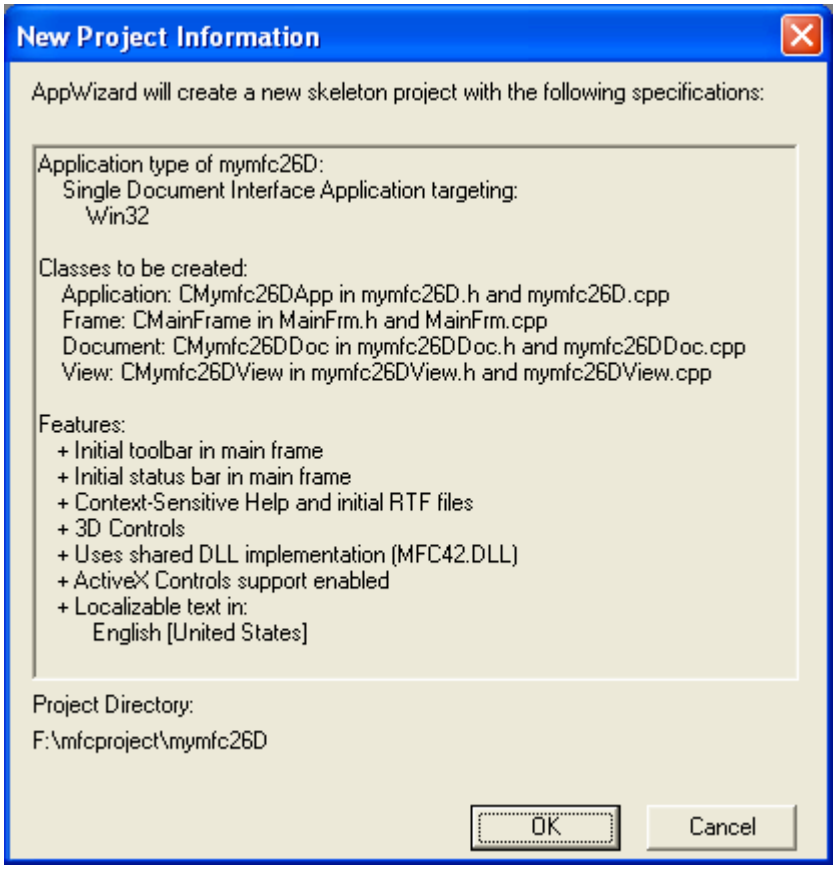

Figure 24: MYMFC26D project summary.

The **Context-Sensitive Help** option was selected for one reason only: it causes AppWizard to copy some bitmap files into your project's **hlp** subdirectory. These bitmaps are supposed to be bound into your project's help file.

Modify the project's `IDD_ABOUTBOX` dialog resource. It's too much hassle to create a new dialog resource for a few buttons, so we'll use the **About** dialog that AppWizard generates for every project. Add three pushbuttons with captions, as shown below, accepting the default IDs `IDC_BUTTON1`, `IDC_BUTTON2`, and `IDC_BUTTON3`. The size of the buttons isn't important because the framework adjusts the button size at runtime to match the bitmap size.

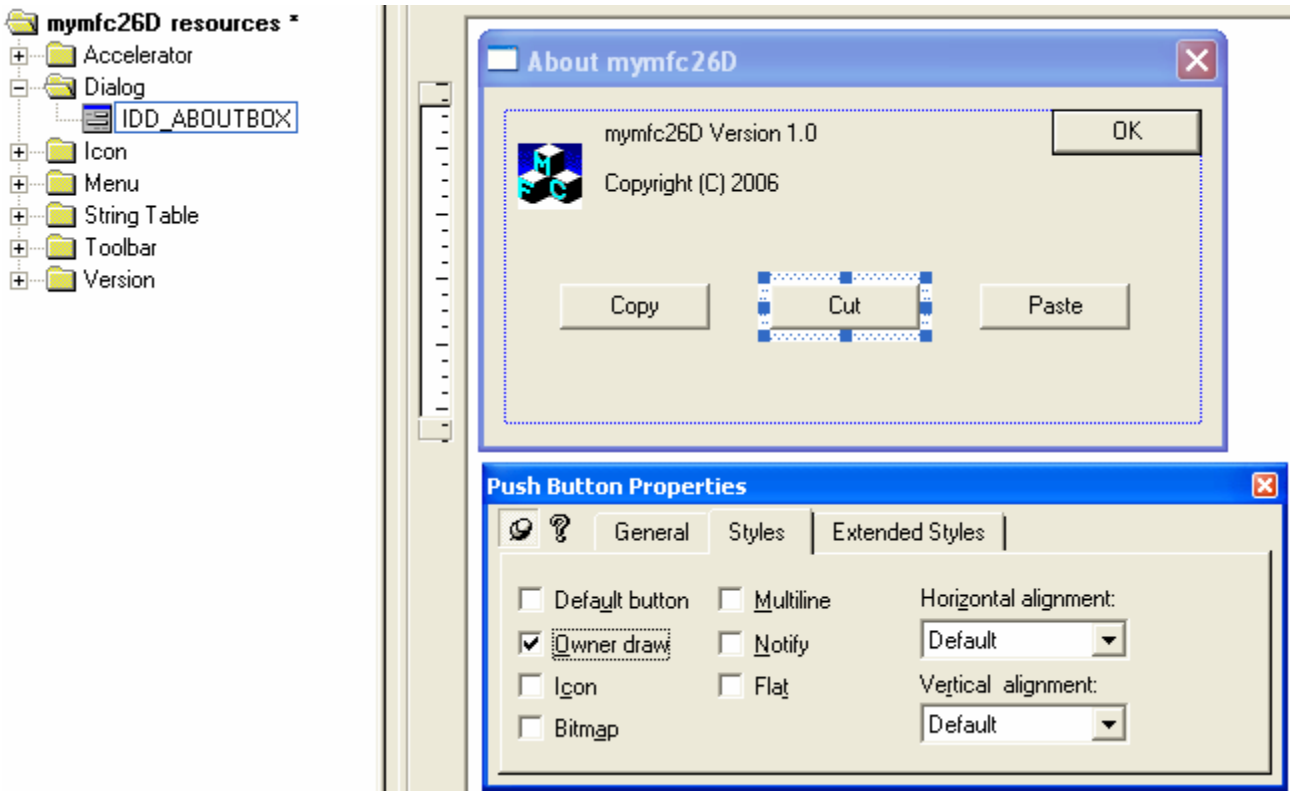

Figure 25: Using the **About** dialog, adding button controls and modifying their properties.

Select the **Owner Draw** property for all three buttons.

Import three bitmaps from the project's **hlp** subdirectory. Choose **Resource** from Visual C++'s **Insert** menu, select the **Bitmap** and then click the **Import** button. Start with **EditCopy.bmp**, as shown below.

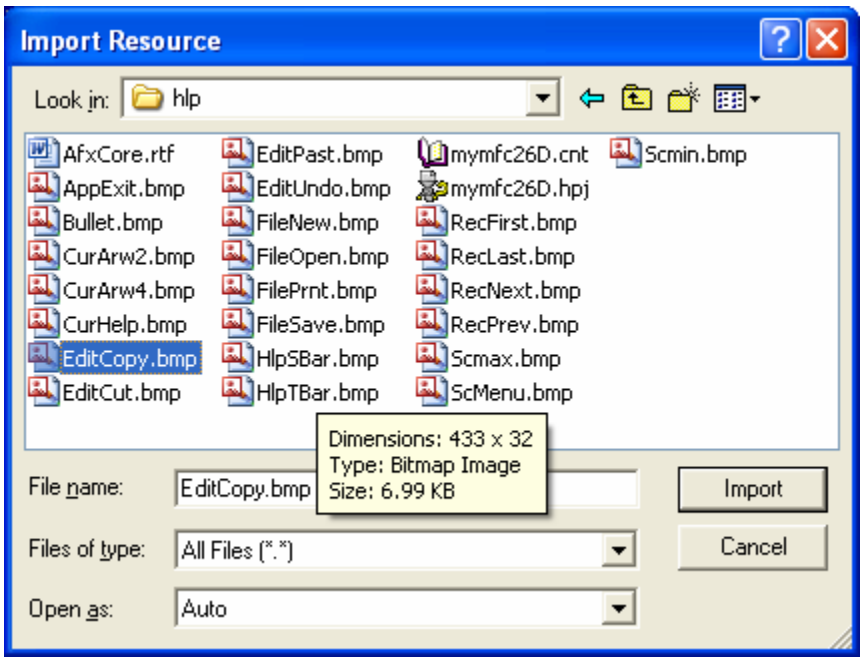

Figure 26: Importing three bitmaps from the project's **hlp** subdirectory.

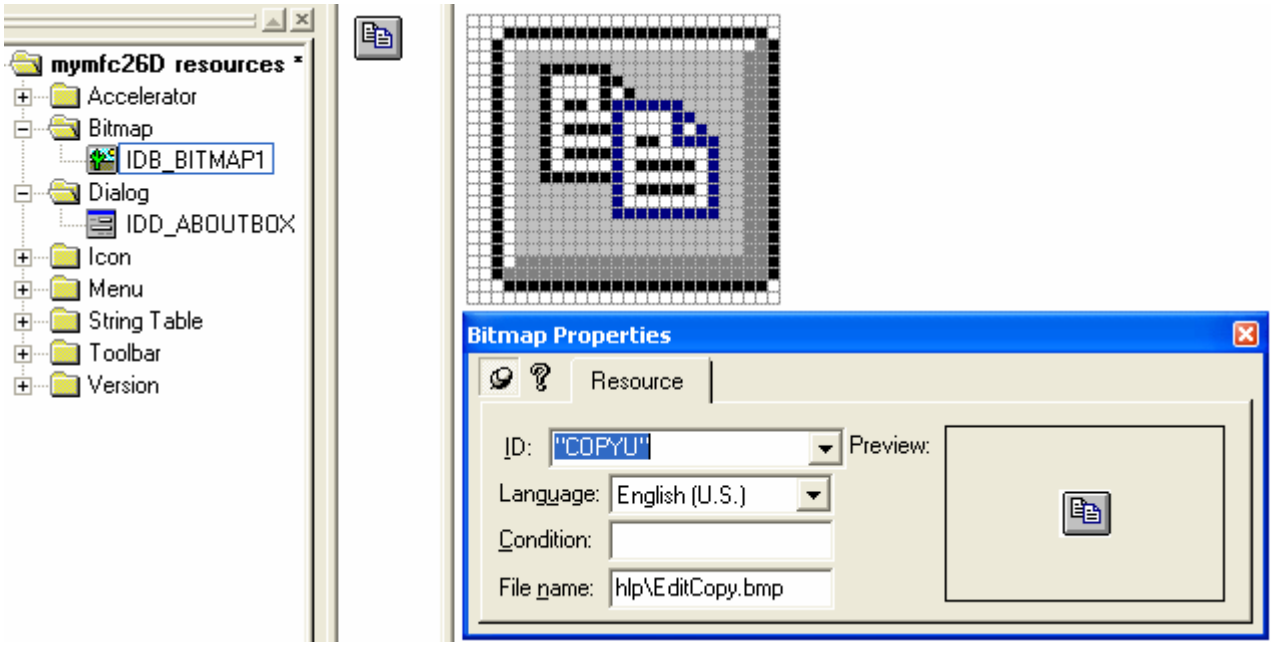

Figure 27: Modifying the properties of the imported bitmaps.

Assign the name "COPYU" as shown.

Be sure to use quotes around the name in order to identify the resource by name rather than by ID. This is now the bitmap for the button's up state. Close the bitmap window and, from the **ResourceView** window, use the clipboard (or drag and drop or **Edit**, **Copy**/**Paste** menu) to make a copy of the bitmap. Rename the copy "COPYD" (down state), and then edit this bitmap. Choose **Invert Colors** from the **Image** menu. There are other ways of making a variation of the up image, but inversion is the quickest.
Repeat the steps listed above for the **EditCut** and **EditPast** bitmaps. When you're finished, you should have the following bitmap resources in your project.

Resource Name	Original File	Invert Colors
"COPYU"	EditCopy.bmp	no
"COPYD"	EditCopy.bmp	yes
"CUTU"	EditCut.bmp	no
"CUTD"	EditCut.bmp	yes
"PASTEU"	EditPast.bmp	no
"PASTED"	EditPast.bmp	yes

Table 18.

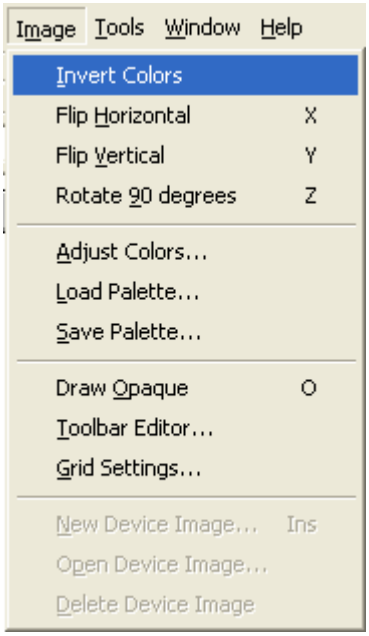

Figure 28: Using the image editor utility to invert color.

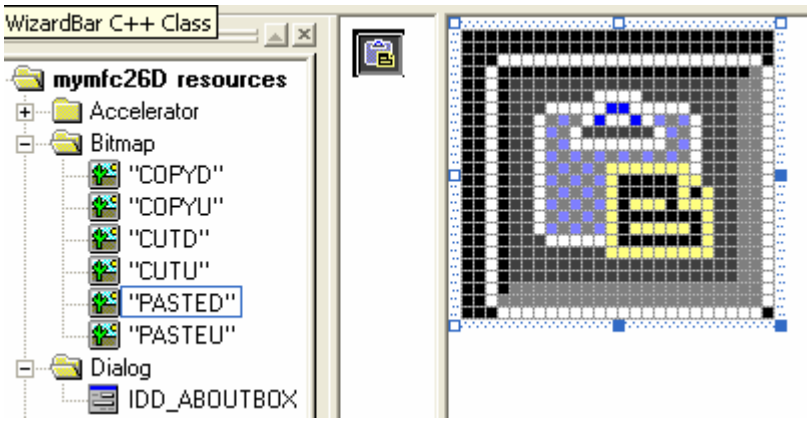

Figure 29: The inverted bitmap color.

Edit the code for the `CAboutDlg` class. Both the declaration and the implementation for this class are contained in the **mymfc26D.cpp** file. First add the three private data members shown here in the class declaration:

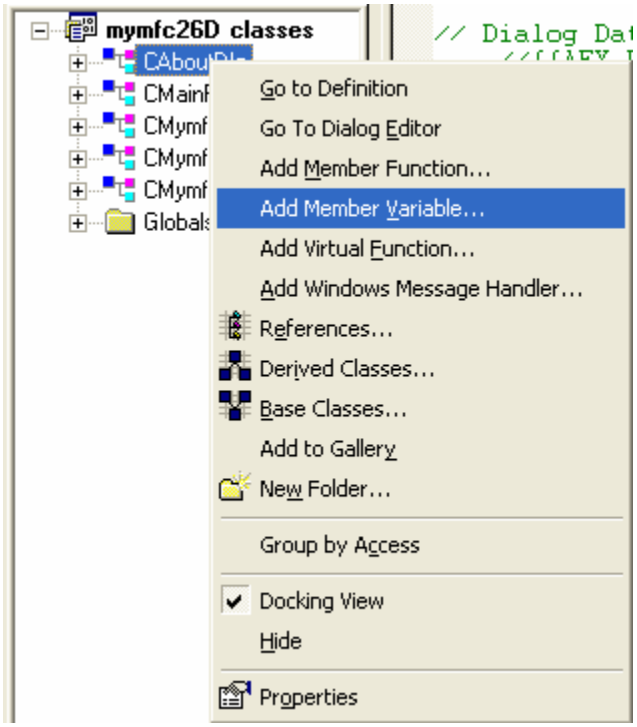

Figure 30: Adding three private data members to `CAboutDlg` class.

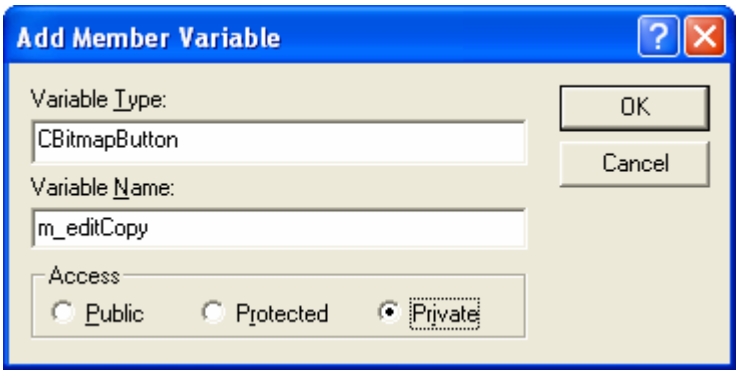

Figure 31: Entering the member variable type and name.

```
    CBitmapButton m_editCopy;
    CBitmapButton m_editCut;
    CBitmapButton m_editPaste;
```

```
    DECLARE_MESSAGE_MAP()
private:
    CBitmapButton m_editPaste;
    CBitmapButton m_editCut;
    CBitmapButton m_editCopy;
};
```

Listing 21.

Then you use ClassWizard to map the `WM_INITDIALOG` message in the dialog class. Be sure that the `CAboutDlg` class is selected. Then click the **Edit Code** button.

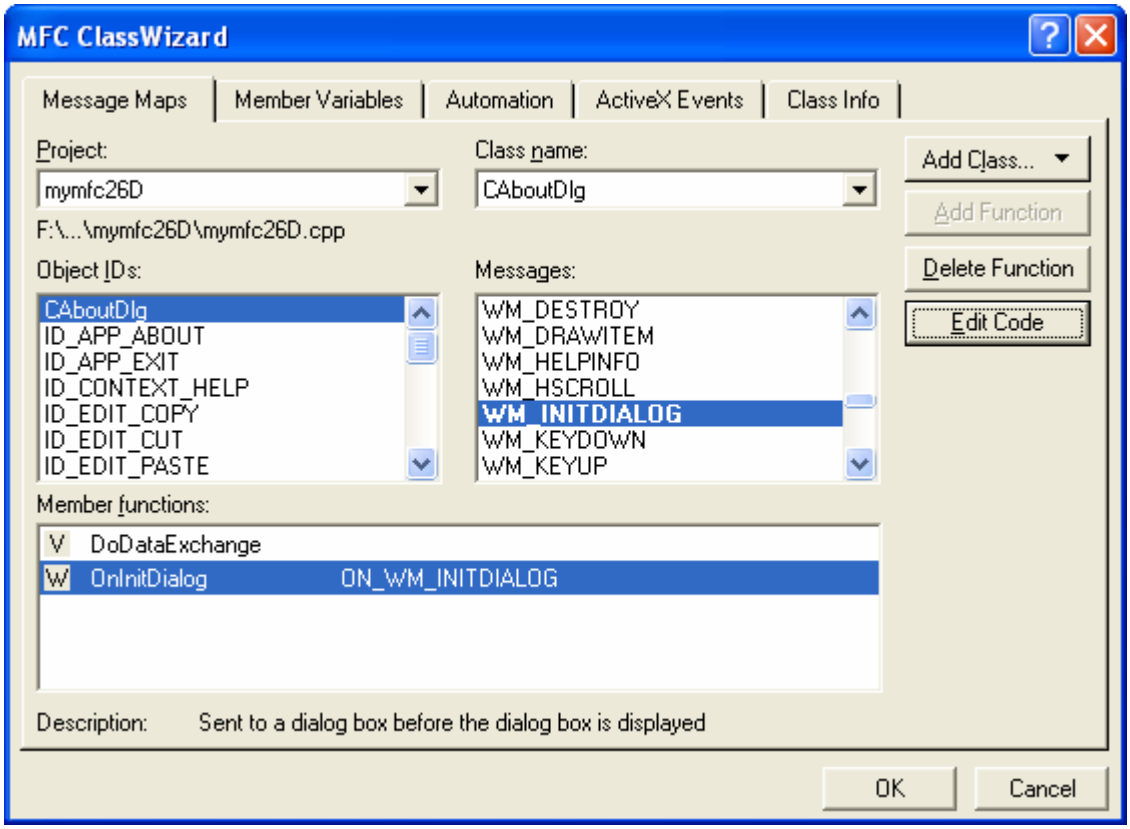

Figure 32: Using ClassWizard to map the `WM_INITDIALOG` message in the dialog class.

The message handler (actually a virtual function) is coded as follows:

```
BOOL CAboutDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    VERIFY(m_editCopy.AutoLoad(IDC_BUTTON1, this));
    VERIFY(m_editCut.AutoLoad(IDC_BUTTON2, this));
    VERIFY(m_editPaste.AutoLoad(IDC_BUTTON3, this));
    return TRUE;  // return TRUE unless you set the focus to a control
                  // EXCEPTION: OCX Property Pages should return FALSE
}
```

```
BOOL CAboutDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    VERIFY(m_editCopy.AutoLoad(IDC_BUTTON1, this));
    VERIFY(m_editCut.AutoLoad(IDC_BUTTON2, this));
    VERIFY(m_editPaste.AutoLoad(IDC_BUTTON3, this));
    return TRUE;  // return TRUE unless you set the focus to a control
                  // EXCEPTION: OCX Property Pages should return FALSE
}
```

Listing 22.

The `AutoLoad()` function connects each button with the two matching resources. The `VERIFY` macro is an MFC diagnostic aid that displays a message box if you didn't code the bitmap names correctly.

Edit the `OnDraw()` function in **mymfc26DView.cpp**. Replace the AppWizard-generated code with the following line:

```
pDC->TextOut(30, 30, "Choose About from the Help menu.");
```

```
// CMymfc26DView drawing

void CMymfc26DView::OnDraw(CDC* pDC)
{
    pDC->TextOut(30, 30, "Choose About from the Help menu.");
}
```

Listing 23.

Build and test the application. When the program starts, choose **About** from the **Help** menu and observe the button behavior. The image below shows the **CUT** button in the down state.

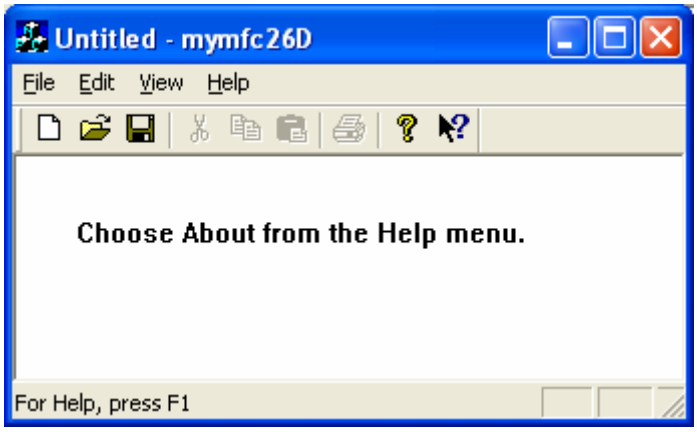

Figure 33: MYMFC26D program output.

Figure 34: MYMFC26D program output with bitmap.

Note that bitmap buttons send `BN_CLICKED` notification messages just as ordinary buttons do. ClassWizard can, of course, map those messages in your dialog class.

Going Further with Bitmap Buttons

You've seen bitmaps for the buttons' up and down states. The `CBitmapButton` class also supports bitmaps for the focused and disabled states. For the **Copy** button, the focused bitmap name would be "`COPYF`", and the disabled bitmap name would be "`COPYX`". If you want to test the disabled option, make a "`COPYX`" bitmap, possibly with a red line through it, and then add the following line to your program:

```
m_editCopy.EnableWindow(FALSE);
```

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.