替换下面的两行：

```
UINT nStart = rect.top / m_cyScreen;
UINT nEnd = min (m_nLinesTotal - 1,
    (rect.bottom + m_cyScreen - 1) / m_cyScreen);
```

然后打开一个大小为 10 KB 或 20 KB 的文件并进行上下滚动。很快您就会明白 OnDraw 不怕麻烦地将剪贴框转换为行号范围的原因。

HexDump 是在 OnPrint 中完成所有打印工作的。CHexView::OnPrint 调用 CHexView::PrintPageHeader 在页顶部打印页眉，并调用 CHexView::PrintPage 打印页的主体内容。OnBeginPrinting 通过下列工作实现设置：用为打印机设定的 10 点阵 Courier New 字体初始化 m_fontPrinter(注意在 CreatePointFont 中所传递的第三个参数中的打印机设备描述表指针)；用行间距初始化 m_cyPrinter；用基于页高度的每页的行数初始化 m_nLinesPerPage；用在打印页上居中打印行所要求的 x 缩进值初始化 m_cxOffset；用每行文本的宽度初始化 m_cxWidth。PrintPage 根据当前页号和每页的行数计算出首尾行号。执行绘制的 for 循环与 OnDraw 中的 for 循环相似，不同之处只在于它在打印页上对齐文本的方式，以及对于输出它使用的是 m_fontPrinter 而不是 m_fontScreen。在打印(或打印预览)结束后，OnEndPrinting 通过删除由 OnBeginPrinting 创建的打印机字体来完成清理工作。

可以用 OnDraw 同时处理屏幕和打印机输出吗？当然。但是按现在这种方式编写的 HexDump 程序(图 13-7)更简单也更直接。MFC 程序员有时会错误地以为它们必须使用 OnDraw 来进行打印工作和屏幕更新。HexDump 不仅说明那样做并不是必须的，同时还提供了一个实用的应用程序，将打印工作和屏幕刷新分开了。

HexDoc.h

```
// HexDoc.h : interface of the CHexDoc class
//
/////////////////////////////////////////////////////////////////////////////

#if !defined(AFX_HEXDOC_H__3A83FDFE_A3E6_11D2_8E53_006008A82731__INCLUDED_)
#define AFX_HEXDOC_H__3A83FDFE_A3E6_11D2_8E53_006008A82731__INCLUDED_

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

class CHexDoc : public CDocument
{
protected: // create from serialization only
    CHexDoc();
    DECLARE_DYNCREATE(CHexDoc)
```

```
// Attributes
public:
// Operations
public:
    UINT GetBytes(UINT nIndex, UINT nCount, PVOID pBuffer);
    UINT GetDocumentLength();

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CHexDoc)
    public:
    virtual BOOL OnNewDocument();
    virtual void Serialize(CArchive& ar);
    virtual void DeleteContents();
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CHexDoc();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
    BYTE* m_pFileData;
    UINT m_nDocLength;
    //{{AFX_MSG(CHexDoc)
      // NOTE - the ClassWizard will add and remove member functions here.
      //    DO NOT EDIT what you see in these blocks of generated code !
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////

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

#endif
// !defined(AFX_HEXDOC_H__3A83FDFE_A3B6_11D2_8E53_006008A82731__INCLUDED_)
```

HexDoc.cpp

```
// HexDoc.cpp : implementation of the CHexDoc class
//
```

```
#include "stdafx.h"
#include "HexDump.h"

#include "HexDoc.h"

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

/////////////////////////////////////////////////////////////////////////////
// CHexDoc

IMPLEMENT_DYNCREATE(CHexDoc, CDocument)

BEGIN_MESSAGE_MAP(CHexDoc, CDocument)
    //{{AFX_MSG_MAP(CHexDoc)
        // NOTE - the ClassWizard will add and remove mapping macros here.
        //    DO NOT EDIT what you see in these blocks of generated code!
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CHexDoc construction/destruction

CHexDoc::CHexDoc()
{
    m_nDocLength = 0;
    m_pFileData = NULL;
}

CHexDoc::~CHexDoc()
{
}

BOOL CHexDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;
    return TRUE;
}
/////////////////////////////////////////////////////////////////////////////
// CHexDoc serialization

void CHexDoc::Serialize(CArchive& ar)
{
    if (ar.IsLoading())
        CFile* pFile = ar.GetFile();
        m_nDocLength = (UINT) pFile->GetLength();
```

```
        //
        // Allocate a buffer for the file data.
        //
        try {
            m_pFileData = new BYTE[m_nDocLength];
        }
        catch (CMemoryException* e) {
            m_nDocLength = 0;
            throw e;
        }

        //
        // Read the file data into the buffer.
        //
        try {
            pFile->Read(m_pFileData, m_nDocLength);
        }
        catch (CFileException* e) {
            delete[] m_pFileData;
            m_pFileData = NULL;
            m_nDocLength = 0;
            throw e;
        }
    }
}

/////////////////////////////////////////////////////////////////////////////
// CHexDoc diagnostics

#ifdef _DEBUG
void CHexDoc::AssertValid() const
{
    CDocument::AssertValid();
}

void CHexDoc::Dump(CDumpContext& dc) const
{
    CDocument::Dump(dc);
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CHexDoc commands

void CHexDoc::DeleteContents()
{
    CDocument::DeleteContents();
```

```
    if (m_pFileData != NULL)
        delete[] m_pFileData;
        m_pFileData = NULL;
        m_nDocLength = 0;
    }
}

UINT CHexDoc::GetBytes(UINT nIndex, UINT nCount, PVOID pBuffer)
{
    if (nIndex >= m_nDocLength)
        return 0;

    UINT nLength = nCount;
    if ((nIndex + nCount) > m_nDocLength)
        nLength = m_nDocLength - nIndex;

    ::CopyMemory (pBuffer, m_pFileData + nIndex, nLength);
    return nLength;
}

UINT CHexDoc::GetDocumentLength()
{
    return m_nDocLength;
}
```

HexView.h

```
// HexView.h : interface of the CHexView class
//
/////////////////////////////////////////////////////////////////////////////

#if !defined(
    AFX_HEXVIEW_H__3A83FE00_A3E6_11D2_8E53_006008A82731__INCLUDED_)
#define AFX_HEXVIEW_H__3A83FE00_A3E6_11D2_8E53_006008A82731__INCLUDED_

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

class CHexView : public CScrollView
{
protected: // create from serialization only
    CHexView();
    DECLARE_DYNCREATE(CHexView)

// Attributes
public:
    CHexDoc* GetDocument();

// Operations
```

```
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CHexView)
    public:
    virtual void OnDraw(CDC* pDC);  // overridden to draw this view
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    protected:
    virtual void OnInitialUpdate(); // called first time after construct
    virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
    virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
    virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
    virtual void OnPrint(CDC* pDC, CPrintInfo* pInfo);
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CHexView();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
    void FormatLine(CHexDoc* pDoc, UINT nLine, CString& string);
    void PrintPageHeader(CHexDoc* pDoc, CDC* pDC, UINT nPageNumber);
    void PrintPage(CHexDoc* pDoc, CDC* pDC, UINT nPageNumber);
    UINT m_cxWidth;
    UINT m_cxOffset;
    UINT m_nLinesPerPage;
    UINT m_nLinesTotal;
    UINT m_cyPrinter;
    UINT m_cyScreen;
    CFont m_fontPrinter;
    CFont m_fontScreen;
    //{{AFX_MSG(CHexView)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG  // debug version in HexView.cpp
inline CHexDoc* CHexView::GetDocument()
```

```
    : return (CHexDoc * )m_pDocument; }
#endif

/////////////////////////////////////////////////////////////////////////////

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

#endif
// !defined(AFX_HEXVIEW_H__3A83FE00_A3E6_11D2_8E53_006008A82731__INCLUDED_)
```

HexView.cpp

```
// HexView.cpp : implementation of the CHexView class
//

#include "stdafx.h"
#include "HexDump.h"

#include "HexDoc.h"
#include "HexView.h"

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

#define PRINTMARGIN 2

/////////////////////////////////////////////////////////////////////////////
// CHexView
IMPLEMENT_DYNCREATE(CHexView, CScrollView)

BEGIN_MESSAGE_MAP(CHexView, CScrollView)
    //{{AFX_MSG_MAP(CHexView)
    ON_WM_CREATE()
    //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CScrollView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CScrollView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CScrollView::OnFilePrintPreview)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CHexView construction/destruction

CHexView::CHexView()
```

```
}

CHexView::~CHexView()
{
}

BOOL CHexView::PreCreateWindow(CREATESTRUCT& cs)
{
    return CScrollView::PreCreateWindow(cs);
}

////////////////////////////////////////////////////////////////////////////
// CHexView drawing

void CHexView::OnDraw(CDC* pDC)
{
    CHexDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    if (m_nLinesTotal != 0)
    {
        CRect rect;
        pDC->GetClipBox(&rect);

        UINT nStart = rect.top / m_cyScreen;
        UINT nEnd = min(m_nLinesTotal - 1,
            (rect.bottom + m_cyScreen - 1) / m_cyScreen);

        CFont* pOldFont = pDC->SelectObject(&m_fontScreen);
        for (UINT i = nStart; i <= nEnd; i++)
        {
            CString string;
            FormatLine(pDoc, i, string);
            pDC->TextOut(2, (i * m_cyScreen) + 2, string);
        }
        pDC->SelectObject(pOldFont);
    }
}

void CHexView::OnInitialUpdate()
{
    CScrollView::OnInitialUpdate();

    UINT nDocLength = GetDocument()->GetDocumentLength();
    m_nLinesTotal = (nDocLength + 15) / 16;

    SetScrollSizes(MM_TEXT, CSize(0, m_nLinesTotal * m_cyScreen),
        CSize(0, m_cyScreen * 10), CSize(0, m_cyScreen));
    ScrollToPosition(CPoint(0, 0));
```

```
}

/////////////////////////////////////////////////////////////////////////////
// CHexView printing

BOOL CHexView::OnPreparePrinting(CPrintInfo* pInfo)
{
    // default preparation
    return DoPreparePrinting(pInfo);
}

void CHexView::OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo)
{
    //
    // Create a printer font.
    //
    m_fontPrinter.CreatePointFont (100, _T ("Courier New"), pDC);

    //
    // Compute the width and height of a line in the printer font.
    //
    TEXTMETRIC tm;
    CFont* pOldFont = pDC->SelectObject (&m_fontPrinter);
    pDC->GetTextMetrics (&tm);
    m_cyPrinter = tm.tmHeight + tm.tmExternalLeading;
    CSize size = pDC->GetTextExtent (_T ("---1----2----"\
        "3----4----5----6----7----8-"), 81);
    pDC->SelectObject (pOldFont);
    m_cxWidth = size.cx;
    //
    // Compute the page count.
    //
    m_nLinesPerPage = (pDC->GetDeviceCaps (VERTRES) -
        (m_cyPrinter * (3 + (2 * PRINTMARGIN)))) / m_cyPrinter;
    UINT nMaxPage = max (1, (m_nLinesTotal + (m_nLinesPerPage - 1)) /
        m_nLinesPerPage);
    pInfo->SetMaxPage (nMaxPage);

    //
    // Compute the horizontal offset required to center
    // each line of output.
    //
    m_cxOffset = (pDC->GetDeviceCaps (HORZRES) - size.cx) / 2;
}

void CHexView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
{
    CHexDoc* pDoc = GetDocument ();
```

```
    PrintPageHeader (pDoc, pDC, pInfo->m_nCurPage);
    PrintPage (pDoc, pDC, pInfo->m_nCurPage);
}

void CHexView::OnEndPrinting(CDC * pDC, CPrintInfo * pInfo)
{
    m_fontPrinter.DeleteObject ();
}

/////////////////////////////////////////////////////////////////////////////
// CHexView diagnostics

#ifdef _DEBUG
void CHexView::AssertValid() const
{
    CScrollView::AssertValid();
}

void CHexView::Dump(CDumpContext& dc) const
{
    CScrollView::Dump(dc);
}

CHexDoc * CHexView::GetDocument() // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CHexDoc)));
    return (CHexDoc * )m_pDocument;
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CHexView message handlers

int CHexView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CScrollView::OnCreate(lpCreateStruct) == -1)
        return -1;

    //
    // Create a screen font.
    //
    m_fontScreen.CreatePointFont (100, _T("Courier New"));

    //
    // Compute the height of one line in the screen font.
    //
    CClientDC dc (this);
    TEXTMETRIC tm;
```

```
    CFont * pOldFont = dc.SelectObject (&m_fontScreen);
    dc.GetTextMetrics (&tm);
    m_cyScreen = tm.tmHeight + tm.tmExternalLeading;
    dc.SelectObject (pOldFont);
    return 0;
}

void CHexView::FormatLine(CHexDoc * pDoc, UINT nLine, CString& string)
{
    //
    // Get 16 bytes and format them for output.
    //
    BYTE b[17];
    ::FillMemory (b, 16, 32);
    UINT nCount = pDoc->GetBytes (nLine * 16, 16, b);

    string.Format (_T ("%0.8X    %0.2X %0.2X %0.2X %0.2X %0.2X %0.2X %0.2X "\
        "%0.2X %0.2X - %0.2X %0.2X %0.2X %0.2X %0.2X %0.2X %0.2X "\
        "%0.2X    "), nLine * 16,
        b[0], b[1], b[2], b[3], b[4], b[5], b[6], b[7],
        b[8], b[9], b[10], b[11], b[12], b[13], b[14], b[15]);

    //
    // Replace non-printable characters with periods.
    //
    for (UINT i = 0; i < nCount; i++) {
        if (!::IsCharAlphaNumeric (b[i]))
            b[i] = 0x2E;
    }

    //
    // If less than 16 bytes were retrieved, erase to the end of the line.
    //
    b[nCount] = 0;
    string += b;

    if (nCount < 16) {
        UINT pos1 = 59;
        UINT pos2 = 60;
        UINT j = 16 - nCount;

        for (i = 0; i < j; i++) {
            string.SetAt (pos1, _T (' '));
            string.SetAt (pos2, _T (' '));
            pos1 -= 3;
            pos2 -= 3;
            if (pos1 == 35) {
                string.SetAt (35, _T (' '));
                string.SetAt (36, _T (' '));
```

```
                pos1 = 33;
                pos2 = 34;
            }
        }
    }
}

void CHexView::PrintPageHeader(CHexDoc * pDoc, CDC * pDC, UINT nPageNumber)
{
    //
    // Formulate the text that appears at the top of page.
    //
    CString strHeader = pDoc->GetPathName();
    if (strHeader.GetLength() > 60)
        strHeader = pDoc->GetTitle();

    CString strPageNumber;
    strPageNumber.Format(_T("Page %d"), nPageNumber);

    UINT nSpaces =
        81 - strPageNumber.GetLength() - strHeader.GetLength();
    for (UINT i = 0; i < nSpaces; i++)
        strHeader += _T(' ');
    strHeader += strPageNumber;

    //
    // Output the text.
    //

    UINT y = m_cyPrinter * PRINTMARGIN;
    CFont * pOldFont = pDC->SelectObject(&m_fontPrinter);
    pDC->TextOut(m_cxOffset, y, strHeader);

    //
    // Draw a horizontal line underneath the line of text.
    //
    y += (m_cyPrinter * 3) / 2;
    pDC->MoveTo(m_cxOffset, y);
    pDC->LineTo(m_cxOffset + m_cxWidth, y);

    pDC->SelectObject(pOldFont);
}

void CHexView::PrintPage(CHexDoc * pDoc, CDC * pDC, UINT nPageNumber)
{
    if (m_nLinesTotal != 0) {
        UINT nStart = (nPageNumber - 1) * m_nLinesPerPage;
        UINT nEnd = min(m_nLinesTotal - 1, nStart + m_nLinesPerPage - 1);
```

```
            CFont * pOldFont = pDC->SelectObject(&m_fontPrinter);
            for (UINT i = nStart; i <= nEnd; i++) {
                CString string;
                FormatLine(pDoc, i, string);
                UINT y = ((i - nStart) + PRINTMARGIN - 3) * m_cyPrinter;
                pDC->TextOut(m_cxOffset, y, string);
            }
            pDC->SelectObject(pOldFont);
        }
    }
```

图 13-7　HexDump 程序

13.3.1　串行化的唯一方法

HexDump 中值得特别注意的一个方面是它对文档进行特殊方式的串行化。当调用 CHexDoc::Serialize 从磁盘上读取文档时，它并没有从档案中读取，而是分配了一个大小与文件尺寸相当的缓冲区并用 CFile::Read 将文件读到缓冲区中。去掉异常处理程序，下列就是实现的代码：

```
if (ar.IsLoading()) {
    CFile* pFile = ar.GetFile();
    m_nDocLength = (UINT) pFile->GetLength();
    m_pFileData = new BYTE[m_nDocLength];
    pFile->Read(m_pFileData, m_nDocLength);
}
```

CArchive::GetFile 返回一个与档案关联的 CFile 指针，应用程序可以用它直接调用 CFile 成员函数。这是 MFC 应用程序用来读取和写入别人保存的二进制文档的一种方法。在文档的 DeleteContents 函数被调用之后，HexDump 释放包含原始文件数据的缓冲区：

```
delete[] m_pFileData;
```

HexDump 没有将文件内容串行化写回磁盘，它是个十六进制文件浏览器而不是编辑器，如果允许文档被编辑保存，就要使用 CFile::Write 将修改后的文档写回磁盘，方法和用 CFile::Read 将它们读取到内存中一样。

对于串行化和查看一个大型文档，分配一个与文件尺寸同等大小的缓冲区不是高效的方法，因为这样的话整个文档就会一下都进入内存。还有解决此问题的回旋余地(内存映射文件就是一个办法)，但是对于 HexDump 来说这还不是一个具有很大实际意义的问题，由 CScrollView 产生的问题要比可用内存的限制问题更关键。找一个有几百 KB 的文件并把它

在 HexDump 打开，就会明白这一点了。如果运行在 Windows 95 或 Windows 98 环境下，HexDump显示不了大约超过 1 000 行的文件，怎么会这样呢？

问题出在 Windows 95 和 Windows 98 中的 16 位继承上。在这两个操作系统中滚动条范围是一个 16 位值。在 CHexView::OnInitialUpdate 调用 SetScrollSizes 之前，它通过用文档中的行数乘以每行的像素数来计算视图的可视高度。如果一行是 16 个像素，文档包含 1 000 行，结果高度就是 16 000。对于小文档，这还凑合；但是 CScrollView 不能处理高度大于 32 767(带符号 16 位整数所能表示的最大正数)的文档，因为这是滚动条范围的上限。实际结果会怎样呢？如果打开一个包含很多行的文档，即使打印和打印预览仍能正常工作，CScrollView 也只能显示文档的一部分。要让 HexDump 可以处理大型文档，最好的办法是创建带有滚动条的 CView 并亲自处理滚动条消息。有关在 MFC 中处理滚动条消息的详细内容可以参考第 2 章。

13.4　打印技巧与诀窍

这里提供了一些技巧和诀窍以及常见问题的答案，以帮助您编写出更好的打印程序，同时还要解决一些在本章示例程序中没有提到的问题。

13.4.1　使用“打印”对话框中的“选定范围”单选按钮

在开始打印之前 MFC 显示的“打印”对话框中包含一个“选定范围”单选按钮，单击它可以打印当前用户选中内容而不是整个或一部分文档。默认时该按钮无效。可以在调用 CDoPreparePrinting 之前给 OnPreparePrinting 添加下列语句使它有效：

```
pInfo->m_pPD->m_pd.Flags &= ~PD_NOSELECTION;
```

有效后为选中该按钮，还要添加如下语句：

```
pInfo->m_pPD->m_pd.Flags |= PD_SELECTION;
```

m_pPD 是传递给 OnPreparePrinting 的 CPrintInfo 结构的数据成员，指向 CPrintDialog 对象，DoPreparePrinting 使用该对象显示“打印”对话框。CPrintDialog::m_pd 保存着对对话框的基础 PRINTDLG 结构的引用，而 PRINTDLG 的 Flags 字段保存着定义了对话框属性的位标志。删除由 CPrintInfo 的构造函数添加的 PD_NOSELECTION 标志可以使“选定范围”按钮有效，添加 PD_SELECTION 标志就可以选中该按钮。如果 DoPreparePrinting 返回非零值，说明对话框由“确定”按钮释放了，通过调用 CPrintDialog::PrintSelection 可以确定“选定范围”按钮是否被选中。非零返回值意味着选中；0 意味着没有选中：

```
if (pInfo->m_pPD->PrintSelection()) {
   // Print the current selection.
}
```

在 DoPreparePrinting 返回之后，可以通过传递给 OnPreparePrinting 的 pInfo 参数来调用 PrintSelection和其他 CPrintDialog 函数，它们返回有关“打印”或“打印设置”对话框中的设置信息。还可以通过传递给 OnBeginPrinting 和其他 CView 覆盖函数的 pInfo 参数来调用它们。

可以用其他方法使用 CPrintInfo::m_pPD 来修改 DoPreparePrinting 生成的“打印”对话框的外观和功能。从 Visual C++ 的联机帮助文档可以得到有关 PRINTDLG 和它的数据成员的详细信息。

13.4.2 不要假定——实践出真知!

在往打印纸上输出时，对打印纸的可打印页区域所作的任何假定通常都有可能是错误的。即使您了解使用的打印纸，比如是 8½ × 11 英寸的打印纸，可打印页区域也可能由于打印机的不同而不同。即使用相同的打印机、相同的打印纸，由于所用的打印机驱动程序不同，可打印页区域也不同，如果用户选择以横向而不是纵向模式打印，可打印页区域的水平和垂直尺寸也会改变。不要假定有足够的打印空间，要像 HexDump 那样在每次打印时都通过提供给 CView 打印函数的 CDC 指针调用 GetDeviceCaps，或使用 OnPrint 中的 CPrintInfo::m_rectDraw 来确定可打印页区域。这种简单的预防措施可以确保您的打印程序能够用于 Windows 支持的所有打印机上，并且会极大地减少在用户使用中出现的问题。

正如您所知道的，用 HORZRES 和 VERTRES 参数调用 GetDeviceCaps 将返回可打印页区域的水平和垂直尺寸。还可以将表 13-3 中给出的值传递给 GetDeviceCaps，以获取有关打印机或其他硬拷贝设备的详细信息。

表 13-3 传递给 GetDeviceCaps 的值

值	说 明
HORZRES	以像素为单位返回可打印页区域的宽度
VERTRES	以像素为单位返回可打印页区域的高度
HORSIZE	以毫米为单位返回可打印页区域的宽度
VERTSIZE	以毫米为单位返回可打印页区域的高度
LOGPIXELSX	返回水平方向上每英寸的像素个数(对于 300-dpi 打印机为 300)
LOGPIXELSY	返回垂直方向上每英寸的像素个数(对于 300-dpi 打印机为 300)
PHYSICALWIDTH	以像素为单位返回页宽度(对于 300-dpi 打印机上使用的 8½ × 11 英寸打印纸为 2 550)
PHYSICALHEIGHT	以像素为单位返回页高度(对于 300-dpi 打印机上使用的 8½ × 11 英寸打印纸为 3 300)

续表

值	说　明
PHYSICALOFFSETX	以像素为单位返回从打印纸的左边到可打印页区域开始处的距离
PHYSICALOFFSETY	以像素为单位返回从打印纸的顶边到可打印页区域开始处的距离
TECHNOLOGY	返回一个值标识 DC 所属的输出设备的类型。最常见的返回值有：DT RASDISPLAY 标识屏幕、DT_RASPRINTER 标识打印机以及 DT_PLOTTER 标识绘图仪
RASTERCAPS	返回一系列位标志标识打印机驱动程序提供的 GDI 支持的级别。例如：RC_BITBLT 标志表明打印机支持 BitBlts，RC_STRETCHBLT 表明打印机支持 StretchBlts
NUMCOLORS	返回打印机支持的颜色数。对于黑白打印机返回值为 2

您已经知道了 NUMCOLORS 参数对 GetDeviceCaps 的用处：查明是否是使用黑白打印机以便进行灰度打印预览输出。PHYSICALOFFSETX 和 PHYSICALOFFSETY 参数用于根据用户在 Page Setup 对话框中输入的信息设置页边距。(MFC 的 CWinApp::OnFilePrintSetup 函数显示 Print Setup 对话框而不是 Page Setup 对话框，但是可以利用 MFC 的 CPageSetupDialog 类来显示 Page Setup 对话框。)例如：如果用户想在打印纸左边得到 1 英寸页边距，就可以从每英寸打印像素点总数(LOGPIXELSX)中减去由 GetDeviceCaps 返回的 PHYSICALOFFSETX 值来计算可打印页区域左边打印开始处的 x 偏移量。如果打印机驱动程序返回准确的信息，结果页边距就会以几像素的误差等于 1 英寸。可以使用 HORZRES、VERTRES、LOGPIXELSX、LOGPIXELSY、PHYSICALWIDTH、PHYSICALHEIGHT、PHYSICALOFFSETX 以及 PHYSICALOFFSETY 值来刻画打印纸上可打印区域特点，并精确地指定在打印纸中可打印区域的位置。

如果您担心某些硬拷贝设备不可以绘制位图，就可以通过使用 RASTERCAPS 参数调用 GetDeviceCaps 并检查返回的标志来明确它是否支持 CDC::BitBlt 和 CDC::StretchBlt。在大多数情况下，只有矢量设备如绘图仪不支持 GDI 的 Blt 函数。如果对于某种光栅设备，驱动程序不支持直接的位图输出，GDI 可以自己进行位图输出来完善此功能。您可以直接确定打印是否必定输出到绘图仪上，只要用 TECHNOLOGY 参数调用 GetDeviceCaps 并检查其返回的值是否等于 DT_PLOTTER 即可。

在使用多种不同的打印机测试应用程序的打印功能时，您会发现打印机驱动程序在报告信息以及生成结果方面居然非常混乱。例如：有些打印机驱动程序对于 PHYSICALWIDTH 和 PHYSICALHEIGHT 返回的值与对 HORZRES 和 VERTRES 返回的值完全相同。有时普通的 GDI 函数如 CDC::TextOut 可以在许多打印机上顺利工作，但会由于驱动程序出错而造成在某种特殊模式下失败，还有时 GDI 函数可能完全不会失败但在不同的打印机上它对打印机进行的操作却完全不同。我曾经遇到过一个打印机驱动程序，虽然其他驱动程序对于同一系列的打印机都能正确地设置设备描述表的默认背景模式为 OPAQUE，可是它却

将背景模式默认设置为 TRANSPARENT。打印机驱动程序的不可靠真是臭名昭著,因此必须事先预见到可能的问题并在尽可能多的打印机上进行完整的测试。您程序中的打印功能越出众,驱动程序的古怪行为就可能越需要您编写一组工具来解决某些打印机上突然出现的问题。

13.4.3 添加默认分页支持

HexDump 从 OnBeginPrinting 而不是从 OnPreparePrinting 中调用 CPrintInfo::SetMaxPage,因为分页处理依赖于可打印页区域,而且 OnBeginPrinting 是第一个用指向打印机设备描述表的指针调用的 CView 函数。由于在 OnPreparePrinting 返回之后才设置最大页编号,所以在 Print 对话框中 From 框中已(用 1)填写了值,而 To 框中没有填写。某些用户可能会感到有些矛盾,应用程序对于打印预览可以进行正确的文档分页却不能填写对话框中的最大页编号。除了正确显示最大页编号以外,许多商业应用程序还在打印预览之外显示页分隔并在状态栏中显示"Page mm of nn"字符串。在不知道文档将被打印到何种打印机上或不知道打印纸的打印方向的情况下,应用程序是如何知道文档是怎样被分页的呢?

答案是它们也无法确切知道,只能根据默认打印机的属性来做出最好的猜测。下列程序段用默认打印机或上次用户在 Print Setup 对话框中选择的打印机的可打印页区域的像素尺寸来初始化 CSize 对象。可以从 OnPreparePrinting 或其他地方调用它来计算页数或获取其他默认分页支持形式所需要的信息:

```
CSize size;
CPrintInfo pi;
if (AfxGetApp()->GetPrinterDeviceDefaults(&pi.m_pPD->m_pd)) {
    HDC hDC = pi.m_pPD->m_pd.hDC;
    if (hDC == NULL)
        hDC = pi.m_pPD->CreatePrinterDC();
    if (hDC != NULL) {
        CDC dc;
        dc.Attach(hDC);
        size.cx = dc.GetDeviceCaps(VERTRES);
        size.cy = dc.GetDeviceCaps(HORZRES);
        ::DeleteDC(dc.Detach());
    }
}
```

CWinApp::GetPrinterDeviceDefaults 用描述默认打印配置的值来初始化 PRINTDLG 结构。0 返回值意味着函数失败,通常表明没有安装打印机或还没有指定默认打印机。CPrintInfo::CreatePrinterDC 根据封装在 CPrintInfo 对象中 PRINTDLG 结构的信息来生成设备描述表句柄。得到设备描述表句柄之后,就可以简单地将它包在 CDC 对象中并调用 CDC::GetDeviceCaps 来测量可打印页区域。

13.4.4　枚举打印机

有时生成一个有效的包含所有打印机的列表很有用，以方便用户在 Print 或 Print Setup 对话框以外选中打印机。下列程序使用了 Win32::EnumPrinters 函数来枚举当前安装了的打印机，并在由 pComboBox 所指的组合框中为每个打印机都添加了一个选项。

```
#include <winspool.h>
   .
   .
   .
DWORD dwSize, dwPrinters;
::EnumPrinters (PRINTER_ENUM_LOCAL, NULL, 5,
     NULL, 0, &dwSize, &dwPrinters);

BYTE* pBuffer = new BYTE[dwSize];

::EnumPrinters (PRINTER_ENUM_LOCAL, NULL, 5,
     pBuffer, dwSize, &dwSize, &dwPrinters);

if (dwPrinters != 0)
    PRINTER_INFO_5* pPrnInfo = (PRINTER_INFO_5*) pBuffer;
    for (UINT i = 0; i < dwPrinters; i++)
        pComboBox->AddString (pPrnInfo->pPrinterName);
        pPrnInfo++;

}

delete[] pBuffer;
```

第 1 次调用::EnumPrinters 来检索为保存描述各个打印机的 PRINTER_INFO_5 结构数组所需的缓冲区的大小。第 2 次调用::EnumPrnters 是为了用 PRINTER_INFO_5 结构数组来初始化由 pBuffer 所指的缓冲区。返回后，dwPrinters 将保存枚举的打印机的数量（等于复制给缓冲区的 PRINTER_INFO_5 结构的总数），每个结构的 pPrinterName 字段都保存一个指向零定界字符串的指针，该字符串包含着相关打印机的设备名称。用 PRINTER_INFO_5 结构枚举打印机速度很快，因为不需要进行远程调用；所有填写缓冲区所需要的信息都可以从寄存器中得到。对于 Windows NT 或 Windows 2000 中的快速打印机枚举可以使用 PRINTER_INFO_4 结构。

如果在组合框中选中一个打印机并希望创建它的设备描述表，就可以将从 PRINTER_INFO_5 结构中复制的设备名称传递给 CDC::CreateDC，如下所示：

```
CString strPrinterName;
```

```
int nIndex = pComboBox->GetCurSel();
pComboBox->GetLBText(nIndex, strPrinterName);

CDC dc;
dc.CreateDC(NULL, strPrinterName, NULL, NULL);
```

可以像使用那些地址被传递给 OnBeginPrinting 和其他 CView 打印函数的 CDC 对象那样来使用这些 CDC 对象。

第Ⅲ部分

高　级　篇

第 14 章　计时器和空闲处理

并非 Micsosoft Windows 应用程序中的所有操作都是响应用户输入才执行的。有些操作处理本身是基于时间的，如自动保存操作以 10 分钟为间隔保存文档并更新在状态栏中显示的与时钟有关的项目。Windows 提供了"计时器"，帮助您编写定期发送消息的程序。另一种与时间有关的处理是"空闲处理"，在消息队列中没有消息等候的"空闲"期间执行。MFC 以虚函数 OnIdle 的形式为空闲时间处理提供了一个主框架，每当 CWinThread 中的消息循环发现消息队列为空时就会调用该函数。

在本章的前半部分，我们将研究计时器，它可以编程实现最短间隔 55 毫秒。下面列出了几种计时器的用途：

- 用在显示挂钟时间的应用程序中。大多数这种应用程序都设置计时器来产生从半秒到 60 秒范围内的时间间隔。在计时器消息到达时，应用程序更新显示来反映当前时间。
- 用在无人看管的备份程序、磁盘碎片清理器以及其他在指定时间之前处于休眠状态的应用程序中。
- 用在资源监视程序、可用内存计量程序以及其他监视系统状态的程序中。

在本章的后半部分研究空闲处理，它是什么、是如何工作的，以及如何用它在 MFC 应用程序中执行后台处理任务。

14.1　计时器

使用计时器只需要了解两个函数。CWnd::SetTimer 用于产生以指定时间间隔发送消息的计时器，CWnd::KillTimer 消除计时器。根据传递给 SetTimer 的参数，计时器通过下面的两种途径通知应用程序间隔时间已到：

- 给指定窗口发送 WM_TIMER 消息
- 调用一个应用程序定义的回调函数

WM_TIMER 方式较简单，但回调函数有时却更常用，特别是在使用了多个计时器时。这两种类型的计时器消息在发送给应用程序时具有较低的优先级别，只有在消息队列中没有其他消息时才处理它们。

计时器消息永远不会积压在消息队列中。如果将计时器设置为每 100 毫秒产生一个消息,而在整整 1 秒内应用程序都忙于处理其他消息了,当消息队列清空以后它并不会突然接收到 10 个快速产生的消息。相反,它只接收到一条。因此您没必要担心处理一个计时器消息会化很长时间,使得还未处理完前一个消息又接收到第二个消息,从而出现消息竞争。但是 Windows 应用程序也决不应该用大量的时间处理一个消息,除非消息处理委派给了后台线程,因为那样的话如果主线程在长时间内不检查消息队列就会造成响应能力的下降。

14.1.1 设置计时器:方法 1

设置计时器最简单的方法是用计时器 ID 和计时器时间间隔调用 SetTimer,然后将 WM_TIMER 消息映射给 OnTimer 函数。计时器 ID 是唯一标识计时器的非零值。在响应 WM_TIMER 消息而激活 OnTimer 时,计时器 ID 将作为一个参数被传递。如果您只使用了一个计时器,您可能对 ID 值就不很关心了,因为所有的 WM_TIMER 消息都来自同一个计时器。使用了两个以上的计时器的应用程序会使用计时器 ID 来标识产生特定消息的计时器。

传递给 SetTimer 的计时器间隔指定了以毫秒为单位连续两次 WM_TIMER 消息之间的时间间隔。有效值从 1 毫秒到 32 位整数所能表示的最大值——$2^{32}-1$ 毫秒,大概是 49 天半。语句

```
SetTimer(1,500,NULL);
```

分配了一个计时器,并赋给其 ID 为 1,它每 500 毫秒给窗口发送一个 WM_TIMER 消息。第三个参数 NULL 将计时器配置为发送 WM_TIMER 消息而不是调用回调函数。虽然指定的间隔是毫秒,但是实际窗口每 550 毫秒接收到一个消息,这是因为在大多数系统中(特别是基于 Intel 的系统),Windows 计时器基于的硬件计时器或多或少每 54.9 毫秒走一下。实际上,Windows 要将传递给 SetTimer 的值四舍五入到下一个 55 毫秒。因此,语句

```
SetTimer(1,1,NULL);
```

安排计时器大约每 55 毫秒发送一个 WM_TIMER 消息,与以下语句相同:

```
SetTimer(1,50,NULL);
```

但如果将间隔修改为 60,如下:

```
SetTimer(1,60,NULL);
```

WM_TIMER 消息就会平均每 110 毫秒到达一次。

一旦计时器被设置以后,WM_TIMER 消息之间的时间间隔又是怎样的呢?表 14-1 中列出的计时器消息之间的时间来自于一个 32 位 Windows 应用程序,它安排了计时器每 500 毫秒发送一次消息。

表 14-1　计时器消息之间的时间

通知顺序号	间　隔	通知顺序号	间　隔
1	0.542 秒	11	0.604 秒
2	0.557 秒	12	0.550 秒
3	0.541 秒	13	0.549 秒
4	0.503 秒	14	0.549 秒
5	0.549 秒	15	0.550 秒
6	0.549 秒	16	0.508 秒
7	1.936 秒	17	0.550 秒
8	0.261 秒	18	0.549 秒
9	0.550 秒	19	0.549 秒
10	0.549 秒	20	0.550 秒

可以看出平均时间接近 550 毫秒，而且大部分单独间隔时间也接近 550 毫秒。唯一一个显著抖动发生在第六和第七个 WM_TIMER 消息之间，用去 1.936 秒，此时在屏幕上拖动了窗口。在这个表中可以明白地看出 Windows 不允许在消息队列中积累计时器消息。假如允许，在 1.936 秒的延时之后窗口会接收到快速连续的 3 或 4 个计时器消息。

从此例中我们知道了决不能依靠计时器进行类似码表那样的精确计时。如果编写一个时钟应用程序，安排计时器为 1 000 毫秒间隔并在每次 WM_TIMER 消息到达时更新显示，则不应该假定 60 个 WM_TIMER 消息就意味着逝去了一分钟。实际上，应该在每次消息到达时都检查当前时间并相应更新时钟。那样的话即使计时器消息流被打断，也会维持时钟的精确值。

如果要编写需要精确计时的应用程序，可以在用到常规计时器的地方使用 Windows 多媒体计时器，并将其设置为 1 毫秒或更少的时间间隔。多媒体计时器提供了高精度计时功能，对于特殊的应用程序如 MIDI 音序器非常理想，但它也会产生额外开销并且对系统中运行的其他程序可能造成负面影响。

如果函数执行成功，SetTimer 返回的值为计时器 ID，若失败则为 0。在 16 位 Windows 中，计时器由全局资源共享并只能具有有限个计时器。在 32 位 Windows 中，系统可以支持的计时器数量实际上是无限的。虽然失败情况很少，但还是要谨慎，应检查返回值以防系统资源不足（不要忘了小疏忽能误大事）。应用程序如果设置太多计时器就会降低整个系统的运行效率）。由 SetTimer 返回的计时器 ID 与函数的第一个参数指定的计时器 ID 相同，除非参数中指定了 0，那样的话 SetTimer 将返回计时器 ID 为 1。如果给两个以上的计时器分配了相同的 ID，SetTimer 也不会失败。相反，它会按要求分配复制的 ID。

还可以使用 SetTimer 改动从前分配的已用完的时间值。如果计时器 1 已经存在，语句

```
SetTimer(1,1000,NULL);
```

将重新给它安排时间间隔 1 000 毫秒，重新规划计时器同时也重置了它的内部时钟，使得下一个消息不会在指定的时间段用完之前到达。

14.1.2　响应 WM_TIMER 消息

MFC 的 ON_WM_TIMER 消息映射表宏将 WM_TIMER 消息映射给了类成员函数 OnTimer。OnTimer 的原型如下：

```
afx_msg void OnTimer(UINT nTimerID)
```

nTimerID 是产生消息的计时器的 ID。可以在 OnTimer 中做能够在其他消息处理函数中所做的任何事情，包括捕获设备描述表以及绘制窗口。下列示例程序使用 OnTimer 处理程序在框架窗口的客户区绘制任意尺寸和颜色的椭圆。在窗口的 OnCreate 处理程序中给计时器设置了 1 000 毫秒的时间间隔：

```
BEGIN_MESSAGE_MAP(CMainWindow, CFrameWnd)
    ON_WM_CREATE()
    ON_WM_TIMER()
END_MESSAGE_MAP()

int CMainWindow::OnCreate(LPCREATESTRUCT lpcs)
{
    if (CFrameWnd::OnCreate(lpcs) == -1)
        return -1;

    if (!SetTimer(ID_TIMER_ELLIPSE, 100, NULL)) {
        MessageBox(_T("Error: SetTimer failed"));
        return -1;
    }
    return 0;
}

void CMainWindow::OnTimer(UINT nTimerID)
{
    CRect rect;
    GetClientRect(&rect);

    int x1 = rand() % rect.right;
    int x2 = rand() % rect.right;
    int y1 = rand() % rect.bottom;
    int y2 = rand() % rect.bottom;

    CClientDC dc(this);
    CBrush brush(RGB(rand() % 255, rand() % 255,
        rand() % 255));
```

```
    CBrush* pOldBrush = dc.SelectObject (&brush);
    dc.Ellipse (min (x1, x2), min (y1, y2), max (x1, x2),
        max (y1, y2));
    dc.SelectObject (pOldBrush);
}
```

下列的程序段给出了应用程序修改后的样子，它使用了两个计时器，一个用于绘制椭圆，另一个用来绘制矩形：

```
BEGIN_MESSAGE_MAP (CMainWindow, CFrameWnd)
    ON_WM_CREATE ()
    ON_WM_TIMER ()
END_MESSAGE_MAP ()

int CMainWindow::OnCreate (LPCREATESTRUCT lpcs)
{
    if (CFrameWnd::OnCreate (lpcs) == -1)
        return -1;

    if (!SetTimer (ID_TIMER_ELLIPSE, 100, NULL) ||
        !SetTimer (ID_TIMER_RECTANGLE, 100, NULL))
        MessageBox (_T ("Error: SetTimer failed"));
        return -1;

    return 0;
}

void CMainWindow::OnTimer (UINT nTimerID)
{
    CRect rect;
    GetClientRect (&rect);

    int x1 = rand () % rect.right;
    int x2 = rand () % rect.right;
    int y1 = rand () % rect.bottom;
    int y2 = rand () % rect.bottom;

    CClientDC dc (this);
    CBrush brush (RGB (rand () % 255, rand () % 255, rand () % 255));
    CBrush* pOldBrush = dc.SelectObject (&brush);
    if (nTimerID == ID_TIMER_ELLIPSE)
        dc.Ellipse (min (x1, x2), min (y1, y2), max (x1, x2),
            max (y1, y2));
    else // nTimerID == ID_TIMER_RECTANGLE
        dc.Rectangle (min (x1, x2), min (y1, y2), max (x1, x2),
            max (y1, y2));
```

```
    dc.SelectObject (pOldBrush);
}
```

可以看到,此版本的 OnTimer 检查传递给它的 nTimerID 值来确定是绘制椭圆还是绘制矩形。

您可能不会编写许多不断地绘制椭圆和矩形的应用程序,但是使用计时器消息执行某种任务或重复进行的任务序列在 Windows 程序设计中给常遇到的一类问题提供了简单的解决方式。假设您要编写一个带有两个按钮控件"开始"和"停止"的应用程序,单击开始按钮可以启动绘图循环,如下:

```
m_bContinue = TRUE;
while (m_bContinue)
    DrawRandomEllipse ();
```

在循环中反复绘制椭圆,直到单击"停止"按钮,将 m_bContinue 设置为 FALSE 以便退出 while 循环。这看上去很合理,但只要试一下就会发现并不是这样。一旦"开始"被单击,while 循环将运行到 Windows 会话结束或用"任务管理器"中断应用程序为止。为什么会这样?因为将 m_bContinue 设置为 FALSE 的语句只有在由"停止"按钮产生的 WM_COMMAND 消息被检索、调度、通过消息映射表传递给相应的 ON_COMMAND 处理程序之后才会执行。但是只要 while 循环持续运行而不检查消息,WM_COMMAND 消息就会闲置在消息队列中等待检索。m_bContinue 永远不会从 TRUE 修改为 FALSE,程序将陷入死循环。

可以用几种方法解决此问题。一种办法是在次要的线程中绘制图形而主要的线程还可以继续提取消息。另一种方法是在 while 循环中添加消息循环,在绘制椭圆时可以周期性地检查消息队列。第三种方法是在响应 WM_TIMER 消息时绘制椭圆。在 WM_TIMER 消息之间,其他消息还可以得到正常的处理。此方法的唯一缺点是绘制椭圆的速度超过每秒 18 个时就需要用多个计时器,而线程在绘制完一个椭圆后会紧接着绘制下一个椭圆,尽管会受视频硬件和椭圆尺寸不同的影响,但每秒也能绘制成百上千个椭圆。

要记住重要的一点,对于其他消息而言 WM_TIMER 消息不是被异步处理的。就是说,WM_TIMER 消息永远不会中断同一个线程中的另一个 WM_TIMER 消息,也不会中断非计时器消息。WM_TIMER 消息和其他消息一样在消息队列中等待被检索,在由消息循环检索调度之前不会得到处理。如果一个定期消息处理函数和 OnTimer 函数使用了公用的成员变量,只要两个消息处理程序属于同一个窗口或是运行在同一个线程上的窗口,您就可以安全地假定对变量的访问不会重叠。

14.1.3 设置计时器:方法 2

计时器并不是必须产生 WM_TIMER 消息。如果愿意,可以配置计时器在应用程序中调

用一个回调函数而不是发送 WM_TIMER 消息。此方法通常用在使用了多个计时器的应用程序中，使得可以给每个计时器都分配唯一的处理函数。

在 Windows 程序员中共同存在着一个误解，认为处理计时器回调函数要比处理计时器消息方便，因为回调函数是直接由操作系统调用的而 WM_TIMER 消息要放置在消息映射表中。事实上，在调用::DispatchMessage 之前对计时器回调函数和计时器消息的处理过程是相同的。在计时器开始运行以后，Windows会在消息队列中设置一个标志，用来表示计时器消息或回调函数是否在等待处理(开/关标志的特性就说明了为什么计时器消息不会在消息队列中堆积。在计时器间隔时间过去以后标志不会累加而只能设置为开状态)。如果::GetMessage 发现消息队列是空的并且没有窗口需要重绘，它就检查计时器标志。如果标志被设置了，::GetMessage 就生成一个 WM_TIMER 消息，接下来由::DispatchMessage 调度。如果产生消息的计时器是 WM_TIMER 类型的，消息就会调度到窗口去处理。但是如果已经注册了回调函数，::DispatchMessage 就要调用回调函数了。所以，回调函数计时器在运作上并不比消息计时器有多少长处。回调函数与消息相比用的开销要稍微少些，因为它不涉及消息映射和窗口处理，但它们之间的差别仅此而已，并不很大。在现实工作中，您会发现 WM_TIMER 类型的计时器和回调函数类型的计时器都经常被使用(或不经常，看您怎么想了)。

要设置一个使用回调函数的计时器，可以在 SetTimer 的第 3 个参数中指定回调函数的名字，如下：

```
SetTimer (ID_TIMER, 100, TimerProc);
```

在本例中被命名为 TimerProc 的回调函数的原型如下：

```
void CALLBACK TimerProc (HWND hWnd, UINT nMsg,
  UINT nTimerID, DWORD dwTime)
```

TimerProc 中的 hWnd 参数保存窗口句柄，nMsg 保存消息 ID WM_TIMER，nTimerID 保存计时器 ID，而 dwTime 指定从 Windows 启动以后经过时间的毫秒数。(有些文献中说 dwTime "以 Coordinated Universal Time 格式指定了系统时间"，实际上这是错的。)回调函数应该是一个静态成员函数或全局函数，防止 this 指针传递给它。有关回调函数和非静态成员函数给 C++ 应用程序所造成问题的更详细说明请参见第 7 章。

在使用静态成员函数作为计时器回调函数时会遇到一个障碍，计时器处理过程中不像一些 Windows 回调函数那样可以接收到用户定义的 lParam 值。在第 7 章中当使用静态成员函数从::EnumFontFamilies 引出回调函数时，我们在 lParam 中传递了一个 CMainWindow 指针并允许回调函数访问非静态类成员。在计时器处理中，如果希望访问非静态函数和数据成员的话就必须采用其他的途径来获得指针。幸运的是，您可以使用 MFC 的 AfxGetMainWnd

函数给应用程序的主窗口获取一个指针。

```
CMainWindow* pMainWnd = (CMainWindow*) AfxGetMainWnd();
```

如果想访问 CMainWindow 函数和数据成员,那么将返回值强制转换为 CMainWindow 指针是必要的,因为 AfxGetMainWnd 返回的指针一般是 CWnd 指针。一旦用这种方法初始化了 pMainWnd,作为 CMainWindow 成员之一的 TimerProc 函数就可以访问非静态 CMainWindow 函数和数据成员了,就如同它自己是非静态成员函数一样。

14.1.4 清除计时器

与 CWnd::SetTimer 对应的是 CWnd::KillTimer,它清除计时器并停止 WM_TIMER 消息流或计时器回调函数。下列语句释放 ID 为 1 的计时器:

```
KillTimer(1);
```

窗口的 OnClose 或 OnDestroy 处理程序是清除 OnCreate 创建的计时器的好地方。如果应用程序在结束之前没有释放计时器,32 位的 Windows 就会在处理结束之后清理它。但是好的编程风格要求每次调用 SetTimer 都应该配对地调用 KillTimer 以确保计时器资源被恰当地释放。

14.2 CLOCK 应用程序

图 14-1 所示的 Clock 应用程序使用了一个计时器,设置为每隔 1 秒就周期性地重绘 Clock 指针来模拟时钟。Clock 不是一个文档/视图应用程序;它使用了本书前几章讲述的 MFC 1.0 样式的应用程序体系结构。它的所有源代码,包括 RC 文件,都是手工生成的(参阅图 14-2)。除了说明如何在 Windows 应用程序中使用计时器以外,Clock 还介绍了称为 CTime 的 MFC 类以及一个新消息 WM_MINMAXINFO。它还具有若干个别的与计时器无关的有趣特性,包括:

- 系统菜单中的一个命令,用来删除窗口的标题栏
- 系统菜单中的一个命令,用来使 Clock 的窗口即使在后台运行时也可以成为最顶层窗口(绘制在其他所有窗口之上的窗口)
- 记得其大小和位置的框架窗口
- 可用其客户区域拖动的框架窗口

在以后几节中我们将讨论应用程序的这些以及其他特性。

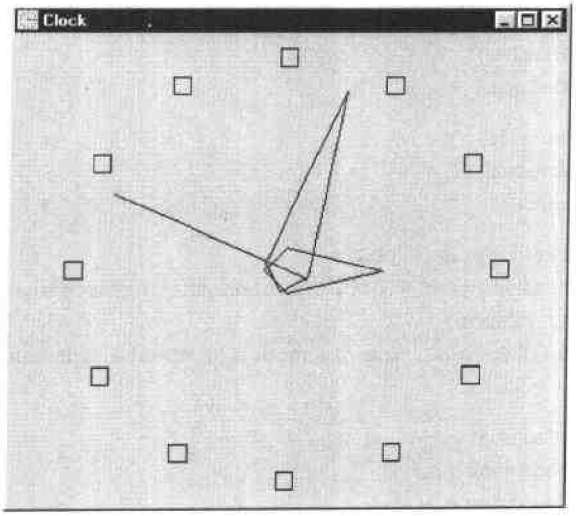

图 14-1　Clock 窗口

Resource.h

```
# define IDM_SYSMENU_FULL_WINDOW      16
# define IDM_SYSMENU_STAY_ON_TOP      32
# define IDI_APPICON                  100
```

Clock.rc

```
# include < afxres.h >
# include "Resource.h"
IDI_APPICON ICON Clock.ico
```

Clock.h

```
class CMyApp :  public CWinApp
{
public:
    virtual BOOL InitInstance ();
};

class CMainWindow : public CFrameWnd
{
```

```
protected:
    BOOL m_bFullWindow;
    BOOL m_bStayOnTop;

    int m_nPrevSecond;
    int m_nPrevMinute;
    int m_nPrevHour;

    void DrawClockFace (CDC * pDC);
    void DrawSecondHand (CDC * pDC, int nLength, int nScale, int nDegrees,
        COLORREF clrColor);
    void DrawHand (CDC * pDC, int nLength, int nScale, int nDegrees,
        COLORREF clrColor);

    void SetTitleBarState ();
    void SetTopMostState ();
    void SaveWindowState ();
    void UpdateSystemMenu (CMenu * pMenu);

public:
    CMainWindow ();
    virtual BOOL PreCreateWindow (CREATESTRUCT& cs);
    BOOL RestoreWindowState ();

protected:
    afx_msg int OnCreate (LPCREATESTRUCT lpcs);
    afx_msg void OnGetMinMaxInfo (MINMAXINFO * pMMI);
    afx_msg void OnTimer (UINT nTimerID);
    afx_msg void OnPaint ();
    afx_msg UINT OnNcHitTest (CPoint point);
    afx_msg void OnSysCommand (UINT nID, LPARAM lParam);
    afx_msg void OnContextMenu (CWnd * pWnd, CPoint point);
    afx_msg void OnEndSession (BOOL bEnding);
    afx_msg void OnClose ();

  DECLARE_MESSAGE_MAP ()
};
```

Clock.cpp

```
#include <afxwin.h>
#include <math.h>
#include "Clock.h"
#include "Resource.h"
#define SQUARESIZE 20
#define ID_TIMER_CLOCK 1
```

```
CMyApp myApp;

//////////////////////////////////////////////////////////////////////////
// CMyApp member functions

BOOL CMyApp::InitInstance ()
{
    SetRegistryKey (_T ("Programming Windows with MFC"));
    m_pMainWnd = new CMainWindow;
    if (! ((CMainWindow *) m_pMainWnd)->RestoreWindowState ())
        m_pMainWnd->ShowWindow (m_nCmdShow);
    m_pMainWnd->UpdateWindow ();
    return TRUE;
}

//////////////////////////////////////////////////////////////////////////
// CMainWindow message map and member functions

BEGIN_MESSAGE_MAP (CMainWindow, CFrameWnd)
    ON_WM_CREATE ()
    ON_WM_PAINT ()
    ON_WM_TIMER ()
    ON_WM_GETMINMAXINFO ()
    ON_WM_NCHITTEST ()
    ON_WM_SYSCOMMAND ()
    ON_WM_CONTEXTMENU ()
    ON_WM_ENDSESSION ()
    ON_WM_CLOSE ()
END_MESSAGE_MAP ()

CMainWindow::CMainWindow ()
{
    m_bAutoMenuEnable = FALSE;

    CTime time = CTime::GetCurrentTime ();
    m_nPrevSecond = time.GetSecond ();
    m_nPrevMinute = time.GetMinute ();
    m_nPrevHour = time.GetHour () % 12;

    CString strWndClass = AfxRegisterWndClass (
        CS_HREDRAW CS_VREDRAW,
        myApp.LoadStandardCursor (IDC_ARROW),
        (HBRUSH) (COLOR_3DFACE + 1),
        myApp.LoadIcon (IDI_APPICON)    );

    Create (strWndClass, _T ("Clock"));
}
```

```
BOOL CMainWindow::PreCreateWindow (CREATESTRUCT& cs)
{
    if (!CFrameWnd::PreCreateWindow (cs))
        return FALSE;

    cs.dwExStyle &= ~WS_EX_CLIENTEDGE;
    return TRUE;
}

int CMainWindow::OnCreate (LPCREATESTRUCT lpcs)
{
    if (CFrameWnd::OnCreate (lpcs) == -1)
        return -1;

    //
    // Set a timer to fire at 1-second intervals.
    //
    if (!SetTimer (ID_TIMER_CLOCK, 1000, NULL)) {
        MessageBox (_T ("SetTimer failed"), _T ("Error"),
            MB_ICONSTOP | MB_OK);
        return -1;
    }

    //
    // Customize the system menu.
    //
    CMenu* pMenu = GetSystemMenu (FALSE);
    pMenu->AppendMenu (MF_SEPARATOR);
    pMenu->AppendMenu (MF_STRING, IDM_SYSMENU_FULL_WINDOW,
        _T ("Remove &Title"));
    pMenu->AppendMenu (MF_STRING, IDM_SYSMENU_STAY_ON_TOP,
        _T ("Stay on To&p"));
    return 0;
}

void CMainWindow::OnClose ()
{
    SaveWindowState ();
    KillTimer (ID_TIMER_CLOCK);
    CFrameWnd::OnClose ();
}

void CMainWindow::OnEndSession (BOOL bEnding)
{
    if (bEnding)
        SaveWindowState ();
    CFrameWnd::OnEndSession (bEnding);
```

```
}

void CMainWindow::OnGetMinMaxInfo (MINMAXINFO* pMMI)
{
    pMMI->ptMinTrackSize.x = 120;
    pMMI->ptMinTrackSize.y = 120;
}

UINT CMainWindow::OnNcHitTest (CPoint point)
{
    UINT nHitTest = CFrameWnd::OnNcHitTest (point);
    if ((nHitTest == HTCLIENT) && (::GetAsyncKeyState (MK_LBUTTON) < 0))
        nHitTest = HTCAPTION;
    return nHitTest;
}

void CMainWindow::OnSysCommand (UINT nID, LPARAM lParam)
{
    UINT nMaskedID = nID & 0xFFF0;

    if (nMaskedID == IDM_SYSMENU_FULL_WINDOW) {
        m_bFullWindow = m_bFullWindow ? 0 : 1;
        SetTitleBarState ();
        return;
    }
    else if (nMaskedID == IDM_SYSMENU_STAY_ON_TOP) {
        m_bStayOnTop = m_bStayOnTop ? 0 : 1;
        SetTopMostState ();
        return;
    }
    CFrameWnd::OnSysCommand (nID, lParam);
}

void CMainWindow::OnContextMenu (CWnd* pWnd, CPoint point)
{
    CRect rect;
    GetClientRect (&rect);
    ClientToScreen (&rect);
    if (rect.PtInRect (point)) {
        CMenu* pMenu = GetSystemMenu (FALSE);
        UpdateSystemMenu (pMenu);

        int nID = (int) pMenu->TrackPopupMenu (TPM_LEFTALIGN |
            TPM_LEFTBUTTON|TPM_RIGHTBUTTON|TPM_RETURNCMD, point.x,
            point.y, this);

        if (nID > 0)
```

```
            SendMessage (WM_SYSCOMMAND, nID, 0);

        return;
    }
    CFrameWnd::OnContextMenu (pWnd, point);
}

void CMainWindow::OnTimer (UINT nTimerID)
{
    //
    // Do nothing if the window is minimized.
    //
    if (IsIconic ())
        return;

    //
    // Get the current time and do nothing if it hasn't changed.
    //
    CTime time = CTime::GetCurrentTime ();
    int nSecond = time.GetSecond ();
    int nMinute = time.GetMinute ();
    int nHour = time.GetHour () % 12;

    if ((nSecond == m_nPrevSecond) &&
        (nMinute == m_nPrevMinute) &&
        (nHour == m_nPrevHour))
        return;

    //
    // Center the origin and switch to the MM_ISOTROPIC mapping mode.
    //
    CRect rect;
    GetClientRect (&rect);

    CClientDC dc (this);
    dc.SetMapMode (MM_ISOTROPIC);
    dc.SetWindowExt (1000, 1000);
    dc.SetViewportExt (rect.Width (), -rect.Height ());
    dc.SetViewportOrg (rect.Width () / 2, rect.Height () / 2);

    //
    // If minutes have changed, erase the hour and minute hands.
    //
    COLORREF clrColor = ::GetSysColor (COLOR_3DFACE);

    if (nMinute != m_nPrevMinute) {
        DrawHand (&dc, 200, 4, (m_nPrevHour * 30) + (m_nPrevMinute / 2),
```

```
            clrColor);
        DrawHand (&dc, 400, 8, m_nPrevMinute * 6, clrColor);
        m_nPrevMinute = nMinute;
        m_nPrevHour = nHour;
    }

    //
    // If seconds have changed, erase the second hand and redraw all hands.
    //
    if (nSecond != m_nPrevSecond) {
        DrawSecondHand (&dc, 400, 8, m_nPrevSecond * 6, clrColor);
        DrawSecondHand (&dc, 400, 8, nSecond * 6, RGB (0, 0, 0));
        DrawHand (&dc, 200, 4, (nHour * 30) + (nMinute / 2),
            RGB (0, 0, 0));
        DrawHand (&dc, 400, 8, nMinute * 6, RGB (0, 0, 0));
        m_nPrevSecond = nSecond;
    }
}

void CMainWindow::OnPaint ()
{
    CRect rect;
    GetClientRect (&rect);

    CPaintDC dc (this);
    dc.SetMapMode (MM_ISOTROPIC);
    dc.SetWindowExt (1000, 1000);
    dc.SetViewportExt (rect.Width (), -rect.Height ());
    dc.SetViewportOrg (rect.Width () / 2, rect.Height () / 2);

    DrawClockFace (&dc);
    DrawHand (&dc, 200, 4, (m_nPrevHour * 30) +
        (m_nPrevMinute/2), RGB (0, 0, 0));
    DrawHand (&dc, 400, 8, m_nPrevMinute * 6, RGB (0, 0, 0));
    DrawSecondHand (&dc, 400, 8, m_nPrevSecond * 6, RGB (0, 0, 0));
}
void CMainWindow::DrawClockFace (CDC * pDC)
{
    static CPoint point[12] = {
        CPoint (   0,  450),      // 12 o'clock
        CPoint ( 225,  390),      //  1 o'clock
        CPoint ( 390,  225),      //  2 o'clock
        CPoint ( 450,    0),      //  3 o'clock
        CPoint ( 390, -225),      //  4 o'clock
        CPoint ( 225,  390),      //  5 o'clock
        CPoint (   0, -450),      //  6 o'clock
        CPoint (-225, -390),      //  7 o'clock
```

```
        CPoint (-390, -225),    //  8 o'clock
        CPoint (-450,    0),    //  9 o'clock
        CPoint (-390,  225),    // 10 o'clock
        CPoint (-225,  390),    // 11 o'clock
    };

    pDC -> SelectStockObject (NULL_BRUSH);

    for (int i = 0; i < 12; i ++)
        pDC -> Rectangle (point[i].x - SQUARESIZE,
            point[i].y + SQUARESIZE, point[i].x + SQUARESIZE,
            point[i].y - SQUARESIZE);
}

void CMainWindow::DrawHand (CDC * pDC, int nLength, int nScale,
    int nDegrees, COLORREF clrColor)
{
    CPoint point[4];
    double nRadians = (double) nDegrees * 0.017453292;

    point[0].x = (int) (nLength * sin (nRadians));
    point[0].y = (int) (nLength * cos (nRadians));

    point[2].x = -point[0].x / nScale;
    point[2].y = -point[0].y / nScale;

    point[1].x = -point[2].y;
    point[1].y = point[2].x;

    point[3].x = -point[1].x;
    point[3].y = -point[1].y;

    CPen pen (PS_SOLID, 0, clrColor);
    CPen * pOldPen = pDC -> SelectObject (&pen);
    pDC -> MoveTo (point[0]);
    pDC -> LineTo (point[1]);
    pDC -> LineTo (point[2]);
    pDC -> LineTo (point[3]);
    pDC -> LineTo (point[0]);

    pDC -> SelectObject (pOldPen);
}

void CMainWindow::DrawSecondHand (CDC * pDC, int nLength, int nScale,
    int nDegrees, COLORREF clrColor)
{
    CPoint point[2];
    double nRadians = (double) nDegrees * 0.017453292;

    point[0].x = (int) (nLength * sin (nRadians));
```

```
    point[0].y = (int) (nLength * cos (nRadians));

    point[1].x = -point[0].x / nScale;
    point[1].y = point[0].y / nScale;

    CPen pen (PS_SOLID, 0, clrColor);
    CPen * pOldPen = pDC -> SelectObject (&pen);

    pDC -> MoveTo (point[0]);
    pDC -> LineTo (point[1]);

    pDC -> SelectObject (pOldPen);
}

void CMainWindow::SetTitleBarState ()
{
    CMenu * pMenu = GetSystemMenu (FALSE);

    if (m_bFullWindow) {
        ModifyStyle (WS_CAPTION, 0);
        pMenu -> ModifyMenu (IDM_SYSMENU_FULL_WINDOW, MF_STRING,
            IDM_SYSMENU_FULL_WINDOW, _T ("Restore &Title"));
    }
    else {
        ModifyStyle (0, WS_CAPTION);
        pMenu -> ModifyMenu (IDM_SYSMENU_FULL_WINDOW, MF_STRING,
            IDM_SYSMENU_FULL_WINDOW, _T ("Remove &Title"));
    }
    SetWindowPos (NULL, 0, 0, 0, 0, SWP_NOMOVE|SWP_NOSIZE |
        SWP_NOZORDER|SWP_DRAWFRAME);
}

void CMainWindow::SetTopMostState ()
{
    CMenu * pMenu = GetSystemMenu (FALSE);

    if (m_bStayOnTop) {
        SetWindowPos (&wndTopMost, 0, 0, 0, 0, SWP_NOMOVE|SWP_NOSIZE);
        pMenu -> CheckMenuItem (IDM_SYSMENU_STAY_ON_TOP, MF_CHECKED);
    }
    else {
        SetWindowPos (&wndNoTopMost, 0, 0, 0, 0, SWP_NOMOVE|SWP_NOSIZE);
        pMenu -> CheckMenuItem (IDM_SYSMENU_STAY_ON_TOP, MF_UNCHECKED);
    }


BOOL CMainWindow::RestoreWindowState ()
{
    CString version = _T ("Version 1.0");
```

```
    m_bFullWindow = myApp.GetProfileInt (version, _T ("FullWindow"), 0);
    SetTitleBarState ();
    m_bStayOnTop = myApp.GetProfileInt (version,_T ("StayOnTop"), 0);
    SetTopMostState ();

    WINDOWPLACEMENT wp;
    wp.length = sizeof (WINDOWPLACEMENT);
    GetWindowPlacement (&wp);

    if (((wp.flags =
            myApp.GetProfileInt (version,_T ("flags"), -1)) != -1) &&
        ((wp.showCmd =
            myApp.GetProfileInt (version,_T ("showCmd"), -1)) != -1) &&
        ((wp.rcNormalPosition.left =
            myApp.GetProfileInt (version,_T ("x1"), -1)) != -1) &&
        ((wp.rcNormalPosition.top =
            myApp.GetProfileInt (version,_T ("y1"), -1)) != -1) &&
        ((wp.rcNormalPosition.right =
            myApp.GetProfileInt (version,_T ("x2"), -1)) != -1) &&
        ((wp.rcNormalPosition.bottom =
            myApp.GetProfileInt (version,_T ("y2"), -1)) != -1)) {

        wp.rcNormalPosition.left = min (wp.rcNormalPosition.left,
            ::GetSystemMetrics (SM_CXSCREEN) -
            ::GetSystemMetrics (SM_CXICON));
        wp.rcNormalPosition.top = min (wp.rcNormalPosition.top,
            ::GetSystemMetrics (SM_CYSCREEN) -
            ::GetSystemMetrics (SM_CYICON));

        SetWindowPlacement (&wp);
        return TRUE;
    }
    return FALSE;
}

void CMainWindow::SaveWindowState ()
{
    CString version = _T ("Version 1.0");
    myApp.WriteProfileInt (version, _T ("FullWindow"), m_bFullWindow);
    myApp.WriteProfileInt (version, _T ("StayOnTop"), m_bStayOnTop);

    WINDOWPLACEMENT wp;
    wp.length = sizeof (WINDOWPLACEMENT);
    GetWindowPlacement (&wp);

    myApp.WriteProfileInt (version, _T ("flags"), wp.flags);
    myApp.WriteProfileInt (version, _T ("showCmd"), wp.showCmd);
    myApp.WriteProfileInt (version,_T ("x1"), wp.rcNormalPosition.left);
```

```
    myApp.WriteProfileInt (version,_T ("y1"), wp.rcNormalPosition.top);
    myApp.WriteProfileInt (version,_T ("x2"), wp.rcNormalPosition.right);
    myApp.WriteProfileInt (version,_T ("y2"), wp.rcNormalPosition.bottom);
}

void CMainWindow::UpdateSystemMenu (CMenu * pMenu)
{
    static UINT nState[2][5] = {
        { MFS_GRAYED,  MFS_ENABLED, MFS_ENABLED,
          MFS_ENABLED, MFS_DEFAULT },
        { MFS_DEFAULT, MFS_GRAYED,  MFS_GRAYED,
          MFS_ENABLED, MFS_GRAYED  }
    };

    if (IsIconic ()) // Shouldn't happen, but let's be safe
        return;

    int i = 0;
    if (IsZoomed ())
        i = 1;

    CString strMenuText;
    pMenu -> GetMenuString (SC_RESTORE, strMenuText, MF_BYCOMMAND);
    pMenu -> ModifyMenu (SC_RESTORE, MF_STRING|nState[i][0], SC_RESTORE,
        strMenuText);

    pMenu -> GetMenuString (SC_MOVE, strMenuText, MF_BYCOMMAND);
    pMenu -> ModifyMenu (SC_MOVE, MF_STRING nState[i][1], SC_MOVE,
        strMenuText);

    pMenu -> GetMenuString (SC_SIZE, strMenuText, MF_BYCOMMAND);
    pMenu -> ModifyMenu (SC_SIZE, MF_STRING nState[i][2], SC_SIZE,
        strMenuText);

    pMenu -> GetMenuString (SC_MINIMIZE, strMenuText, MF_BYCOMMAND);
    pMenu -> ModifyMenu (SC_MINIMIZE, MF_STRING|nState[i][3], SC_MINIMIZE,
        strMenuText);

    pMenu -> GetMenuString (SC_MAXIMIZE, strMenuText, MF_BYCOMMAND);
    pMenu -> ModifyMenu (SC_MAXIMIZE, MF_STRING|nState[i][4], SC_MAXIMIZE,
        strMenuText);

    SetMenuDefaultItem (pMenu -> m_hMenu, i ? SC_RESTORE :
        SC_MAXIMIZE, FALSE);
}
```

图 14-2　Clock 应用程序

14.2.1 处理计时器消息

Clock 使用 SetTimer 在 OnCreate 中设置计时器。在 OnClose 中调用 KillTimer 清除计时器。当 WM_TIMER 消息到达之后,CMainWindow::OnTimer 就得到当前的时间,它将时、分和秒分别与在成员变量 m_nPrevHour、m_nPrevMinute 以及 m_nPrevSecond 中记录的时、分和秒进行比较。如果当前的时、分和秒与以前记录的时、分以及秒相同,OnTimer 不执行任何操作。否则,它将记录新的时间并移动 Clock 的指针。CMainWindow::DrawHand 绘制时针和分针,CMainWindow::DrawSecondHand 绘制秒针。通过两次调用对应的绘制函数可以实现移动指针:一次是用窗口背景颜色(COLOR_3DFACE)重绘指针而清除它,一次是用黑色在新位置处绘制。

由于使用了此 OnTimer 机制,时钟的秒针大概每秒移动一次,而时针和分针在当前经过一小时的分钟数与上次记录的经过一小时的分钟数不同时就移动一次。因为绘制指针是为了反映当前时间而不是反映基于接收到的 WM_TIMER 消息数量的假定时间,所以在窗口被拖动或缩放时如果跳过了 WM_TIMER 消息也是无关紧要的。如果您仔细看,偶尔会发现秒针向前移动了两秒而不是一秒。这是因为偶尔 WM_TIMER 消息会在新的一秒快要走完之前到达而下一个 WM_TIMER 消息在隔开一秒后的下一个新的一秒中到达。您可以减小计时器的时间间隔来防止发生这种现象,如减少到 0.5 秒就可以。而这样做的代价是增加了系统总开销,但是额外的开销将是很小的,因为 OnTimer 的构造就使得只有当上次计时器消息到达之后设置的时间被修改时它才重绘 Clock 指针。

在执行其他任何操作之前,OnTimer 要调用主窗口的 IsIconic 函数来确定窗口当前是否已最小化。对于最小化了的窗口 IsIconic 返回非零值,对未最小化的窗口返回 0(补充函数 CWnd::IsZoomed,如果窗口已最大化则返回非零值,否则返回 0。)。如果 IsIconic 返回非零值,OnTimer 会很快退出以防止在窗口未显示时更新时钟显示。在 Windows 95 及其更高版本或 Windows NT 4.0 及其更高版本中,当最小化窗口调用 GetClientRect 时,返回的矩形是坐标值等于 0 的 NULL 矩形。应用程序可能尝试在这个矩形中绘制输出,但 GDI 将把输出剪切掉。对于计时器每次走动都检查窗口是否最小化可以省去不必要的绘图从而减少了 CPU 的负担。

如果更希望在它的窗口最小化时 Clock 不闲置在那里,可以试着重写 OnTimer 函数的开头部分,如下:

```
CTime time = CTime::GetCurrentTime();
int nSecond = time.GetSecond();
int nMinute = time.GetMinute();
int nHour = time.GetHour() % 12;

if (IsIconic()) {
    CString time;
```

```
        time.Format (_T ("%0.2d: %0.2d: %0.2d"), nHour, nMinute, nSecond);
        SetWindowText (time);
        return;
    }

    else {
        SetWindowText (_T ("Clock"));
          .
          .
          .
    }
```

应用程序通过使用 CWnd::SetWindowText 修改窗口标题来修改在任务栏中靠着其图标显示的文本。如果按上例进行了修改，当最小化时，Clock 会在任务栏显示时钟的走动。

14.2.2　获得当前时间：CTime 类

要查询系统当前时间，Clock 使用了 CTime 对象。CTime 是表示时间和日期的一个 MFC 类。它包括一些便于使用的成员函数，可以用来获取日期、时间、一周中各天（星期天、星期一、星期二等等）以及其他信息。重载后的运算符如 +、? 和 > 允许用简单的整数操作时间和日期。

我们感兴趣的 CTime 成员函数是 GetCurrentTime，它是一个静态函数，返回用当前日期和时间初始化了的 CTime 对象；GetHour 返回小时（从 0 到 23）；GetMinute 返回分钟数（从 0 到 59）；GetSecond 返回秒数（从 0 到 59）。OnTimer 使用下列语句检索当前的小时、分钟和秒，以便确定时钟显示是否需要更新：

```
CTime time = CTime::GetCurrentTime ();
int nSecond = time.GetSecond ();
int nMinute = time.GetMinute ();
int nHour = time.GetHour () % 12;
```

对于 GetHour 的返回值以 12 取模的操作将小时值变换为从 0 到 11 的整数。CMainWindow 的构造函数使用类似的程序代码初始化 m_nPrevHour、m_nPrevMinute 以及 m_nPrevSecond。

14.2.3　使用 MM_ISOTROPIC 映射方式

直到现在为止，我们开发的大多数应用程序都使用默认的 MM_TEXT 映射方式。映射方式管理着 Windows 怎样将传递给 CDC 绘图函数的逻辑单位转换成屏幕上的设备单位（像素）。在 MM_TEXT 映射方式下，逻辑单位和设备单位是一致的，因此如果应用程序绘制一条从 (0,0) 到 (50,100) 的线段，线段就会从显示平面的左上角像素处延伸到距左上角向右 50 像素向下 100 像素的地方。当然前提假定是绘图原点仍旧在默认的显示平面的左上角

没有改变。

MM_TEXT 对大多数应用程序都适合,但还可以使用其他 GDI 映射方式来减少应用程序对显示设备物理特性的依赖(参考第 2 章复习一下 GDI 映射方式。)。例如:在 MM_LOENGLISH映射方式下,一个逻辑单位等于 1 英寸的 1/100,因此如果想要绘制精确的 1 英寸长的线段,可以使用 100 单位长度,而 Windows 在将线段扫描变换为像素时会把比例因素计算在内而得到每英寸中的像素总数。对于屏幕 DC 转换可能不完美,因为对于屏幕,Windows使用假定的每英寸像素值,它并不基于物理屏幕的大小。但是,对于打印机和其他硬拷贝设备,Windows 能够获得精确的每英寸像素值,因此在打印机输出中使用 MM_LOENGLISH,确实可以绘制 1 英寸长的线段。

Clock 使用了 MM_ISOTROPIC 映射方式,其中沿着 x 轴测量的逻辑单位具有与沿着 y 轴测量的逻辑单位相同的物理尺寸。在响应 WM_TIMER 或 WM_PAINT 消息而绘制时钟的表面和指针之前,Clock 用 GetClientRect 测量了窗口的客户区并创建了一个设备描述表。然后它将映射方式设置为 MM_ISOTROPIC,移动坐标系的原点使逻辑点 (0,0) 位于窗口客户区的中心,并设置窗口范围使窗口客户区在每个方向上都具有 1 000 个逻辑单位。程序代码如下:

```
CRect rect;
GetClientRect (&rect);

CClientDC dc (this); // In OnPaint, use CPaintDC instead.
dc.SetMapMode (MM_ISOTROPIC);
dc.SetWindowExt (1000, 1000);
dc.SetViewportExt (rect.Width (), -rect.Height ());
dc.SetViewportOrg (rect.Width () / 2, rect.Height () / 2);
```

传递给 SetViewportExt 的负值指定了视口的物理高度,确定坐标系的方向为 y 的增量方向是向上的方向。如果负符号被省略,y 的增加值将向屏幕下方移动而不是向上,因为 Windows 对位于屏幕底端的像素编号要大于对顶端像素的编号。图 14-3 显示了变换以后坐标系的样子。坐标系位于窗口客户区的中间,向右和向上移动时 x 和 y 值将相应增加。这样就形成一个四象限笛卡儿坐标系,碰巧可作为绘制模拟时钟表面的非常便于使用的模型。

一旦用此方法设置了坐标系,您就可以不考虑窗口的物理尺寸而编写绘制时钟表面和指针的例程了。当调用 DrawHand 绘制时钟指针时,第二个参数传递的是长度值,对于时针为 200,对于分针为 400。DrawSecondHand 中传递的秒针长度值也为 400。因为从坐标系的原点到窗口的任何一边的距离都是 500 逻辑单位,所以分针和秒针的长度就为到最近边距离的百分之八十,时针为百分之四十。如果使用了 MM_TEXT 映射方式,就必须在按比例调整了每个坐标值和距离值以后才能把它们传递给 GDI。

14.2.4 隐藏和显示标题栏

Clock 的系统菜单包含了两个额外的命令:Remove Title 和 Stay On Top。RemoveTitle 命令

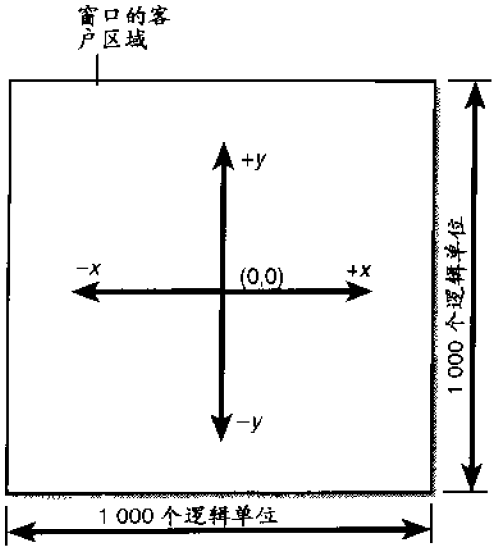

图 14-3　Clock 的屏幕输出坐标系

删除窗口的标题使得时钟表面填充全部窗口。通过再次显示系统菜单并选中 Restore Title 命令可以还原标题栏，Restore Title 命令出现在以前 Remove Title 命令所在的地方。这种奇妙的切换技术其实很简单，但是即使有经验的 Windows 程序员初次遇到任意地添加和删除标题栏的情况也会感到困惑。

秘密就在 CMainWindow::SetTitleBarState 中。决定窗口是否具有标题栏的属性是 WS_CAPTION 样式位，它包含在大多数框架窗口使用的 WS_OVERLAPPEDWINDOW 样式中。创建没有标题栏的窗口很简单，只要省略 WS_CAPTION 即可。所以通过清除 WS_CAPTION 位就可以从已存在的窗口中删除标题栏。MFC 的 CWnd::ModifyStyle 函数通过简单的函数调用就可以修改窗口样式。在 Clock 的系统菜单中选中 Remove/Restore Title 命令后，CMainWindow::OnSysCommand 将保存在 CMainWindow::m_bFullWindow 中的值从 1 切换到 0，或从 0 切换到 1，然后调用 CMainWindow::SetTitleBarState，该函数根据 m_bFullWindow 中的当前值再增加或删除 WS_CAPTION：

```
if (m_bFullWindow) {
    ModifyStyle(WS_CAPTION, 0);
    pMenu->ModifyMenu(IDM_SYSMENU_FULL_WINDOW, MF_STRING,
        IDM_SYSMENU_FULL_WINDOW, _T("Restore &Title"));
}
else {
    ModifyStyle(0, WS_CAPTION);
    pMenu->ModifyMenu(IDM_SYSMENU_FULL_WINDOW, MF_STRING,
        IDM_SYSMENU_FULL_WINDOW, _T("Remove &Title"));
}
```

传递给 ModifyStyle 的第 1 个参数指定了要删除的样式,第 2 个参数指定了要添加的样式。SetTitleBarState 还设置了菜单项文本与样式位的状态匹配:如果标题栏被显示则为"Remove Title",否则为"Restore Title"。

然而,切换 WS_CAPTION 开关状态仅仅是工作的一半。关键技巧在于窗口样式修改之后怎样重新绘制窗口的非客户区。调用 CWnd::Invalidate 不行,要用 SWP_DRAWFRAME 参数调用 SetWindowPos 才可以:

```
SetWindowPos (NULL, 0 0, 0, 0, SWP_NOMOVE|SWP_NOSIZE|
  SWP_NOZORDER SWP_DRAWFRAME);
```

SetWindowPos 和 SWP_DRAWFRAME 的组合使得整个窗口包括标题栏被重绘。其他传递给 SetWindowPos 的 SWP 标志保存了窗口的位置、大小以及 z 向上的顺序位置,该位置是从前向后的窗口位置,决定所要绘制的在其他窗口上的窗口。

14.2.5　实现客户区拖动

对于没有标题栏的窗口,有一个问题那就是无法用鼠标改变它的位置。窗口是通过它们的标题栏来拖动的,如果没有标题栏,用户就没法抓住它们。Clock 解决了这个小困难,它对窗口的 WM_NCHITTEST 施展了一点手段,使得窗口可以通过它的客户区被拖动,这一功能 Windows 程序员叫做"客户区拖动"。

在 Windows 中,每个鼠标消息之前都有一个 WM_NCHITTEST 消息,它带有屏幕坐标用以标识光标的位置。消息通常由 ::DefWindowProc 处理,它返回一个代码告诉 Windows 光标位于窗口的哪个部分。Windows 使用返回值来确定要发送什么类型的鼠标消息。例如:如果在窗口的标题栏上单击鼠标左键,::DefWindowProc 的 WM_NCHITTEST 处理程序将返回HTCAPTION,并且 Windows 就会给窗口发送一个 WM_NCLBUTTONDOWN 消息。如果 ::DefWindowProc 返回的是 HTCLIENT,Windows 就会将光标坐标从平面坐标转换到客户区坐标,并且在 WM_LBUTTONDOWN 消息中把它们传递给窗口。

应用程序是以原始的形式看待鼠标消息的,这使得一些有趣的操作有了实现的可能性。下面的 OnNcHitTest 处理程序实现了客户区拖动功能,它使 Windows 误以为鼠标是在标题栏上而事实上它是在窗口的客户区中:

```
UINT CMainWindow::OnNcHitTest (CPoint point)
{
  UINT nHitTest = CFrameWnd::OnNcHitTest (point);
  if (nHitTest == HTCLIENT)
      nHitTest = HTCAPTION;
  return nHitTest;
}
```

使用了这个 OnNcHitTest 处理程序,通过客户区拖动窗口就和通过标题栏拖动它一样简单

了。而且就算窗口没有标题栏，它也照常工作。试一下：在 Clock 的客户区单击鼠标左键，并保持按下状态移动鼠标，窗口就会跟着鼠标到处移动。

Clock 使用的 OnNcHitTest 处理程序与上面给出的程序相似，唯一不同之处在于，Clock 在用 HTCAPTION 代替 HTCLIENT 返回代码之前，要验证是鼠标左键被按下了，以便其他鼠标消息（特别是在 WM_CONTEXTMENU 消息之前的鼠标右键消息）不会受到影响：

```
UINT CMainWindow::OnNcHitTest (CPoint point)
{
  UINT nHitTest = CFrameWnd::OnNcHitTest (point);
  if ((nHitTest == HTCLIENT) &&
      (::GetAsyncKeyState (VK_LBUTTON) < 0))
      nHitTest = HTCAPTION;
      return nHitTest;
}
```

调用 ::GetAsyncKeyState 来检查鼠标左键，如果当前被按下则返回一个负值。

14.2.6　使用系统菜单作为上下文菜单

删除窗口的标题栏还有其他的含意。没有标题栏，用户就没有办法显示系统菜单，从而使标题栏被还原。Clock 的解决方法是使用 OnContextMenu 处理程序，当在窗口的客户区中单击鼠标右键，它将以上下文菜单的形式显示系统菜单。在任意位置弹出系统菜单说来容易做起来难，因为没有便于使用的 API 函数可以用来自动显示系统菜单。Clock 说明了一种可供使用的技术。当用鼠标右键单击 Clock 的客户区时，CMainWindow 的 OnContextMenu 处理程序就用 GetSystemMenu 检索指向系统菜单的 CMenu 指针，并用 CMenu::TrackPopupMenu 来显示菜单：

```
CMenu* pMenu = GetSystemMenu (FALSE);
 .
 .
 .
int nID = (int) pMenu->TrackPopupMenu (TPM_LEFTALIGN |
  TPM_LEFTBUTTON | TPM_RIGHTBUTTON | TPM_RETURNCMD, point.x,
  point.y, this);
```

使用这种方法存在一个问题：从菜单选择的命令会生产 WM_COMMAND 消息而不是 WM_SYSCOMMAND 消息。作为补偿，Clock 把 TPM_RETURNCMD 标志传递给 TrackPopupMenu，指示它返回选中的菜单项 ID。如果 TrackPopupMenu 返回正的非零值，表明有菜单项被选中，Clock 将给自己发送一个 WM_SYSCOMMAND 消息，其中 wParam 等于菜单项 ID，如下：

```
if (nID > 0)
   SendMessage (WM_SYSCOMMAND, nID, 0);
```

因此,就会调用 OnSysCommand 来处理从假系统菜单内选中的命令,就好像是从真的系统菜单选中的那样。为防止主框架因为缺少 ON_COMMAND 处理程序而使添加到系统菜单中的项目无效,CMainWindow 的构造函数将 m_bAutoMenuEnable 设置为 FALSE。通常,主框架的自动使菜单项有效和无效的功能不会对添加到系统菜单中的项目产生影响,但是 Clock 的系统菜单是一个例外,因为用 TrackPopupMenu 显示时它被当作了常规菜单。

到目前为止一切顺利。可是我们还有一个问题没解决。Windows 交互地使系统菜单中的某个命令有效和无效,以便供选的命令与窗口状态一致。例如:在最大化窗口的系统菜单中,Move、Size 和 Maximize 命令变灰,而 Restore 和 Minimize 命令未变灰。如果同一个窗口恢复到它的未最大化状态后,Restore 命令就会变灰,其他所有命令处于有效状态。不幸的是,当您用 GetSystemMenu 得到菜单指针时,菜单项还没得到更新。因此,OnContextMenu 调用了 CMainWindow 函数 UpdateSystemMenu,根据窗口的现行状态来手工更新菜单项状态。在 UpdateSystemMenu 通过在 CMenu::GetMenuString 和 CMenu::ModifyMenu 中进行一系列函数调用而更新了系统菜单之后,它使用 ::SetMenuDefaultItem API 函数根据窗口状态将默认菜单项(用黑体字类型显示)设置为 Restore 或 Maximize。使用 UpdateSystemMenu 并不是理想办法,但是它能够完成工作,实际上我还没有找到更好的方法来维持程序自动显示的系统菜单中的菜单项与菜单所属的窗口状态同步。

14.2.7 最顶层窗口

Windows 3.1 引入的一项创新内容是"最顶层窗口"的概念,该窗口在 z 方向的顺序位置隐含地高于那些常规(或称为非最顶层)窗口。通常,处于 z 方向顶端的窗口被绘制在其他窗口上,在 z 向顺序中第二个窗口被绘制在除了第一个以外的其他窗口上,等等。而且最顶层窗口接收的优先级也高于其他窗口,就算它在 z 向的底端也不会被遮住。即使它在后台中运行,它也总是可见的。

Windows 任务栏是最顶层窗口的一个非常好的例子。在默认状态下,任务栏被指定为最顶层窗口,以便绘制在其他窗口上。如果同时显示两个(或更多)最顶层窗口,就用 z 向顺序标准规则来决定它们彼此之间的可见性。应该节约使用最顶层窗口,因为如果所有的窗口都是最顶层窗口,那么最顶层窗口将不再具有优先于其他窗口的意义了。

最顶层窗口和非最顶层窗口之间的不同之处在于一个扩展窗口样式位。WS_EX_TOPMOST 使一个窗口成为最顶层窗口,可以在调用 Create 中包括一个 WS_EX_TOPMOST 标志来创建最顶层框架窗口,如下:

```
Create (NULL, _T ("MyWindow"), WS_OVERLAPPEDWINDOW, rectDefault,
  NULL, NULL, WS_EX_TOPMOST);
```

另一种方法是在窗口创建后用 &wndTopMost 参数调用 SetWindowPos 来添加样式位,如下:

```
SetWindowPos (&wndTopMost, 0, 0, 0, 0, SWP_NOMOVE|SWP_NOSIZE);
```

如果 SetWindowPos 的第一个参数是 &wndNoTopMost 而不是 &wndTopMost，最顶层窗口则变成非最顶层窗口。

Clock 使用 SetWindowPos 来在系统菜单中选中 Stay On Top 命令后使其窗口成为最顶层窗口，而在 Stay On Top 未选中时使窗口成为非最顶层窗口。工作由 CMainWindow::SetTopMostState 完成，它是由 OnSysCommand 调用的。在 Stay On Top 被选中时，任何时候 Clock 在屏幕都是可见的，就算它在后台运行也会遮盖在前台运行的应用程序。

14.2.8　保留配置设置

Clock 是目前创建的第一个这样的应用程序，它将对程序的设置记录在磁盘中从而维持了它的持久性。“持久性”这个词在 Windows 程序设计中包含许多内容。说某信息具有持久性就意味着在整个会话过程中被保留。如果希望 Clock 在屏幕右下角运行于小窗口中，您可以调整它的大小并把它放在那里，下次启动后，它会自动地以相同大小和位置显示。对于喜欢以固定方式安排桌面的用户，类似这样的一点小功能就可以区分出好的和非常好的应用程序。其他的 Clock 配置设置也可以保留，如标题栏和处于最上层状态。

在整个会话过程中保留配置信息的关键是把它存储在硬盘上，以便在下次应用程序启动时可以再次读取回来。在 16 位 Windows 中，应用程序通常使用 ::WriteProfileString、::GetProfileString 和其他 API 函数在 Win.ini 或私有 INI 文件中存储配置设置。在 32 位 Windows 中，仍然支持 INI 文件以维持向下兼容性，但是并不鼓励程序员使用它们。32 位应用程序应该在注册表中存储配置设置。

注册表是一个二进制数据库，作为操作系统和它宿主的应用程序的中央数据仓库。存储在注册表中的信息是按分层结构组织的，使用了键和子键的体系，与硬盘上的目录和子目录结构类似。像目录可以包含文件那样，键可以包含数据输入项。数据输入项有名字并且可以给它们分配文本或二进制的值。在注册表分层结构的最上层是一组 6 个根键，分别为 HKEY_CLASSES_ROOT、HKEY_USERS、HKEY_CURRENT_USER、HKEY_LOCAL_MACHINE、HKEY_CURRENT_CONFIG 以及 HKEY_DYN_DATA。按照 Microsoft 的建议，Windows 应用程序应该在

```
HKEY_CURRENT_USER\Software\CompanyName\ProductName\Version
```

键下存储私有配置设置，其中 CompanyName 是公司名称；ProductName 是产品名称；Version 是版本编号。作为 WinWidgets，Inc 公司的一个产品，WidgetMaster 的 2.0 版本用于记录用户可选窗口背景颜色的注册表输入项如下：

```
HKEY_CURRENT_USER\Software\WinWidgets, Inc.\WidgetMaster\Version 2.0\BkgndColor = 4
```

因为信息是存储在 HKEY_CURRENT_USER 下面，所以它是针对每个用户而维持的。也就

是说,如果另一个用户注册并运行了同一应用程序,但是选择不同的背景色,那么就会为该用户记录一个独自的 BkgndColor 值。

Win32 API 中包括多种读取和写入注册表的函数,但是 MFC 在 API 基础上又提供了一层功能,使得读取和写入应用程序特定的注册表值与使用平常的 INI 文件没有什么不同。用注册表键的名字调用 CWinApp::SetRegistryKey 可以指导主框架使用注册表而不是使用 INI 文件。传递给 SetRegistryKey 的键名与公司名称一致,例如在上例中是"WinWidgets, Inc"。字符串和数值用 CWinApp 的 WriteProfileString 和 WriteProfileInt 函数写入注册表,而用 GetProfileString 和 GetProfileInt 读取回来。在名为 MyWord.exe 的应用程序中,语句

```
SetRegistryKey (_T ("WordSmith"));
WriteProfileInt (_T ("Version 1.0"),_T ("MRULength"), 8);
```

创建了下列数字注册表输入项:

```
HKEY_CURRENT_USER\Software\WordSmith\MYWORD\Version 1.0\MRULength = 8
```

语句

```
m_nMRULength = GetProfileInt (_T ("Version 1.0"),_T ("MRULength"), 4);
```

将它读取回来而且如果输入项不存在则返回 4。注意,MFC 通过从可执行文件名中去掉.exe 扩展名为您的产品生成了名称。

在它终止以前,Clock 在注册表中记录了下列配置设置:

- CMainWindow::m_bFullWindow 的值,表明是否显示标题栏
- CMainWindow::m_bStayOnTop 的值,表明是否选中 Stay On Top
- 框架窗口的大小和位置

下次启动时,Clock 将设置读取回来。图 14-4 给出了 Clock 保存在注册表中输入项的完整补码。CMainWindow 函数 SaveWindowState 和 RestoreWindowState 执行读取和写入。SaveWindowState 由窗口的 OnClose 和 OnEndSession 处理程序调用,而这两个处理程序分别在应用程序结束和 Windows 关机前执行。如果 Windows 被关闭,正在运行的应用程序不会接收到 WM_CLOSE 消息,它会接收 WM_ENDSESSION 消息。如果想要知道 Windows 是否正在进行关机准备,可以简单地把 ON_WM_ENDSESSION 输入项加到主窗口的消息映射表中,并编写一个相应的 OnEndSession 处理程序即可。传递给 OnEndSession 的 bEnding 参数表明 Windows 实际上是否正在关闭系统。非零值意味着是,而 0 意味着 Windows 即将关闭但操作被另一个应用程序禁止了。WM_ENDSESSION 消息之前是 WM_QUERYENDSESSION 消息,这样就给每个应用程序都提供了允许或禁止即将关机操作的机会。

Clock 的标题栏和位于最顶层设置用 SaveWindowState 中的下列语句保存在注册表的

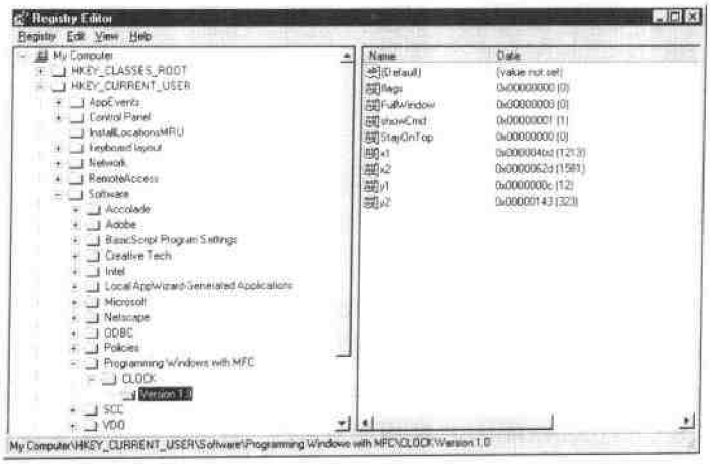

图 14-4　在 Registry Editor (RegEdit.exe) 中看到的 Clock 的注册表输入项

HKEY_CURRENT_USER \Software \Programming with MFC \CLOCK \Version 1.0 分支中了。

```
CString version = _T ("Version 1.0");
myApp.WriteProfileInt (version,_T ("FullWindow"), m_bFullWindow);
myApp.WriteProfileInt (version,_T ("StayOnTop"), m_bStayOnTop);
```

在 RestoreWindowState 中设置被读回并应用到窗口上：

```
CString version = _T ("Version 1.0");
m_bFullWindow = myApp.GetProfileInt (version,_T ("FullWindow"), 0);
SetTitleBarState ();
m_bStayOnTop = myApp.GetProfileInt (version,_T ("StayOnTop"), 0);
SetTopMostState ();
```

CMyApp::InitInstance 在窗口创建之后但还未在屏幕上显示之前调用 RestoreWindowState。

保存和恢复窗口的大小和位置更需要一点技巧。如果您从来没有编写过这样的应用程序，其窗口可以记忆自己的大小和位置，您或许会认为这很简单，只要保存由 CWnd::GetWindowRect 返回的坐标以便传递给 Create 或 CreateEx 即可。但实际上并非如此。如果您没有考虑窗口的当前状态(最小化、最大化或未最小也未最大化)，糟糕的事情都可能发生。例如：如果您将最大化窗口的坐标传递给了 Create 或 CreateEx，生成的窗口将会占据整个屏幕，可是它的标题栏会具有最大化框而不是恢复框。在最小化或最大化状态下关闭的持久性窗口在重新出现时还应该是最小化或最大化的，它还应该记住自己的标准尺寸，使得在恢复时可以回到从前的大小。

保留窗口的尺寸和位置并考虑到相关状态信息的关键就在于一对名为 GetWindowPlacement 和 SetWindowPlacement 的 CWnd 函数。每个函数都接受 WINDOWPLACEMENT 结构的地址，其定义如下：

```
typedef struct tagWINDOWPLACEMENT {
  UINT length;
  UINT flags;
  UINT showCmd;
  POINT ptMinPosition;
  POINT ptMaxPosition;
  RECT rcNormalPosition;
} WINDOWPLACEMENT;
```

WINDOWPLACEMENT 集中了 Windows 需要知道的所有说明窗口屏幕状态的信息。length 指定了 WINDOWPLACEMENT 结构的大小。CWnd::GetWindowPlacement 和 CWnd::SetWindowPlacement 都会读写此字段。flags 包含零个或多个标志用来指定有关最小化窗口的信息。WPF_RESTORETOMAXIMIZED 标志如果存在,则表明最小化窗口在恢复时会被最大化。showCmd 指定了窗口的当前显示状态。如果窗口为最小化,则设置为 SW_SHOWMINIMIZED;如果窗口是最大化的,则设置为 SW_SHOWMAXIMIZED;或者窗口既不是最小化也不是最大化时,设置为 SW_SHOWNORMAL。ptMinPosition 和 ptMaxPosition 分别保存了最小化和最大化窗口的左上角屏幕坐标(不要期望依靠 ptMinPosition 可以执行什么工作,现今 Windows 版本中窗口最小化时将 ptMinPosition 设置为了 (3000,3000))。rcNormalPosition 包含了窗口处于"普通"状态(未最小化也未最大化)时在屏幕上位置的屏幕坐标。如果窗口处于最小化或最大化状态,rcNormalPosition 就指定了窗口将恢复到的位置和大小,当然前提是没有设置 WPF_RESTORETOMAXIMIZED 标志来强迫恢复的窗口充满屏幕。

通过在窗口的 WINDOWPLACEMENT 结构中保存 flags、showCmd 和 rcNormalPosition 的值并在窗口重新创建时恢复它们,就可以实现在整个会话过程中保留窗口的屏幕状态。没必要保存 ptMinPosition 和 ptMaxPosition,因为在窗口最小化或最大化时 Windows 会填写它们的值。Clock 的 SaveWindowState 函数使用 GetWindowPlacement 来初始化 WINDOWPLACEMENT 结构并将结构的相关成员写入注册表中。窗口状态的恢复是 CMyApp::InitInstance 调用 CMainWindow::RestoreWindowState 实现的,该函数接下来调用 CetWindowPlacement 填写 WINDOWPLACEMENT 结构;从注册表中读取 flags、showCmd 和 rcNormalPosition 的值;把它们复制给结构体;并调用 SetWindowPlacement。在 showCmd 中传递给 SetWindowPlacement 的 SW_SHOWMINIMIZED、SW_SHOWMAXIMIZED 或 SW_SHOWNORMAL 参数使得窗口可见,因此,如果 RestoreWindowState 返回 TRUE 说明成功地恢复了窗口状态,那么就没必要调用 ShowWindow 了。实际上,如果 RestoreWindowState 返回 TRUE,您就应该从 InitInstance 中跳过通常对 ShowWindow 的调用,因为应用程序对象的 m_nCmdShow 参数可能已经修改了窗口状态。Clock 的 InitInstance 函数如下:

```
BOOL CMyApp::InitInstance ()
{
  SetRegistryKey (_T ("Programming Windows with MFC"));
```

```
    m_pMainWnd = new CMainWindow;
    if (!((CMainWindow*) m_pMainWnd)->RestoreWindowState())
        m_pMainWnd->ShowWindow(m_nCmdShow);
    m_pMainWnd->UpdateWindow();
    return TRUE;
}
```

在第一次运行 Clock 时，由于 RestoreWindowState 返回 FALSE 所以就按普通方式调用了 ShowWindow。在以后的调用中，RestoreWindowState 设置了窗口的尺寸、位置和可见性状态，这样 ShowWindow 就会被跳过去。

在调用 SetWindowPlacement 应用从注册表中得到的状态值以前，因为如果将窗口放置在具有 1 024 × 768 模式的屏幕边缘，当 Windows 以 640 × 480 或 800 × 600 模式重新启动时，窗口就有可能不会被显示出来，为避免发生类似情况，RestoreWindowState 将对窗口的普通位置和屏幕范围进行比较：

```
wp.rcNormalPosition.left = min(wp.rcNormalPosition.left,
  ::GetSystemMetrics(SM_CXSCREEN) -
  ::GetSystemMetrics(SM_CXICON));
wp.rcNormalPosition.top = min(wp.rcNormalPosition.top,
  ::GetSystemMetrics(SM_CYSCREEN)
  ::GetSystemMetrics(SM_CYICON));
```

用 SM_CXSCREEN 和 SM_CYSCREEN 参数调用 ::GetSystemMetrics 将分别以像素为单位返回屏幕的宽度和高度。如果从注册表检索到的窗口坐标为 700 和 600，而 Windows 正运行在分辨率为 640 × 480 模式下，这个简单的过程就会将 700 和 600 转换为 640 和 480 减去图标的宽度和高度。最后的输出结果是窗口不会被显示在屏幕以外而不可见，从而使得用户怀疑是否运行了应用程序，相反窗口会出现在屏幕的右下角。

测试可以保留窗口位置和大小的程序，最好的方法是以任意尺寸调整窗口的大小，把它最大化、最小化，然后在窗口最小化的状态下关闭应用程序。在程序重新启动时，窗口应该以最小化的形式出现。单击任务栏上最小化窗口的图标可以将它最大化。单击恢复按钮可以把它恢复到原始的尺寸和位置。用这种方法试验一下 Clock，您会发现它将出色地通过测试。

14.2.9　控制窗口大小：WM_GETMINMAXINFO 消息

Clock 中值得注意的最后一个方面是它的 OnGetMinMaxInfo 处理程序。在窗口被缩放时，它会接收到一系列 WM_GETMINMAXINFO 消息，其中的 lParam 指向 MINMAXINFO 结构，该结构保存着有关窗口最小和最大“跟踪”尺寸信息。可以通过编程来限制窗口的最小和最大尺寸，方法是处理 WM_GETMINMAXINFO 消息并将最小宽度和高度保存在结构的 ptMinTrackSize 字段中的 x 和 y 成员内，将最大宽度和高度保存在 ptMaxTrackSize 字段的 x 和

y 成员内。Clock 通过以下 OnGetMinMaxInfo 处理程序实现了禁止窗口的水平和垂直长度小于 120 像素：

```
void CMainWindow::OnGetMinMaxInfo (MINMAXINFO * pMMI)
{
 pMMI -> ptMinTrackSize.x = 120;
 pMMI -> ptMinTrackSize.y = 120;
}
```

复制给 MINMAXINFO 的跟踪尺寸是用设备单位(或称为像素)测量的。在本例中,窗口具有的最大尺寸没有限制,因为没有修改 pMMI -> ptMaxTrackSize。通过给消息处理程序添加下列语句,可以限制最大窗口尺寸为屏幕的一半：

```
pMMI -> ptMaxTrackSize.x = ::GetSystemMetrics (SM_CXSCREEN) / 2;
pMMI -> ptMaxTrackSize.y = ::GetSystemMetrics (SM_CYSCREEN) / 2;
```

14.3　空闲处理

由于是 MFC 的应用程序类 CWinApp 提供了消息循环来检索和调度消息,所以在没有消息等待处理时就在应用程序中调用一个函数,对 CWinApp 来讲就很简单。您看一下 CWinThread::Run 函数的源程序代码,该函数由 WinMain 调用来启动消息循环,可以看到如下内容：

```
BOOL bIdle = TRUE;
LONG lIdleCount = 0;

for (;;)
{
   while (bIdle &&
       !::PeekMessage(&m_msgCur, NULL, NULL, NULL, PM_NOREMOVE))
   {
       if (!OnIdle(lIdleCount ++ ))
          bIdle = FALSE;
   }

   do
   {
       if (!PumpMessage())
          return ExitInstance();

       if (IsIdleMessage(&m_msgCur))
       {
          bIdle = TRUE;
          lIdleCount = 0;
```

```
        }
    } while (::PeekMessage(&m_msgCur, NULL, NULL, NULL, PM_NOREMOVE));
}
```

在它调用 PumpMessage 来检索和调度消息之前，Run 会用 PM_NOREMOVE 标志来调用 ::PeekMessage检查消息队列。如果有消息等待，::PeekMessage 就将该消息复制到 MSG 结构中并返回非零值，但不会从队列中删除消息。如果没有消息等待，::PeekMessage 返回 0。与 ::GetMessage 不同，::PeekMessage 并不等待消息从消息队列中出现，它会立即返回。如果 ::PeekMessage 返回非零值，说明有消息等待处理，CWinThread::Run 就会进入 do-while 循环反复调用 CWinThread::PumpMessage 来检索和调度消息。如果 ::PeekMessage 返回 0 并设置了 bIdle 标志，CWinThread::Run 就会调用一个成员函数 OnIdle 给应用程序一个机会去执行空闲处理。因为 OnIdle 是虚函数，CWinApp 又是从 CWinThread 派生来的，所以派生应用程序类可以用自己的 OnIdle 函数替换 CWinApp::OnIdle 来钩住空闲循环。

在使用 Windows 3.x 时期，应用程序本质上是单线程的，OnIdle 的最好用途是执行后台处理任务，如后台打印和无用单元回收等。在 32 位 Windows 中，CWinApp::OnIdle 具有了更大的用处，因为那些具有低优先级别的任务可以在执行的后台线程中更高效地完成了。OnIdle 还有其他合法的用途。MFC 使用它通过调用注册在消息映射表中的更新处理程序来更新工具栏按钮和其他总是可见的用户界面对象。在应用程序不忙于处理消息时，可以充分利用时间来删除由函数如 CWnd::FromHandle 和 CWnd::GetMenu 生成的临时对象。

当您调用 FromHandle 将窗口句柄转换为 CWnd 指针时，MFC 要查阅内部列表，该列表称为“句柄映射表”，它将 CWnd 对象和窗口句柄联系起来。如果找到了要找的句柄，MFC 就返回一个相应的指向 CWnd 对象的指针。但是，如果由于相应的 CWnd 不存在而使窗口句柄没有出现在句柄映射表中，FromHandle 就会创建一个临时的 CWnd 对象并将该对象的地址返回给调用者。在下次调用 OnIdle 时（只有在调用 FromHandle 的消息处理程序返回后才发生），MFC 执行清除工作来删除此临时 CWnd 对象。这就是为什么有些 MFC 函数的相关文献警告返回的临时指针“不应该保留到以后使用”的原因。实际上这意味着由这些临时指针引用的对象并不能保证存在于当前消息处理程序作用范围以外，一旦处理程序返回，在任何时候都可能会调用 OnIdle，使对象被删除。

14.3.1　使用 OnIdle

MFC 应用程序可以重载从 CWinApp 继承来的 OnIdle 虚函数来指定空闲处理的任务。OnIdle 的原型如下：

```
virtual BOOL OnIdle(LONG lCount)
```

lCount 是一个 32 位值，指定了从上次消息处理以来 OnIdle 被调用的总次数。计数值总数会连续增加直到 CWinThread::Run 中的消息循环调用 PumpMessage 检索和调度另一个消息为

上。然后计数值重置为 0 再从头开始。WM_PAINT 消息、WM_SYSTIMER 消息和某些鼠标消息不会使 lCount 复位(WM_SYSTIMER 消息没有文献说明 Windows 内部使用。)。lCount 可以用来对上次消息调度后经过的时间,或叫做应用程序空闲的时间,进行大致的测量。如果您有两个后台任务希望在空闲时间内执行,一个优先级别高,一个优先级别低,那您就可以安排优先级别高的任务每次 lCount 达到 10 时执行,而优先级别低的任务在 lCount 达到 100 或甚至 1 000 时执行。

如果您能在不降低其执行速度的情况下记录对应用程序的 OnIdle 函数的调用次数,就会发现 1 000 并不是很大的数字。通常,每秒会调用 OnIdle 100 次或更多,一个 lCount 达到 1 000就启动的低优先级别的后台任务一般会在鼠标和键盘闲置几秒后就得到执行。lCount 达到 10 就执行的高优先级别任务会更频繁地得到执行,因为即使消息循环相当忙,计数值也经常达到或超过 10。空闲处理应该尽可能地快速执行,否则在 OnIdle 返回之前消息传输就可能被堵塞。

OnIdle 返回值决定了是否再次调用 OnIdle。如果 OnIdle 返回非零值且消息队列仍然为空时会得到再次调用。如果 OnIdle 返回 0,就会暂停进一步的调用,直到另一个消息设法进入消息队列并在消息被调度之后重新进入空闲状态。实现此工作的机制是 CWinThread::Run 中的 bIdle 标志,它最初被设置为 TRUE,如果 OnIdle 返回 FALSE 它将设置为 FALSE。调用 OnIdle 的 while 循环在每次迭代开始时都检测 bIdle 的值,如果 bIdle 为 FALSE 则退出循环。在消息从消息队列中出现并调用 PumpMessage 时 bIdle 被重新设置为 TRUE。在实际中,可以通过下述方法节省 CPU 周期:如果在某时后台处理完成而又不希望在继续消息流之前再次调用 OnIdle,就可以直接从 OnIdle 返回 FALSE。但是要小心不要在主框架完成最近执行的大量空闲处理杂务之前返回 FALSE,这样的话会剥夺它需要的空闲时间。

使用 OnIdle 的主要原则是从重载版本中调用 OnIdle 的基类版本。下列 OnIdle 重载函数说明了其适当的用法。首先调用基类的 OnIdle 函数,在调用返回之后应用程序执行自己的空闲处理:

```
BOOL CMyApp::OnIdle (LONG lCount)
{
 CWinApp::OnIdle (lCount);
 DoIdleWork (); // Do custom idle processing.
 return TRUE;
}
```

实际上当 lCount 等于 0 和 1 时是主框架在执行它的处理。因此给予主框架的 OnIdle 处理程序高优先级别的更好方法是将您自己的空闲处理延迟到 lCount 达到 2 或更高。

```
BOOL CMyApp::OnIdle (LONG lCount)
{
 CWinApp::OnIdle (lCount);
```

```
if (lCount == 2)
   DoIdleWork(); // Do custom idle processing.
return TRUE;
```

通过研究 Thrdcore.cpp 中的 CWinThread::OnIdle 和 Appcore.cpp 中的 CWinApp::OnIdle 源程序代码，您可以了解 MFC 在空闲时间内所做的工作。

由于上段中给出的 OnIdle 实现总是返回 TRUE，所以即使您和主框架使用 OnIdle 执行的工作按时完成，对 OnIdle 的调用也会持续下去。而下面的 OnIdle 重载函数在 MFC 的空闲处理和应用程序的空闲处理结束以后就返回 FALSE，从而减少了系统消耗：

```
BOOL CMyApp::OnIdle(LONG lCount)
{
 BOOL bContinue = CWinApp::OnIdle(lCount);
 if(lCount == 2)
   DoIdleWork(); // Do custom idle processing.
 return (bContinue||lCount < 2);
}
```

应用程序特定的空闲处理直到 lCount 等于 2 时才启动意味着：如果应用程序的 OnIdle 函数返回 FALSE，主框架也不会被剥夺所需要的空闲时间。

有一点很重要，就是执行空闲处理要尽可能地快，以避免对应用程序的相应能力产生负面影响。如果必要，可以将大的 OnIdle 任务分解为更小的更易管理的任务段，连续调用 OnIdle，每次执行一个小任务段。以下 OnIdle 函数在 lCount 达到 2 时开始工作并继续响应对 OnIdle 的调用直到 DoIdleWork 返回 0：

```
BOOL CMyApp::OnIdle(LONG lCount)
{
  BOOL bMFCContinue = CWinApp::OnIdle(lCount);
  BOOL bAppContinue = TRUE;
  if(lCount >= 2)
     bAppContinue = DoIdleWork(); // Do custom idle processing.
  return (bMFCContinue||bAppContinue);
}
```

因为 DoIdleWork 的返回值也用作 OnIdle 的返回值，所以一旦 DoIdleWork 完成自己的分配任务就会停止调用 OnIdle。

14.3.2　对比空闲处理和多线程处理

在第 17 章中，将学到另一种执行后台任务的方法，它涉及到执行多个线程。多线程处理是优秀的程序设计范例，它是执行并行的两个以上任务的理想选择。它还可以扩展功能：

在包含 n 个 CPU 的多处理器系统中,Windows NT 和 Windows 2000 可以同时执行 n 个线程调度,每个线程运行在不同的处理器上。(与 Windows 95 和 Windows 98 对比,它们即使在多处理器系统中也会迫使所有线程共享单个 CPU。)

因为 32 位 Windows 具有健壮的多线程支持,我们有理由知道何时在用到多线程的地方应该使用空闲处理。下面是两个答案:

- 当您的后台任务必须在应用程序的主线程中执行时,与用户界面相关的任务通常都与线程关系密切,这正是 MFC 在主线程中执行用户界面更新的原因。
- 当有后台任务要执行而且编写的应用程序必须在 16 位 Windows 和 32 位 Windows 上运行时,在 16 位 Windows 中不支持多线程处理。

在这些情况下,在 OnIdle 中执行后台任务就很合理。而在其他时候,多线程处理可能会更合适。

第 15 章　位图、调色板以及区域

牢固掌握 Microsoft Windows GDI 是成为 Windows 程序员的重要环节，因为屏幕、打印机和其他设备上的图像输出都是通过 GDI 实现的。在本书中目前我们已使用了代表 GDI 对象的 6 个 MFC 类中的 3 个：CPen、CBrush 和 CFont。本章将讨论剩余的 3 个：CPalette、CBitmap 和 CRgn。

CPalette 代表调色板，它是颜色列表，用来使 Windows 调解应用程序需要更多的视频适配器无法提供的颜色而造成的冲突。如果所有视频适配器都可以显示每像素 24 位颜色（红、绿、蓝各 8 位），那就不需要调色板了。但是 256 色视频适配器还在发展过程中，可能还需要一段时间才能实现。在默认状态下，执行在 256 色环境下的 Windows 应用程序只能得到 20 种颜色。如果您仔细地选取这些颜色使它们成为调色板的一部分，就可以将颜色供选种类扩展到 256 种，可以编写出这样的应用程序，其颜色输出在显示 256 种颜色的屏幕上和在显示上万种颜色的屏幕上效果相同。本章中将学习如何在应用程序中使用调色板来生成硬件设备允许的丰富的颜色输出。

MFC 的 CBitmap 类代表 GDI 位图。CBitmap 是原始类，它自己几乎不做任何工作。但是与 MFC 的 CDC 类结合 CBitmap 就使得以下操作非常易于实现：在内存中虚拟设备显示表面上绘制输出；加载位图资源；以及在屏幕上显示简单的位图图像。还可以使用 CBitmap 创建功能更强大的位图类，用来开发利用 Windows 设备无关位图（DIB）处理机的功能。本章将介绍一项技术，用来从 BMP 文件创建 DIB 片段并把它附加给普通的 CBitmap 对象，整个过程只需要 3 行代码。

CRgn 是 MFC 中较难理解的类之一，但是可以使用它处理外来的图像元素。为了不破坏您的兴致，我将详细内容留在本章最后再讲解。

15.1　调色板

您是否曾经开发过这样的应用程序，使用了丰富的颜色却发现输出结果在 16 色和 256 色视频适配器上都不理想？如果视频适配器只支持 16 种颜色，那您可能对此毫无办法，而如果是 256 色的设备，通过一些工作就可以改进颜色输出，其关键就在于 MFC 的 CPalette 类。在我们进入 CPalette 的具体内容之前，先来简要介绍一下在 Windows 中颜色信息的编码方式以及 Windows 对您提供的颜色信息所做的处理。

15.1.1 Windows 使用颜色的方式

设备无关输出方式的一个优点是可以指定应用程序使用的颜色而不必考虑输出设备的物理特性。当给 Windows GDI 传递一种颜色时,就传递一个 COLORREF 值,其中包含红、绿、蓝三色各 8 位。RGB 宏将单独的红色、绿色和蓝色值组合在一个 COLORREF 中。语句

```
COLORREF clr = RGB(255, 0, 255);
```

生成一个名为 clr 的 COLORREF 值,代表品红(将相等的红色和蓝色混合后得到的颜色)。反过来,可以使用 GetRValue、GetGValue 和 GetBValue 宏从 COLORREF 值中提取出 8 位红色、绿色、蓝色值。许多 GDI 函数,包括生成画笔和画刷的函数,都接受 COLORREF 值。

GDI 对 COLORREF 值的处理依赖于几个因素,包括视频硬设备的颜色分辨率和使用颜色的设备描述表。最简单、最理想的方案是视频适配器为每像素 24 位的设备,COLORREF 值可以直接转换成屏幕上的颜色。支持 24 位颜色(真彩色)的视频适配器正得到越来越普遍的使用,但是 Windows 仍然运行在许多显示器只能支持每像素 4 或 8 位的 PC 机上。通常,这些设备是"调色板化的设备",就是说它们支持的颜色范围很广,但是只能一次显示其中的一部分。例如:标准的 VGA 可以显示 262 144 种不同的颜色(红、绿、蓝各 6 位),但是运行在分辨率为 640 × 480 像素模式下的 VGA 同时只能显示 16 种不同的颜色,因为每个像素在视频缓冲区中只能保存 4 位颜色信息。更普遍的情况是可以显示超过 1.67 千万种颜色的视频适配器只能同时显示 256 种颜色。可以显示的 256 种颜色是根据 RGB 值确定的,该值编入了适配器的硬件调色板中。

Windows 通过在适配器硬件调色板中编制标准的供选颜色来处理调色板化的设备。在表 15-1 中列出了 20 种为 256 色适配器编制的所谓"静态颜色"。有星号标记的 4 种颜色易于被操作系统的命令修改,因此不要编写依赖于它们程序。

表 15-1 静态调色板颜色

颜色	R	G	B	颜色	R	G	B
Black	0	0	0	Cream *	255	251	240
Dark red	128	0	0	Intermediate gray *	160	160	164
Dark green	0	128	0	Medium gray	128	128	128
Dark yellow	128	128	0	Red	255	0	0
Dark blue	0	0	128	Green	0	255	0
Dark magenta	128	0	128	Yellow	255	255	0
Dark cyan	0	128	128	Blue	0	0	255
Light gray	192	192	192	Magenta	255	0	255
Money green *	192	220	192	Cyan	0	255	255
Sky blue *	166	202	240	White	255	255	255

* 表示易于改变的默认颜色。

在调色板化的设备上绘图时，GDI 使用简单的颜色匹配算法把每个 COLORREF 值都映射给最接近的静态颜色。如果给生成画笔的函数传递一个 COLORREF 值，Windows 就会给画笔分配一个最接近的静态颜色。如果把 COLOREF 值传递给生成画刷的函数而又找不到匹配的静态颜色时，Windows 就会使用静态颜色来抖动实现画刷颜色。因为静态颜色包含多种(如果限制的话)不同的色调，所以 Windows 可以执行一些合适的工作来模拟您提供的任何 COLORREF 值。用 100 种不同的红色绘制的图像，显示的效果就不会太好，因为 Windows 只能用两种红色来模拟所有 100 种色调。但是这可以保证红色决不会被转化为蓝色、绿色或其他颜色，因为有静态颜色在那里而且总是有效。

对于许多应用程序而言，Windows 使用静态颜色进行颜色映射的原始形式已经足够了。但是对于其他特别关心精确颜色输出的应用程序，20 种颜色远远不够。在单任务环境如 MS-DOS 中，运行在 256 色适配器上的程序可以编制自己的硬件调色板任意地使用 256 种颜色。但是在 Windows 中，不允许应用程序直接编写硬件调色板，因为视频适配器是共享资源。那么在 Windows 添加了 20 种静态颜色之后，在 256 色适配器中如何利用剩下未用的 236 种颜色呢？答案就在 GDI 对象中，通常称为逻辑调色板。

15.1.2　逻辑调色板和 CPalette 类

“逻辑调色板”是一个 RGB 颜色值的列表，告诉 Windows 应用程序想要显示的颜色。相关术语“系统调色板”指的是适配器的硬件颜色调色板。对于应用程序的申请，内置于 Windows 中的调色板管理器将逻辑调色板中的颜色传送给系统调色板中未使用的输入项(此过程称为实现调色板)，以便应用程序可以充分利用视频适配器的颜色性能。在逻辑调色板的帮助下，运行在 256 色视频适配器上的应用程序就可以使用 20 种静态颜色外加 236 种自选的颜色了。因为所有对实现调色板的申请都要通过 GDI，所以调色板管理器可以作为产生颜色需求冲突的程序之间的仲裁者，这样可以保证系统调色板被共同使用。

如果两个以上的应用程序要实现逻辑调色板，并且它们申请的颜色总数超出了 256 色视频适配器所能处理的附加 236 种颜色，那将出现什么情况呢？调色板管理器基于每个窗口在 z 向上的顺序来分配颜色优先级别。在 z 向顶端的窗口接收最高优先级别，第二个窗口接收到次高优先级别，等等。如果前端窗口实现了一个具有 200 种颜色的调色板，这 200 种颜色就都会被映射到系统调色板上。如果后来后台窗口实现了比如 100 种颜色的调色板，那么 36 种会被编制在系统调色板内剩下未使用的存储槽中，64 种颜色会映射为最接近的匹配颜色。这是最坏的情况。除非指示要这样做，否则调色板管理器会避免在系统调色板中重复输入项。因此，如果有 4 种前端窗口的颜色和 10 种后台窗口的颜色与静态颜色匹配，并且另外有 10 种后台窗口的颜色与前端窗口的非静态颜色匹配，那么最终后台窗口在系统调色板中会得到 60 种准确的匹配颜色。

您可以通过以下方法查看调色板管理器的工作，将 Windows 切换到 256 色模式下，运行两个 Windows “画图”应用程序实例，通过单击使它们前后变化。在前端的位图颜色效果总

足最好,因为它首先控制系统调色板。在后台的位图则获得剩下的颜色。如果两个位图的颜色相似,那么后台的图像看上去还不是很差。但是如果颜色完全不同,例如,如果位图 A 包含鲜明的颜色而位图 B 主要是土石的颜色,那么在后台窗口中的图像就会由于颜色恶化而无法识别出来。在处理过程中调色板管理器的作用就是试图满足两个程序的需要。在需要出现冲突时,前端窗口得到的优先级别要比其他窗口的高,使得用户正在使用的应用程序看上去效果更好。

15.1.3 创建逻辑调色板

编写使用逻辑调色板的应用程序并不困难。在 MFC 中,用 CPalette 类代表逻辑调色板并用 CPalette 成员函数来创建和初始化它。一旦创建了逻辑调色板,就可以把它选入设备描述表,并用 CDC 的成员函数来实现了。

为创建调色板,CPalette 提供了两个成员函数。CreatePalette 根据指定的 RGB 值创建自定义调色板,CreateHalftonePalette 创建"半色调"调色板,包含有一般的相当均匀的颜色分布。对于几乎不含有显著差异的颜色,但具有许多色调上精细变化的图像,自定义调色板可以提供更好的效果。而对于包含颜色范围很广的图像,半色调调色板就很好用。语句

```
CPalette palette;
palette.CreateHalftonePalette (pDC);
```

针对 pDC 指向的设备描述表创建了一个半色调调色板。如果设备描述表与 256 色设备对应,半色调调色板也会包含 256 种颜色。其中 20 种与静态颜色匹配;其他 236 种通过添加红色、绿色和蓝色的色调变化颜色以及三原色的混合色扩充了有效的供选颜色。具体地讲,256 色半色调调色板包含的颜色有:6×6×6 颜色立方体中所有的颜色(由红色、绿色、蓝色各自的 6 种色调组成的颜色);灰度级成像的灰色数组;以及其他 GDI 选中的颜色。如果传递给 CreateHalftonePalette 的设备句柄为 NULL,就可以创建与输出设备特性无关的 256 色半色调调色板。如果传递的是一个 NULL DC 句柄,在调试编译过程中 CPalette::CreateHalftonePalette 会产生有效性检查错误,于是必须降级使用 Windows API:

```
CPalette palette;
palette.Attach (::CreateHalftonePalette (NULL));
```

::CreateHalftonePalette 是与 CPalette::CreateHalftonePalette 等价的 API 函数。

创建一个自定义调色板要复杂一些,因为在调用 CreatePalette 之前必须用描述调色板颜色的输入项初始化一个 LOGPALETTE 结构。LOGPALETTE 定义如下:

```
typedef struct tagLOGPALETTE {
    WORD palVersion;
    WORD palNumEntries;
    PALETTEENTRY palPalEntry[1];
```

```
} LOGPALETTE;
```

palVersion 指定 LOGPALETTE 版本号；在目前所有 Windows 版本中，它应该被设置为 0x300。palNumEntries 指定调色板中的颜色数量。palPalEntry 是定义颜色的 PALETTEENTRY 结构的数组，数组中的元素个数等于 palNumEntries 中的值。PALETTEENTRY 定义如下：

```
typedef struct tagPALETTEENTRY {
    BYTE peRed;
    BYTE peGreen;
    BYTE peBlue;
    BYTE peFlags;
} PALETTEENTRY;
```

peRed、peGreen 和 peBlue 指定了颜色的 8 位 RGB 成分。peFlags 包含 0 个或更多的位标志用来描述调色板输入项的“类型”。它可以被设置为表 15-2 中列出的任意值，如果设置为 0 则会创建一个“标准”调色板输入项。

表 15-2　可设置的值

标志	说　明
PC_EXPLICIT	创建一个调色板输入项用来指定系统调色板而不是 RGB 颜色的索引号。用于显示系统调色板内容的程序
PC_NOCOLLAPSE	创建一个调色板输入项使它映射到系统调色板中未使用的输入项上，即使该颜色已经有了输入项也要这样做。用来确保调色板颜色的唯一性
PC_RESERVED	创建一个对应用程序而言是私有的调色板输入项。当 PC_RESERVED 输入项添加到系统调色板后，即使颜色匹配它也不会将颜色映射到其他逻辑调色板上。用于执行调色板动画的程序

通常，peFlags 被简单地设置为 0。在本章稍后有关调色板动画的内容中，我们将讨论如何使用 PC_RESERVED 标志。

LOGPALETTE 结构中的 PALETTEENTRY 数组只声明了一个数组元素，因为 Windows 无法预计逻辑调色板会包含的颜色数量。所以，不可以仅仅是在堆栈上声明一个 LOGPALETTE 实例并填写其中内容；相反应该基于它所包含的 PALETTEENTRY 结构的数量来为它分配内存。下列程序在堆栈上分配了一个“完整”的 LOGPALETTE 结构，然后创建了一个包含 32 种不同红色调的逻辑调色板：

```
struct {
    LOGPALETTE lp;
    PALETTEENTRY ape[31];
} pal;

LOGPALETTE * pLP = (LOGPALETTE *) &pal;
```

```
pLP->palVersion = 0x300;
pLP->palNumEntries = 32;

for (int i = 0; i < 32; i++) {
    pLP->palPalEntry[i].peRed = i * 8;
    pLP->palPalEntry[i].peGreen = 0;
    pLP->palPalEntry[i].peBlue = 0;
    pLP->palPalEntry[i].peFlags = 0;
}

CPalette palette;
palette.CreatePalette (pLP);
```

与其他 GDI 对象一样,逻辑调色板应该在不再需要时删除。由 CPalette 对象表示的逻辑调色板在相应的 CPalette 对象被删除或超出有效范围后会被自动删除。

一个逻辑调色板可以包含多少个输入项呢?可以包含任意多个。当然,可以直接映射到系统调色板上的颜色数量要受到视频适配器性能的限制。如果您在 256 色输出设备上实现了一个包含 1 024 种颜色的调色板,则只有前 236 种颜色可以得到直接映射;剩下的颜色会与系统调色板中已经存在的颜色尽可能地匹配。在使用逻辑调色板(特别是大型逻辑调色板)时,最好按重要性顺序安排颜色,palPalEntry[0] 可以用来定义最重要的颜色,palPalEntry[1] 定义次重要的颜色,等等。调色板管理器按数组顺序映射调色板颜色,所以将重要的颜色放在前面可以增加这些颜色以它们的本色被显示的机会。通常,没必要使逻辑调色板过分地大。调色板越大开销就越大,前端窗口所用的颜色越多,调色板管理器留给 z 向靠后使用调色板窗口的有效颜色就越少。

在调色板创建以后,可以使用 CPalette::GetPaletteEntries 检索单个调色板输入项或使用 CPalette::SetPaletteEntries 修改它们。还可以使用 CPalette::ResizePalette 调整调色板的大小。如果调色板被放大,新的输入项初始值都是 0。

15.1.4 实现逻辑调色板

为使逻辑调色板有效,必须将其选入设备描述表并在任何绘制工作执行以前实现它。当前的逻辑调色板是设备描述表的属性,如同当前的画笔和画刷是设备描述表的属性一样(其实,设备描述表默认一个很小的逻辑调色板,其中的输入项与静态颜色相应)。下列 OnPaint 处理程序将逻辑调色板 m_palette 选入设备描述表并在重绘屏幕之前实现了调色板:

```
void CMainWindow::OnPaint ()
{
    CPaintDC dc (this);
    CPalette * pOldPalette = dc.SelectPalette (&m_palette, FALSE);
    dc.RealizePalette ();
    // Do some drawing.
```

```
    dc.SelectPalette (pOldPalette, FALSE);
}
```

在本例中，保存指向默认调色板的指针是为了以后从设备描述表中选出 m_palette 时再使用。注意调色板是用 CDC::SelectPalette 而不是用 CDC::SelectObject 选取的。第二个参数是 BOOL 值，如果为 TRUE，就要强迫调色板即使是选用它的窗口在前端也要像它在后台一样操作。在使用了多个调色板的应用程序中，使用后台调色板可能会方便些，但一般情况下都要在调用 SelectPalette 时指定为 FALSE。CDC::RealizePalette 通过申请调色板管理器将逻辑调色板中的颜色映射到系统调色板上，从而实现了当前选入设备描述表中的调色板。

15.1.5　用调色板颜色绘图

一旦创建了调色板，把它选入设备描述表，以及实现它以后，就可以画图了。如果使用 CDC::BitBlt 来显示位图，就会自动使用已实现的调色板。但是如果是使用画刷、画笔或那些类似于 CDC::FloodFill 的既不用画刷也不用画笔的函数通过 COLORREF 值直接画图，那就有一些问题您必须考虑了。

RGB 宏是创建 COLORREF 值的 3 个宏之一。其他两个是 PALETTEINDEX 和 PALETTERGB。它们决定了 GDI 如何对待 COLORREF。当使用由 RGB 生成的 COLORREF 值绘图时，GDI 将忽略实现逻辑调色板时给系统调色板添加的颜色而只使用静态颜色。如果希望 GDI 使用所有调色板颜色，就要使用 PALETTERGB 宏。PALETTERGB 生成"调色板相关"颜色。PALETTEINDEX 宏生成的 COLORREF 值指定了逻辑调色板而不是 RGB 颜色值的索引号。该值被称为"调色板索引号"颜色值。用这种颜色画图最快，因为它使得 GDI 可以不必将 RGB 颜色匹配给逻辑调色板中的颜色。

下列程序示例说明了 3 种宏的使用方法：

```
void CMainWindow::OnPaint ()
{
    CPaintDC dc (this);

    // Select and realize a logical palette.
    CPalette* pOldPalette = dc.SelectPalette (&m_palette, FALSE);
    dc.RealizePalette ();

    // Create three pens.
    CPen pen1 (PS_SOLID, 16, RGB (242, 36, 204));
    CPen pen2 (PS_SOLID, 16, PALETTERGB (242, 36, 204));
    CPen pen3 (PS_SOLID, 16, PALETTEINDEX (3));

    // Do some drawing.
    dc.MoveTo (0, 0);
    CPen* pOldPen = dc.SelectObject (&pen1);
    dc.LineTo (300, 0);    // Nearest static color
```

```
    dc.SelectObject (&pen2);
    dc.LineTo (150, 200);    // Nearest static or palette color
    dc.SelectObject (&pen3);
    dc.LineTo (0, 0);    // Exact palette color
    dc.SelectObject (pOldPen);

    // Select the palette out of the device context.
    dc.SelectPalette (pOldPalette, FALSE);
}
```

因为画笔使用的是原色而非抖动的颜色,由于它的 COLORREF 值是用 RGB 宏指定的,所以 pen1 用静态颜色绘图,该颜色最接近 RGB 值 (242, 36, 204)。而另一方面,从静态颜色或 m_palette 中给 pen2 分配了一个最接近的匹配颜色。pen3 不管颜色是什么都将使用系统调色板中与逻辑调色板中第四个颜色(索引号为 3)相应的颜色。

15.1.6 WM_QUERYNEWPALETTE 和 WM_PALETTECHANGED 消息

在编写使用逻辑调色板的应用程序时,应该包含处理一对 WM_QUERYNEWPALETTE 和 WM_PALETTECHANGED 消息的处理程序。当顶层窗口的子窗口接收到输入焦点时,就会给顶层窗口发送一个 WM_QUERYNEWPALETTE 消息。当调色板的实现导致系统调色板改动时,就会给系统中的顶层窗口发送 WM_PALETTECHANGED 消息。对这两个消息,应用程序通常的响应是实现自己的调色板并重绘自身。实现调色板并进行重绘来响应 WM_QUERYNEWPALETTE 消息,就使得即将进入前端的窗口可以利用实现调色板的最高优先级来展示自己最好的外观。实现调色板并进行重绘来响应 WM_PALETTECHANGED 消息使得后台的窗口可以适应系统调色板中的变动,而且还可以利用所有 z 向上高层窗口实现它们的调色板以后剩下的未用的输入项。

下列消息处理程序说明了对 WM_QUERYNEWPALETTE 消息的典型响应:

```
// In the message map
ON_WM_QUERYNEWPALETTE ()
    .
    .
    .
BOOL CMainWindow::OnQueryNewPalette ()
{
    CClientDC dc (this);
    CPalette* pOldPalette = dc.SelectPalette (&m_palette, FALSE);

    UINT nCount;
    if (nCount = dc.RealizePalette ())
        Invalidate ();
```

```
    dc.SelectPalette(pOldPalette, FALSE);
    return nCount;
}
```

通常实现调色板和迫使重绘的策略是使窗口的客户区无效。RealizePalette 返回的值是映射到系统调色板输入项上的调色板输入项个数。返回值 0 说明实现的调色板无效，对于前端窗口而言这是极其少见的。如果 RealizePalette 返回 0，就应该跳过对 Invalidate 的调用。如果逻辑调色板已经实现，OnQueryNewPalette 应该返回非零值，否则返回 0。如果试图实现调色板，可是 RealizePalette 返回为 0 时，它也要返回 0。在现在的 Windows 版本中用不到此返回值。

对 WM_PALETTECHANGED 消息的处理与前一个相似。下面给出典型的 OnPaletteChanged 处理程序：

```
// In the message map
ON_WM_PALETTECHANGED()
    .
    .
    .
void CMainWindow::OnPaletteChanged(CWnd* pFocusWnd)
{
    if (pFocusWnd != this) {
        CClientDC dc(this);
        CPalette* pOldPalette = dc.SelectPalette(&m_palette,
            FALSE);
        if (dc.RealizePalette())
            Invalidate();
        dc.SelectPalette(pOldPalette, FALSE);
    }
}
```

传递给 OnPaletteChanged 的 CWnd 指针标识因实现调色板而激发了 WM_PALETTECHANGED 消息的窗口，为避免不必要的递归和可能产生的死循环，如果 pFocusWnd 指向自己的窗口时 OnPaletteChanged 应该什么操作都不执行。这就是用 if 语句对 pFocusWnd 和 this 进行比较的原因。

在响应 WM_PALETTECHANGED 消息时为了不执行全部重绘，应用程序可以调用 CDC::UpdateColors 来有选择地进行绘制。UpdateColors 通过将每个像素的颜色与系统调色板的颜色匹配来更新窗口。通常，这样做要比全部重绘快许多，但是效果一般不怎么好，因为颜色匹配是基于系统调色板修改前的内容。如果要使用 UpdateColors，可以维持一个变量用来记录 UpdateColors 被调用的次数。然后每隔 3 到 4 次，做一次全部重绘并将计数器复位到 0。这样就可以防止后台窗口中的颜色与系统调色板中的颜色变得非常不一致。

在文档/视图应用程序中处理调色板消息

使用上面的 OnQueryNewPalette 和 OnPaletteChanged 处理程序的前提是要更新的窗口是应用程序的主窗口。在文档/视图应用程序中,情形却不是这样,需要更新的是视图而不是顶层窗口。理想的解决办法是将 OnQueryNewPalette 和 OnPaletteChanged 处理程序放在视图类中,但这又无法实现,因为视图不能接收调色板消息(只有顶层窗口可以)。

而大多数文档/视图应用程序所采用的方法是让它们的主窗口响应调色板消息来更新视图。下列 OnQueryNewPalette 和 OnPaletteChanged 处理程序使用于大多数 SDI 应用程序:

```
BOOL CMainFrame::OnQueryNewPalette()
{
    CDocument* pDoc = GetActiveDocument();
    if (pDoc != NULL)
        GetActiveDocument()->UpdateAllViews(NULL);
    return TRUE;
}

void CMainFrame::OnPaletteChanged(CWnd* pFocusWnd)
{
    if (pFocusWnd != this) {
        CDocument* pDoc = GetActiveDocument();
        if (pDoc != NULL)
            GetActiveDocument()->UpdateAllViews(NULL);
    }
}
```

在 MDI 应用程序中使用调色板就有点麻烦。如果每个打开的文档都有与其关联的唯一的调色板(大多数情况下是这样),活动的视图应该用前端调色板来重绘,而不活动的视图应该用后台调色板重绘。对于使用了多个调色板的 MDI 应用程序存在着另一个问题,就是当用户在视图间切换时需要更新视图的颜色。最好的解决办法是重载 CView::OnActivateView,使视图知道什么时候处于活动状态什么时候不活动,从而相应地实现自己的调色板。在 MDI 应用程序中处理调色板的一个很好的例子就是 Visual C++ 中提供的 DIBLOOK 示例程序。

15.1.7 确定是否需要逻辑调色板

既然已经理解了调色板的使用机制,现在可以问自己这样的一个问题:我怎么能知道特别需要逻辑调色板呢?如果颜色的准确性是您所关心的问题,当您的应用程序运行在调色板化的 256 色视频适配器上时可能就需要使用逻辑调色板。但是同样的应用程序当硬件颜色深度为 24 位时就不需要逻辑调色板了,因为在这样的环境下可以任意实现完美的颜色输出。而且应用程序运行在标准 16 色 VGA 上时,也没必要使用调色板,因为系统调色板只

有 16 种静态颜色而没有额外的空间存放逻辑调色板中的颜色。

在运行过程中可以确定逻辑调色板是否可以改善颜色输出，用 RASTERCAPS 参数调用 CDC::GetDeviceCaps 并检查返回值中的 RC_PALETTE 位，如下：

```
CClientDC dc(this);
BOOL bUsePalette = FALSE;
if(dc.GetDeviceCaps(RASTERCAPS) & RC_PALETTE)
   bUsePalette = TRUE;
```

RC_PALETTE 会在调色板化的颜色模式下被设置，在非调色板化的模式下被清除。一般而言，RC_PALETTE 位在 8 位颜色模式下被设置，在 4 位和 24 位颜色模式下被清除。如果适配器运行在 16 位色（“丰富颜色”）模式下 RC_PALETTE 位也会被清除，该模式对于大多数应用程序而言每位生成的颜色输出都与真彩色相当。不要犯某些程序员曾犯过的错误，不要依赖位的个数来确定是否需要使用调色板。您可以确信，有时会遇到古怪的视频适配器，它一反常规使应用程序在不需要调色板的时候误使用调色板，或是在真正需要它帮助的时候却不使用。

如果忽略 RC_PALETTE 设置，不管颜色深度如何而使用逻辑调色板会出现什么情况呢？因为调色板管理器即使在非调色板化的设备上也能工作，所以应用程序仍然会得到执行。如果 RC_PALETTE 是 0，调色板仍然会被创建并选入设备描述表中，但是 RealizePalette 不会执行任何动作。PALETTEINDEX 值被间接引用并转换为逻辑调色板中的 RGB 颜色，而 PALETTERGB 值被简单地当作标准 RGB 颜色值。因为没有发送 WM_QUERYNEWPALETTE 和 WM_PALETTECHANGED 消息，所以不会调用 OnQueryNewPalette 和 OnPaletteChanged。所有这些正如在一篇精彩的文章《The Palette Manager: How and Why》（在 Microsoft Developer Network（MSDN）上可以找到）中讲述的那样，“目的是为了让应用程序以设备无关方式使用调色板而不必关心实际设备驱动器的调色板性能”。

但是，通过检查 RC_PALETTE 标志，在标志被清除时跳过与调色板有关的函数，可以节省 CPU 周期。如果您的应用程序依赖硬件调色板的支持，没有它就不能工作，例如应用程序使用了调色板动画（过一会儿将讨论到），那么就可以使用 RC_PALETTE 来确定应用程序是否能在当前硬件上运行。

在考虑是否使用逻辑调色板时，另一个同样重要的问题是：“程序的颜色输出需要达到怎样的精度？”使用与静态颜色匹配的颜色画图的应用程序根本不需要调色板。而另一方面，位图文件查看器几乎肯定需要调色板支持，因为如果没有它的话可能只有最简单的位图在 256 色视频适配器上显示的效果会好些。了解自己程序的颜色需要，仅仅做有必要做的工作，您可以编写出更好的应用程序。

15.1.8　PaletteDemo 应用程序

图 15-1 所示的应用程序说明了在文档/视图应用程序中处理调色板的基本技巧。

PaletteDemo 用了一系列蓝色画刷绘制背景,从蓝到黑使颜色平滑过渡。而且,它在 256 色视频适配器上生成漂亮的梯度填充。在 256 色屏幕上高质量输出的关键是 PaletteDemo 使用了包含 64 种不同蓝色调的逻辑调色板,从纯黑(R = 0,G = 0,B = 3) 到高亮度蓝(R = 0,G = 0,B = 255)。画刷颜色是用与调色板相关的 COLORREF 值指定的,以便在实现逻辑调色板之后 GDI 可以将画刷颜色与系统调色板中的颜色匹配。在 8 位和 24 位颜色模式下分别运行 PaletteDemo,注意输出效果,您会发现它们是一样的。只有当运行在 16 色模式下时,PaletteDemo 就无法生成平滑的梯度填充了。但是即使在这种情况下输出效果也不坏,因为 GDI 抖动了画刷的颜色。

图 15-1 PaletteDemo 窗口

在图 15-2 所示的 PaletteDemo 源程序中有些有趣的内容。就启动程序而言,PaletteDemo 的主窗口在响应 WM_ERASEBKGND 消息时绘制梯度填充的背景。在 WM_PAINT 处理程序绘制前端以前,要发送 WM_ERASEBKGND 消息来擦除窗口背景。像 PaletteDemo 这样绘制自定义窗口背景的 WM_ERASEBKGND 处理程序应该返回一个非零值,通知 Windows 背景已经被"擦除"了。(您可以试一下让 WM_ERASEBKGND 处理程序什么也不做只返回 TRUE,会得到什么结果?一个透明的窗口!是不是很棒?!)否则,Windows 自己会用 WNDCLASS 的背景画刷填充窗口的客户区来擦除背景。

PaletteDemo 创建了一个逻辑调色板,在 CMainWindow::OnCreate 中用来绘制窗口背景。调色板本身是一个名为 m_palette 的 CPalette 数据成员。在创建调色板之前,OnCreate 要检查 CDC::GetDeviceCaps 的返回值中的 RC_PALETTE 位。如果该位没有被设置,OnCreate 就不会初始化 m_palette。在选择和实现调色板以前,CMainWindow::OnEraseBkgnd 将检查

m_palette，看是否存在调色板：

```
if ((HPALETTE) m_palette != NULL) {
    pOldPalette = pDC->SelectPalette(&m_palette, FALSE);
    pDC->RealizePalette();
}
```

CPalette 的 HPALETTE 运算符返回属于 CPalette 对象的调色板的句柄。句柄为 NULL 意味着 m_palette 没有得到初始化。当且仅当视频硬件是调色板化的设备时，OnEraseBkgnd 通过选择和实现逻辑调色板来使自己适应所运行于其中的环境。因为画刷颜色是用 PALETTERGB 宏指定的，所以绘制窗口背景的 DoGradientFill 函数可以用也可以不用调色板来工作。

PaletteDemo 没有考虑的一种情况是，在应用程序运行过程中，如果颜色深度改变了怎么办？您可以通过处理 WM_DISPLAYCHANGE 消息来对付这种情况，该消息是在用户修改了屏幕分辨率和颜色深度后发送的，只需用新的设置重新初始化调色板即可。因为没有 ON_WM_DISPLAYCHANGE 宏，所以必须用 ON_MESSAGE 手工完成消息映射。封装在 WM_DISPLAYCHANGE 消息中的 wParam 参数包含有新的颜色深度，以每像素多少位的形式表达出来；lParam 的低字和高字包含着最新的以像素为单位的水平和垂直屏幕分辨率。

WM_DISPLAYCHANGE 不仅仅用于使用调色板的应用程序。在其他情况下也要使用它，例如：在应用程序开始时用系统字体的平均宽度和高度来初始化变量，之后可以用这些变量来调整输出的大小和位置。如果屏幕分辨率修改后没有重新初始化变量，以后的输出就会被扭曲。

PaletteDemo.h

```
class CMyApp : public CWinApp
{
public:
    virtual BOOL InitInstance();
};

class CMainWindow : public CFrameWnd
{
protected:
    CPalette m_palette;
    void DoGradientFill(CDC* pDC, LPRECT pRect);
    void DoDrawText(CDC* pDC, LPRECT pRect);

public:
    CMainWindow();

protected:
    afx_msg int OnCreate(LPCREATESTRUCT lpcs);
    afx_msg BOOL OnEraseBkgnd(CDC* pDC);
```

```
    afx_msg void OnPaint ();
    afx_msg BOOL OnQueryNewPalette ();
    afx_msg void OnPaletteChanged (CWnd* pFocusWnd);
    DECLARE_MESSAGE_MAP ()
};
```

PaletteDemo.cpp

```
#include <afxwin.h>
#include "PaletteDemo.h"

CMyApp myApp;

//////////////////////////////////////////////////////////////////////////
// CMyApp member functions

BOOL CMyApp::InitInstance ()
{
    m_pMainWnd = new CMainWindow;
    m_pMainWnd->ShowWindow (m_nCmdShow);
    m_pMainWnd->UpdateWindow ();
    return TRUE;
}

//////////////////////////////////////////////////////////////////////////
// CMainWindow message map and member functions

BEGIN_MESSAGE_MAP (CMainWindow, CFrameWnd)
    ON_WM_CREATE ()
    ON_WM_ERASEBKGND ()
    ON_WM_PAINT ()
    ON_WM_QUERYNEWPALETTE ()
    ON_WM_PALETTECHANGED ()
END_MESSAGE_MAP ()

CMainWindow::CMainWindow ()
{
    Create (NULL, _T ("Palette Demo"));
}

int CMainWindow::OnCreate (LPCREATESTRUCT lpcs)
{
    if (CFrameWnd::OnCreate (lpcs) == -1)
        return -1;

    //
    // Create a logical palette if running on a palettized adapter.
```

```
    //
    CClientDC dc(this);
    if (dc.GetDeviceCaps(RASTERCAPS) & RC_PALETTE) {
        struct {
            LOGPALETTE lp;
            PALETTEENTRY ape[63];
        } pal;

        LOGPALETTE* pLP = (LOGPALETTE*) &pal;
        pLP->palVersion = 0x300;
        pLP->palNumEntries = 64;

        for (int i=0; i<64; i++) {
            pLP->palPalEntry[i].peRed = 0;
            pLP->palPalEntry[i].peGreen = 0;
            pLP->palPalEntry[i].peBlue = 255 - (i * 4);
            pLP->palPalEntry[i].peFlags = 0;
        }
        m_palette.CreatePalette(pLP);
        }
        return 0;
}

BOOL CMainWindow::OnEraseBkgnd(CDC* pDC)
{
    CRect rect;
    GetClientRect(&rect);

    CPalette* pOldPalette;
    if ((HPALETTE) m_palette != NULL) {
        pOldPalette = pDC->SelectPalette(&m_palette, FALSE);
        pDC->RealizePalette();
    }

    DoGradientFill(pDC, &rect);

    if ((HPALETTE) m_palette != NULL)
        pDC->SelectPalette(pOldPalette, FALSE);
    return TRUE;
}

void CMainWindow::OnPaint()
{
    CRect rect;
    GetClientRect(&rect);
    CPaintDC dc(this);
    DoDrawText(&dc, &rect);
}
```

```
BOOL CMainWindow::OnQueryNewPalette()
{
    if ((HPALETTE) m_palette == NULL) // Shouldn't happen, but
        return 0;       // let's be sure.

    CClientDC dc(this);
    CPalette* pOldPalette = dc.SelectPalette(&m_palette, FALSE);

    UINT nCount;
    if (nCount = dc.RealizePalette())
        Invalidate();

    dc.SelectPalette(pOldPalette, FALSE);
    return nCount;
}

void CMainWindow::OnPaletteChanged(CWnd* pFocusWnd)
{
    if ((HPALETTE) m_palette == NULL) // Shouldn't happen, but
        return;       // let's be sure.

    if (pFocusWnd != this) {
        CClientDC dc(this);
        CPalette* pOldPalette = dc.SelectPalette(&m_palette, FALSE);
        if (dc.RealizePalette())
            Invalidate();
        dc.SelectPalette(pOldPalette, FALSE);
    }
}

void CMainWindow::DoGradientFill(CDC* pDC, LPRECT pRect)
{
    CBrush* pBrush[64];
    for (int i=0; i<64; i++)
        pBrush[i] = new CBrush(PALETTERGB(0, 0, 255 - (i * 4)));

    int nWidth = pRect->right - pRect->left;
    int nHeight = pRect->bottom - pRect->top;
    CRect rect;

    for (i=0; i<nHeight; i++) {
        rect.SetRect(0, i, nWidth, i + 1);
        pDC->FillRect(&rect, pBrush[(i * 63) / nHeight]);
    }

    for (i=0; i<64; i++)
      delete pBrush[i];
```

```
void CMainWindow::DoDrawText (CDC * pDC, LPRECT pRect)

    CFont font;
    font.CreatePointFont (720,_T ("Comic Sans MS"));

    pDC -> SetBkMode (TRANSPARENT);
    pDC  > SetTextColor (RGB (255, 255, 255));

    CFont * pOldFont = pDC -> SelectObject (&font);
    pDC -> DrawText (_T ("Hello, MFC"), -1, pRect, DT_SINGLELINE
        DT_CENTER|DT_VCENTER);
    pDC -> SelectObject (pOldFont);
}
```

图 15-2　PaletteDemo 应用程序

15.1.9　调色板动画

逻辑调色板的更新颇用途是实现调色板动画。传统的计算机动画是通过在屏幕上不断地绘制、擦除、再绘制图像来实现的，调色板动画却不需要绘制和擦除，但它同样可以使图像移动。调色板动画的一个典型的例子是模拟熔岩流动，使红色、橙色、黄色的不同色调循环变化，产生类似熔岩从山上流动下来的图像。有趣的是这个图像只绘制了一次。产生这种运动幻觉的原因是程序反复调整了系统调色板，使得红色变为橙色，橙色变为黄色，黄色变为红色，这样循环下去。因为不需要移动任何像素，所以调色板动画执行起来很快。把一个简单的值写入视频适配器上的调色板注册表内可以在眨眼间修改整个屏幕上所有像素的颜色，精确地讲显示器的电子枪大约要用 1/60 秒的时间就可以完成一个屏幕刷新周期。

在 Windows 中实现调色板动画只需要 3 步工作：

1. 调用 GetDeviceCaps 并检查 RC_PALETTE 来核实是否支持调色板。如果没有设置 RC_ PALETTE，调色板动画不会执行。
2. 创建一个逻辑调色板，其中包含想要使用的动画颜色，并用 PC_RESERVED 标志给每个调色板输入项作标记。只有标记为 PC_RESERVED 的调色板输入项才能用于调色板动画。
3. 使用逻辑调色板中的颜色绘制一个图像，然后调用 CPalette::AnimatePalette 反复修改调色板颜色。用 AnimatePalette 每修改调色板颜色一次，图像中的颜色就会作出相应的修改。

图 15-3 和图 15-4 所示的 LivePalette 应用程序说明了调色板动画的工作原理。窗口背景

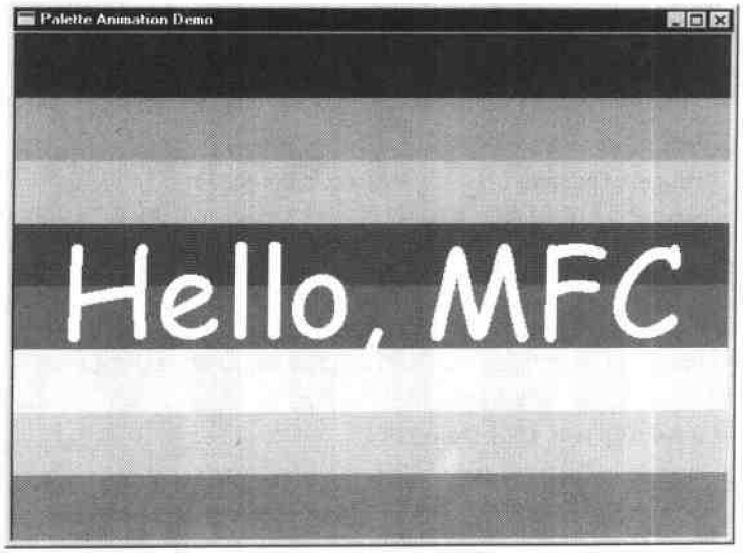

图 15-3　LivePalette 窗口

是用从逻辑调色板的 PC_RESERVED 输入项中得到的一组颜色(8 种不同的颜色)绘制的。用 PALETTEINDEX 值指定了画刷的颜色。用 PALETTERGB 值也可以,但是用普通 RGB 值就不行了,因为那些颜色要变化的像素用逻辑调色板中标记为 PC_RESERVED 的颜色绘制的,而不能用静态颜色。LivePalette 设置了一个时间间隔为 500 毫秒的计数器,其响应函数 OnTimer 用来改变调色板:

```
PALETTEENTRY pe[8];
m_palette.GetPaletteEntries (7, 1, pe);
m_palette.GetPaletteEntries (0, 7, &pe[1]);
m_palette.AnimatePalette (0, 8, pe);
```

调用 CPalette::GetPaletteEntries 并用逻辑调色板中的值以此初始化 PALETTEENTRY 结构数组,并同时将每种颜色向上循环一个位置,使得颜色 7 变为颜色 0、颜色 0 变为颜色 1 等等。AnimatePalette 然后直接将数组中的值复制到系统调色板中相应的输入项从而更新了屏幕上的颜色。没必要去调用 RealizePalette,因为已经执行过等价的调色板实现了。

剩下的程序就与上节中的 PaletteDemo 程序相似了,但还有一个值得注意的例外情况:如果 RC_PALETTE 为 NULL,InitInstance 将显示一个消息框,提示用户当前环境不支持调色板动画,并返回 FALSE 而关闭应用程序。除了 256 色视频模式以外,在其他模式下运行 LivePalette 都会得到一个消息框。

LivePalette.h

```
class CMyApp : public CWinApp
```

```
};
public:
    virtual BOOL InitInstance ();
};

class CMainWindow : public CFrameWnd
{
protected:
    CPalette m_palette;
    void DoBkgndFill (CDC * pDC, LPRECT pRect);
    void DoDrawText (CDC * pDC, LPRECT pRect);

public:
    CMainWindow ();

protected:
    afx_msg int OnCreate (LPCREATESTRUCT lpcs);
    afx_msg BOOL OnEraseBkgnd (CDC * pDC);
    afx_msg void OnPaint ();
    afx_msg void OnTimer (UINT nTimerID);
    afx_msg BOOL OnQueryNewPalette ();
    afx_msg void OnPaletteChanged (CWnd * pFocusWnd);
    afx_msg void OnDestroy ();
    DECLARE_MESSAGE_MAP ()
};
```

LivePalette.cpp

```
#include <afxwin.h>
#include "LivePalette.h"

CMyApp myApp;

////////////////////////////////////////////////////////////////////////////
// CMyApp member functions

BOOL CMyApp::InitInstance ()
{
    //
    // Verify that the host system is running in a palettized video mode.
    //
    CClientDC dc (NULL);
    if ((dc.GetDeviceCaps (RASTERCAPS) & RC_PALETTE) == 0) {
        AfxMessageBox (_T ("Palette animation is not supported on this "\
            "device. Set the color depth to 256 colors and try again."),
            MB_ICONSTOP|MB_OK);
        return FALSE;
```

```
    }

    //
    // Initialize the application as normal.
    //
    m_pMainWnd = new CMainWindow;
    m_pMainWnd->ShowWindow (m_nCmdShow);
    m_pMainWnd->UpdateWindow ();
    return TRUE;
}

/////////////////////////////////////////////////////////////////////////////
// CMainWindow message map and member functions

BEGIN_MESSAGE_MAP (CMainWindow, CFrameWnd)
    ON_WM_CREATE ()
    ON_WM_ERASEBKGND ()
    ON_WM_PAINT ()
    ON_WM_TIMER ()
    ON_WM_QUERYNEWPALETTE ()
    ON_WM_PALETTECHANGED ()
    ON_WM_DESTROY ()
END_MESSAGE_MAP ()

CMainWindow::CMainWindow ()
{
    Create (NULL, _T ("Palette Animation Demo"));
}

int CMainWindow::OnCreate (LPCREATESTRUCT lpcs)
{
    static BYTE bColorVals[8][3] = {
        128, 128, 128, // Dark Gray
        0, 0, 255, // Blue
        0, 255, 0, // Green
        0, 255, 255, // Cyan
        255, 0, 0, // Red
        255, 0, 255, // Magenta
        255, 255, 0, // Yellow
        192, 192, 192 // Light gray
    };
     if (CFrameWnd::OnCreate (lpcs) == -1)
        return -1;

    //
    // Create a palette to support palette animation.
    //
```

```
    struct
        LOGPALETTE lp;
        PALETTEENTRY ape[7];
    } pal;

    LOGPALETTE* pLP = (LOGPALETTE*) &pal;
    pLP->palVersion = 0x300;
    pLP->palNumEntries = 8;

    for (int i = 0; i < 8; i++) {
        pLP->palPalEntry[i].peRed = bColorVals[i][0];
        pLP->palPalEntry[i].peGreen = bColorVals[i][1];
        pLP->palPalEntry[i].peBlue = bColorVals[i][2];
        pLP->palPalEntry[i].peFlags = PC_RESERVED;
    }

    m_palette.CreatePalette (pLP);

    //
    // Program a timer to fire every half-second.
    //
    SetTimer (1, 500, NULL);
    return 0;
}

void CMainWindow::OnTimer (UINT nTimerID)
{
    PALETTEENTRY pe[8];
    m_palette.GetPaletteEntries (7, 1, pe);
    m_palette.GetPaletteEntries (0, 7, &pe[1]);
    m_palette.AnimatePalette (0, 8, pe);
}

BOOL CMainWindow::OnEraseBkgnd (CDC* pDC)
{
    CRect rect;
    GetClientRect (&rect);

    CPalette* pOldPalette;
    pOldPalette = pDC->SelectPalette (&m_palette, FALSE);
    pDC->RealizePalette ();
    DoBkgndFill (pDC, &rect);

    pDC->SelectPalette (pOldPalette, FALSE);
    return TRUE;
}

void CMainWindow::OnPaint ()
{
```

```
    CRect rect;
    GetClientRect (&rect);
    CPaintDC dc (this);
    DoDrawText (&dc, &rect);
}

BOOL CMainWindow::OnQueryNewPalette ()
{
    CClientDC dc (this);
    dc.SelectPalette (&m_palette, FALSE);

    UINT nCount;
    if (nCount = dc.RealizePalette ())
        Invalidate ();

    return nCount;
}

void CMainWindow::OnPaletteChanged (CWnd * pFocusWnd)
{
    if (pFocusWnd != this) {
        CClientDC dc (this);
        dc.SelectPalette (&m_palette, FALSE);
        if (dc.RealizePalette ())
            Invalidate ();
    }
}

void CMainWindow::OnDestroy ()
{
    KillTimer (1);
}

void CMainWindow::DoBkgndFill (CDC * pDC, LPRECT pRect)
{
    CBrush * pBrush[8];
    for (int i = 0; i < 8; i++)
        pBrush[i] = new CBrush (PALETTEINDEX (i));
    int nWidth = pRect -> right - pRect -> left;
    int nHeight = (pRect -> bottom - pRect -> top) / 8;

    CRect rect;
    int y1, y2;

    for (i = 0; i < 8; i++) {
        y1 = i * nHeight;
        y2 = (i == 7) ? pRect -> bottom - pRect -> top : y1 + nHeight;
        rect.SetRect (0, y1, nWidth, y2);
        pDC -> FillRect (&rect, pBrush[i]);
```

```
}

for (_ = 0; i < 8; _++)
    delete pBrush[i];
}

void CMainWindow::DoDrawText (CDC * pDC, LPRECT pRect)
{
    CFont font;
    font.CreatePointFont (720,_T ("Comic Sans MS"));

    pDC -> SetBkMode (TRANSPARENT);
    pDC -> SetTextColor (RGB (255, 255, 255));

    CFont * pOldFont = pDC -> SelectObject (&font);
    pDC -> DrawText ( _T ("Hello, MFC"), -1, pRect, DT_SINGLELINE|
        DT_CENTER|DT_VCENTER);
    pDC -> SelectObject (pOldFont);
}
```

图 15-4　LivePalette 应用程序

15.1.10　::SetSystemPaletteUse 函数

关于调色板的使用，最后要讲的是：如果您的应用程序绝对需要使用整个系统调色板，而不仅仅是使用添加了静态颜色后剩下的未使用的输入项，那么就可以用设备描述表句柄和 SYSPAL_NOSTATIC 参数来调用::SetSystemPaletteUse，以将静态颜色从 20 种减少到两种（黑色和白色）。在 256 色视频适配器上，这就意味着可以有 254 种而不是 236 种颜色从逻辑调色板复制到系统调色板。Win32 API 文献中很明白地介绍了::SetSystemPaletteUse 和它的对应函数::GetSystemPaletteUse 的用法，在这里就不过多介绍了。但是您必须意识到用自己的颜色替换静态颜色是极端不友好的行为，这样可能会毁坏整个系统中的标题栏、按钮以及其他窗口元素的颜色，除非很必要，否则请不要这么做。

15.2　位图

位映射图像，简称“位图”，是重要的一种现代计算机图形，它使得计算机可以用 1 和 0 的形式保存复杂的图像。在 Windows 中，位图是 GDI 对象，它像字体、画刷、画笔和其他 GDI 对象一样具有相当高的处理级别。可以用绘图程序创建位图，把它们作为资源嵌入应用程序的 EXE 文件，并用简单的函数加载它们；也可以使用 GDI 函数随意创建位图，把它们绘制到内存中的虚拟显示平面上。一旦创建了位图，就可以通过几个简单的函数调用把它显示

在屏幕上或复制到打印机上了。

在 32 位 Windows 中支持两类位图:“设备相关位图”(DDB)和“设备无关位图”(DIB)。另外还支持一种在设备无关位图基础上的变体,首先是在 Windows NT 中引入的,程序员通常称为“DIB 扩展”。DDB 是最简单也是功能最有限的位图。它们碰巧也是唯一一种 MFC 彻底封装的位图。我们将按以下顺序来学习它们的基本内容:首先从 CBitmap 和 DDB 开始,然后再学习功能强大的 DIB 和 DIB 扩展。在本章中我们常用术语“位图”来指代更专门的术语 DDB、DIB 和 DIB 扩展。到底位图指的是哪种(或属于哪种)具体的位图,应该从文章的上下文去理解。

15.2.1 DDB 和 CBitmap 类

众所周知,在使用位图之前必须要首先创建它。创建位图的一种途径是构造 CBitmap 对象并调用 CBitmap::CreateCompatibleBitmap:

```
CBitmap bitmap;
bitmap.CreateCompatibleBitmap(&dc, nWidth, nHeight);
```

在本例中,dc 代表屏幕设备描述表,nWidth 和 nHeight 是以像素为单位的位图尺寸。CreateCompatibleBitmap 要求一个设备描述表指针,这是因为所生成的 DDB 的格式与输出设备的结构紧密相关。提供一个设备描述表的指针,使 Windows 可以构造 DDB,使它与希望显示它的设备兼容。另一种创建位图的方法是调用 CBitmap::CreateBitmap 或 CBitmap::CreateBitmapIndirect,并指定颜色位面数和每个颜色位面上每个像素的位个数,所有这两个都是设备相关值。目前 CreateBitmap 和 CreateBitmapIndirect 的唯一实际用途是创建单色位图。有时单色位图即使在彩色环境中也很有用,本章中的一个示例程序将说明这一点。

用 CreateCompatibleBitmap 创建的 DDB 最初包含随机数据。如果想使用 DDB,如在屏幕中显示它们,就需要首先在其中绘制些内容。要使用 GDI 函数绘制位图,首先创建一个特殊类型的设备描述表,称为“内存设备描述表”,然后将位图选入内存 DC。实际上,选入内存设备描述表的位图会成为设备描述表的显示面,正如相应的屏幕 DC 是屏幕自身一样。下列程序创建了一个未初始化的 DDB,它具有 100 × 100 像素,然后再创建一个内存设备描述表,将位图选入其中,并将所有位图中的像素初始化为蓝色:

```
CClientDC dcScreen(this);
CBitmap bitmap;
bitmap.CreateCompatibleBitmap(&dcScreen, 100, 100);

CDC dcMem;
dcMem.CreateCompatibleDC(&dcScreen);

CBrush brush(RGB(0, 0, 255));
CBitmap* pOldBitmap = dcMem.SelectObject(&bitmap);
```

```
dcMem.FillRect (CRect (0, 0, 100. 100), &brush);
dcMem.SelectObject (pOldBitmap);
```

CDC::CreateCompatibleDC 创建一个与指定的设备描述表兼容的内存设备描述表。其中传递的地址是设备描述表的地址，通常指的是屏幕设备描述表，但是如果图像准备给打印机输出而不是输出到屏幕，那么也可以很容易地修改为打印机设备描述表。在位图选入内存设备描述表之后，就可以用给屏幕或打印机设备描述表绘制输出时所用的同样的 CDC 成员函数来对内存设备描述表（也是给位图）绘制图像了。

给内存设备描述表和给屏幕设备描述表绘图的巨大区别在于给内存绘制的图像不会显示出来。要显示它们，就必须将其从内存设备描述表复制到设备描述表。首先在内存设备描述表中绘制好，然后再将像素传送到屏幕上的方式非常便于在屏幕上复制几次相同的图像。不需要每次都重新绘制图像，在内存设备描述表上绘制一次，然后就可以任意次地将图像传送到屏幕设备描述表。（但是要注意，如果将图像从内存设备描述表到屏幕设备描述表复制一次，然后再按需要复制屏幕上已经存在的图像，这样的话对于许多显示器而言执行效果会更好。）在程序处理过程中位图起了很大的作用，因为当内存设备描述表首次被创建时，它仅包含一个可绘制的像素，该像素还是单色像素。把位图选入内存设备描述表之后就给您提供了更大的绘图显示面，并且只要位图不是单色的那就可以使用更多的颜色了。

15.2.2　按位将位图传送到屏幕和其他设备

怎样把位图绘制在屏幕上呢？不能把位图选入非内存 DC 中，如果这样做，SelectObject 将返回 NULL。但是可以使用 CDC::BitBlt 或 CDC::StretchBlt 将像素从内存设备描述表“位块传送”到屏幕设备描述表上。BitBlt 可以从一个设备描述表到另一个设备描述表传送一大块像素并保留像素块的尺寸。StretchBlt 可以在设备描述表之间传送成块的像素，还可以按要求缩放块尺寸。如果 dcMem 是一个内存设备描述表，包含有 100 × 100 个像素的图像，而且 dcScreen 是屏幕设备描述表，则语句

```
dcScreen.BitBlt (0, 0, 100, 100, &dcMem, 0, 0, SRCCOPY);
```

可以将图像复制给屏幕使其在屏幕上显示。传递给 BitBlt 的前两个参数指定了图像的左上角在目标（屏幕）设备描述表中的坐标，接下来的两个参数指定了所传送像素块的宽度和高度，第 5 个参数是指向源（内存）设备描述表的指针，第 6、第 7 个参数指定了在源 DC 中像素块左上角的坐标，第 8 个即最后一个参数指定了在传送中所用的光栅操作类型。SRCCOPY 将像素从内存设备描述表复制到屏幕设备描述表而不进行任何修改。

使用 StretchBlt 可以在位块传送时将位图缩小或放大。StretchBlt 的参数列表与 BitBlt 的很相似，但它还包含一对附加的参数用来指定图像缩放后的宽度和高度。下列语句将一个

100 × 100 的图像从内存设备描述表位块传送到屏幕设备描述表,并将该图像伸展来适合一个50 × 200的矩形:

```
dcScreen.StretchBlt (0, 0, 50, 200, &dcMem, 0, 0, 100, 100, SRCCOPY);
```

在默认状态下,当目标 DC 中的图像宽度和高度小于源 DC 中的宽度和高度时,在结果图像中相应的行列像素就会被简单地删除。在调用 StretchBlt 之前可以调用 CDC::SetStretchBltMode 来指定其他伸展模式,这些模式采用了多种方法来保留被丢弃的颜色信息。参考有关 SetStretchBltMode 的文献可以得到更详细的说明,这里要提醒的是最有可能用到的伸展模式 HALFTONE(使用抖动的方法来模拟不能直接显示的颜色)只能用在 Windows NT 和 Windows 2000 中,而不能用在 Windows 95 和 Windows 98 中。

通过给 CBitmap::GetBitmap 传递一个指向 BITMAP 结构的指针,可以获取有关位图的信息。BITMAP 的定义如下:

```
typedef struct tagBITMAP {
    LONG    bmType;
    LONG    bmWidth;
    LONG    bmHeight;
    LONG    bmWidthBytes;
    WORD    bmPlanes;
    WORD    bmBitsPixel;
    LPVOID    bmBits;
} BITMAP;
```

bmType 字段总是为 0。bmWidth 和 bmHeight 以像素为单位指定了位图的尺寸。bmWidthBytes 以字节为单位指定了位图中每行的长度并且总是 2 的倍数,因为位的行数以 16 位为边界进行了拆分。bmPlanes 和 bmBitsPixel 指定了颜色位面数以及每个颜色位面中每个位的像素数。如果 bm 是已初始化了的 BITMAP,则可以使用以下语句来确定位图可包含的颜色的最大数量:

```
int nColors = 1 << (bm.bmPlanes * bm.bmBitsPixel);
```

最后,在调用 GetBitmap 之后,如果位图是 DDB 则 bmBits 包含一个 NULL 指针。如果 bitmap 表示 CBitmap 对象,语句

```
BITMAP bm;
bitmap.GetBitmap (&bm);
```

就会用有关位图的信息初始化 bm。

由 GetBitmap 返回的位图尺寸是以设备单位(像素)表示的,可是 BitBlt 和 StretchBlt 都使用逻辑单位。如果您想编写一个一般的 DrawBitmap 函数给设备描述表位块传送位图,那就必须预见到设备描述表被设置为非 MM_TEXT 映射模式的可能性。下列 DrawBitmap 函数,

设计作为 CBitmap 派生类的成员函数，可以在所有的映射模式下执行。pDC 指向位图正在被位块传送到的目标设备描述表；x 和 y 指定了图像左上角在目标设备中的位置：

```
void CMyBitmap::DrawBitmap (CDC * pDC, int x, int y)
{
    BITMAP bm;
    GetBitmap (&bm);
    CPoint size (bm.bmWidth, bm.bmHeight);
    pDC -> DPtoLP (&size);

    CPoint org (0, 0);
    pDC -> DPtoLP (&org);

    CDC dcMem;
    dcMem.CreateCompatibleDC (pDC);
    CBitmap * pOldBitmap = dcMem.SelectObject (this);
    dcMem.SetMapMode (pDC -> GetMapMode ());
    pDC -> BitBlt (x, y, size.x, size.y, &dcMem, org.x, org.y, SRCCOPY);
    dcMem.SelectObject (pOldBitmap);
}
```

MFC 的 CDC::DPtoLP 函数对 CSize 对象的处理有些令人迷惑，保存位图尺寸的 size 变量是 CPoint 对象而不是 CSize 对象。当您给 CDC::DPtoLP 传递一个 CPoint 对象的地址时，就会顺利地调用 ::DPtoLP API 函数并恰当地实现转换，即使得到一个以上为负的坐标值。但是当给 CDC::DPtoLP 传递 CSize 对象的地址时，MFC 会亲自执行转换并将所有负值都转换为正值。可能直觉上讲尺寸值不应该是负值，但是那确实是 y 轴指向上的映射模式下 BitBlt 期望得到的结果。

15.2.3　位图资源

如果仅仅是显示一个预定义的位图（该位图是用 Visual C++ 资源编辑器或其他画图程序或图像编辑器生成的 BMP 文件），那么就可以用类似于下面给出的语句将位图资源添加到应用程序的 RC 文件中：

```
IDB_MYLOGO BITMAP Logo.bmp
```

然后再加载它，如下：

```
CBitmap bitmap;
bitmap.LoadBitmap (IDB_MYLOGO);
```

在本例中，IDB_MYLOGO 是位图的整数资源 ID，Logo.bmp 包含位图的文件名称。还可以赋予位图资源一个字符串 ID 并按以下方式加载：

```
bitmap.LoadBitmap (_T ("MyLogo"));
```

LoadBitmap 接受所有类型的资源 ID。在加载了位图资源之后,就可以用显示其他位图的方法显示它了,将其选入内存设备描述表并位块传送到屏幕设备描述表。在 Visual C++ 启动时可以看到的那种显眼的欢迎屏幕通常被保存为位图资源,它们就是用 LoadBitmap(或其等价的 API 函数 ::LoadBitmap) 在显示之前加载的。

CBitmap 包括一个相关的成员函数 LoadMappedBitmap,用来加载位图资源并将位图中的一种以上的颜色变换为指定的颜色。LoadMappedBitmap 是对 ::CreateMappedBitmap 的封装,该函数添加到 API 是为了将用来绘制所有者画按钮、工具栏和其他控件的位图颜色在位图加载时变换为系统颜色。语句

```
bitmap.LoadMappedBitmap (IDB_BITMAP);
```

加载了一个位图资源并自动地实现了以下颜色的变换:黑色像素变换为系统色 COLOR_BTNTEXT;深灰色(R = 128,G = 128,B = 128) 像素变换为 COLOR_BTNSHADOW;浅灰色(R = 192,G = 192,B = 192) 像素变换为 COLOR_BTNFACE;白色像素变换为 COLOR_BTNHIGHLIGHT;深蓝色(R = 0,G = 0,B = 128) 像素变换为 COLOR_HIGHLIGHT;以及品红(R = 255,G = 0,B = 255) 像素变换为 COLOR_WINDOW。将品红映射给 COLOR_WINDOW 是因为用品红色绘制位图可以得到"透明"像素。如果 LoadMappedBitmap 将品红像素变换为了 COLOR_WINDOW 像素,并且位图显示在具有 COLOR_WINDOW 颜色的背景上,则重新映射的像素在窗口背景上将是不可见的。

通过给 LoadMappedBitmap 传递一个指向 COLORMAP 结构型数组的指针可以实现自定义颜色转换,该结构指定了想要转换的源颜色和转换到的目标颜色。自定义颜色映射的一个用途是可以用来模拟透明像素,可以把任意背景颜色变换为您自己选择的背景颜色。在本章稍后,我们将研究一种用透明像素绘制位图的技术,它对于任何背景(即使是那些不是原色的背景)都适用而且不需要颜色映射。

15.2.4 DIB 和 DIB 分区

设备相关位图存在的问题是——它们是设备相关的。您可以使用 CBitmap::GetBitmapBits 和 CBitmap::SetBitmapBits 直接操纵 DDB 中的各个位,但是由于像素颜色数据是以设备相关的格式保存的,所以除非位图是单色的,否则很难知道如何去使用由 GetBitmapBits 返回的数据(或给 SetBitmapBits 传递怎样的数据)。更糟糕的是,在 DDB 中保存的颜色信息只对显示它的设备驱动器有意义。如果将 DDB 写入某个 PC 的磁盘然后在另一个 PC 上将其读出,很容易看到结果颜色不再相同了。DDB 很适合于加载和显示位图资源(虽然在位图资源包含的颜色比硬件设备显示的颜色多时会得到较差的输出效果),也适合于给输出设备提交位图之前在内存设备描述表上绘制它们。但由于可移植性差,而不适用于其他工作。

这就是 Windows 3.0 引入设备无关位图(DIB)的原因。术语"DIB"描述了用来保存位图数据的设备无关格式,该格式在显示驱动器描述表外部甚至在 Windows 本身之外都是有意

义的。在调用::CreateBitmap（与 API 等价的是 CBitmap::CreateBitmap）创建一个位图时，会得到一个 HBITMAP 句柄。而调用::CreateDIBitmap 创建位图时，也会得到一个 HBITMAP 句柄。不同之处在于其内部。传递给::CreateBitmap 的像素数据是以设备相关格式保存的，而传递给::CreateDIBitmap 的数据是以 DIB 格式保存的。而且，DIB 格式还包含其他颜色信息，可使不同的设备驱动程序输出一致的颜色。API 包括一对函数::GetDIBits 和::SetDIBits，分别用来读取和写入 DIB 格式位。它还包括一些将应用程序缓冲区中的原始 DIB 数据渲染输出到设备的函数。Windows BMP 文件是以 DIB 格式存储的，所以很容易用::CreateDIBitmap 编写一个函数将 BMP 文件的内容转换为 GDI 位图对象。

DIB 扩展与 DIB 相似，它曾用来解决 Windows NT 中::StretchDIBits 函数的执行效率问题。一些图形程序会分配缓冲区来存储 DIB 位，然后用::StretchDIBits 将这些位直接渲染输出到屏幕。程序并没有将位传递给::CreateDIBitmap 并生成 HBITMAP，但它直接访问了位图数据并在屏幕上显示了位图。不幸的是，Windows NT 和 Windows 2000 客户/服务器结构要求：从客户端缓冲区中以块方式传递的位应该在传送给帧缓冲区之前复制到服务器端缓冲区内，而这些额外的开销使::StretchDIBits 执行起来很迟钝。

为了适应这种体系结构，Windows NT 并发组发明了 DIB 扩展。DIB 扩展是 Windows NT 和 Windows 2000 中的独立功能：可以在 DC 中选入 DIB 扩展并将它按块传送给屏幕（这样可以避免不必要的内存到内存的移动），但是也可以直接访问位图位。Windows 95 和 Windows 98 中对于::StretchDIBits 函数速度不是大问题，因为这些系统的体系结构与 Windows NT 和 Windows 2000 的不同，但是 Windows 95 和 Windows 98 与 Windows NT 和 Windows 2000 一样都支持 DIB 扩展，而且还为处理它们提供了一些便于使用的 API 函数。鼓励 Win32 程序员尽可能地在用到普通的 DIB 和 DDB 的地方都使用 DIB 扩展，这样可以给予操作系统处理位图数据的最大灵活性。

有关 DIB 和 DIB 扩展的问题是目前 MFC 版本并没有封装它们。要在 MFC 应用程序中使用 DIB 和 DIB 扩展，您必须求助于 API 或者编写自己的类来封装相关的 API 函数。为 DIB 和 DIB 扩展编写基本的 CDib 类或包含别的函数来扩展 CBitmap 类都不困难，但是这里不打算讨论它们了，因为将来的 MFC 版本中很可能包含一组全面的代表 DIB 和 DIB 扩展的类。现在要讲述的是如何发挥 MFC 的 CBitmap 类的最大能力，将 CBitmap 和 API 函数结合起来使普通的 CBitmap 具有类似 DIB 的功能。

15.2.5　位块传送、光栅操作以及颜色映射

CDC::BitBlt 常用来将位图按块传送到屏幕，但是 BitBlt 的用途不仅限于传送原始位。实际上，它是一个很复杂的函数。它通过逻辑运算算出所输出像素的颜色，进而将来自源 DC、目标 DC 和当前所选目标 DC 画刷的像素合成。光栅操作码 SRCCOPY 很简单；它只是将像素从来源复制到目的地。其他光栅操作码就没有这样简单了，例如：MERGEPAINT 将源像素的颜色用“非”运算进行反转，并把结果与目标像素颜色进行“或”运算。BitBlt 支持所有

的 256 种光栅操作码。对于表 15-3 给出的 15 个光栅操作码,在 Wingdi.h 中已经用 # define 语句定义了名称。

表 15-3 BITBLT 的光栅操作码

名字	等价二进制值	执行的操作
SRCCOPY	0xCC0020	dest = source
SRCPAINT	0xEE0086	dest = source OR dest
SRCAND	0x8800C6	dest = source AND dest
SRCINVERT	0x660046	dest = source XOR dest
SRCERASE	0x440328	dest = source AND (NOT dest)
NOTSRCCOPY	0x330008	dest = (NOT source)
NOTSRCERASE	0x1100A6	dest = (NOT src) AND (NOT dest)
MERGECOPY	0xC000CA	dest = (source AND pattern)
MERGEPAINT	0xBB0226	dest = (NOT source) OR dest
PATCOPY	0xF00021	dest = pattern
PATPAINT	0xFB0A09	dest = pattern OR (NOT src) OR dest
PATINVERT	0x5A0049	dest = pattern XOR dest
DSTINVERT	0x550009	dest = (NOT dest)
BLACKNESS	0x000042	dest = BLACK
WHITENESS	0xFF0062	dest = WHITE

通过对下面列表中的位值执行必要的逻辑运算,并按其结果在 Microsoft 的 Platform SDK 中“ Ternary Raster Operations "部分查找相应的 DWORD 型光栅操作码,可以派生出自定义的光栅操作码。

Pat 1 1 1 1 0 0 0 0

Src 1 1 0 0 1 1 0 0

Dest 1 0 1 0 1 0 1 0

Pat (代表“模式”) 表示选入目标设备描述表的画刷的颜色;Src 代表源设备描述表中像素颜色;Dest 代表目标设备描述表中像素颜色。若想得到用来反转源位图的光栅操作码,那么就要将源位图中的像素与目标像素进行“与”运算,并将结果与画刷颜色进行“或”运算。首先对列表中每一列位都进行相同的运算。结果如下:

Pat 1 1 1 1 0 0 0 0

Src 1 1 0 0 1 1 0 0

Dest 1 0 1 0 1 0 1 0

1 1 1 1 0 0 1 0 = 0xF2

在三元组光栅操作码表中查找 0xF2，就会找到完整的光栅操作码是 0xF20B05。因此，就可以给 BitBlt 传递一个十六进制值 0xF20B05 以取代 SRCCOPY 或其他光栅操作码，进而执行以上假设的光栅操作了。

那么所有这些光栅操作码可以用来干什么呢？实际上在彩色环境下其中大部分可能不会用到。除了 SRCCOPY 以外，最有用的光栅操作是 SRCAND、SRCINVERT 和 SRCPAINT　但是正如下节中示例程序说明的那样，使用未命名的光栅操作码有时会减少实现期望输出结果所必须的操作步骤。

BitBlt 是 CDC 位块传送函数家族中的一员，其中还包括 StretchBlt（前面已经讨论过了）、PatBlt、MaskBlt 和 PlgBlt。PatBlt 将目标设备描述表中矩形内的像素与选入设备描述表中的画刷结合，主要用于复制不使用源设备描述表的 BitBlt 光栅操作的一个子集。MaskBlt 将源设备描述表和目标 DC 中的像素结合起来，并使用单色位图作为掩码。一种光栅操作（“前景”光栅操作）只在掩码为 1 的像素上执行，另一种光栅操作（“背景”光栅操作）只在掩码为 0 的像素上执行。PlgBlt 将源设备描述表上的矩形像素块按钮传送到目标 DC 上的平行四边形中，并且在传送过程中可使用单色位图作为掩码。相应于掩码中所有 1 的像素被位块传送到平行四边形中；而不传送相应于掩码中所有 0 的像素。不幸的是，MaskBlt 和 PlgBlt 只在 Windows NT 3.51 和更高版本以及 Windows 2000 中得到了支持，而 Windows 95 和 Windows 98 并不支持该函数。如果在 Windows 95 或 Windows 98 中调用它们，会得到返回值 0，说明函数失败。

一些输出设备（特别是绘图仪）并不支持 BitBlt 和其他位块传送函数。为确定给定的设备是否支持 BitBlt，可以获取设备描述表并用 RASTERCAPS 参数调用 GetDeviceCaps。如果在返回值中设置了 RC_BITBLT 位，说明设备支持 BitBlts；如果还设置了 RC_STRETCHBLT 位，那么说明设备还支持 StretchBlts。其他位块传送函数就没有相应的 RASTERCAPS 位，但如果是在 Windows NT 上编写程序，并且发现不支持 BitBlt，那么就可以假定该设备也不支持 PatBlt、MaskBlt 和 PlgBlt。通常，那些不支持位块传送的绘图仪和其他矢量设备会在 GetDeviceCap返回的值中设置 RC_NONE 位，表面它们不支持任何类型的光栅操作。

当源设备描述表和目标设备描述表的颜色特性匹配时，BitBlt 和其他位块传送函数的输出效果最好（执行效率也最高）。如果将具有 256 色的位图块传送到 16 色的目标 DC，Windows必须将源设备描述表中的颜色映射为目标设备描述表中的颜色。可是有些时候我们可以利用颜色映射。在用 BitBlt 将单色位图块传送到彩色 DC 上时，它就会把 0 位转换为目标 DC 的当前前景颜色（CDC::SetTextColor），将 1 位转换为目标设备描述表的当前背景颜色（CDC::SetBkColor）。反过来，当将彩色位图位块传送到单色 DC 上时，BitBlt 会把与目标设备描述表背景颜色匹配的像素转换为 1 而将所有其他像素转换为 0。可以使用后一种颜色映射的形式根据彩色位图创建单色掩码，可以在例程中使用该掩码将某种颜色以外的所有像素从位图位块传送到屏幕设备描述表，结果可以在位图中生成透明像素。

听上去很有趣吧？通过为每个图标图像都保存两个位图就可以在图标中实现透明像素

了：一个单色“与”掩码和一个彩色“异或”掩码。只要编写一个输出例程，使用 BitBlts 和光栅操作来任意创建“与”和“异或”掩码，就可以用透明像素绘制位图了。下节中的 BitmapDemo 示例程序说明了具体的实现方法。

15.2.6 BitmapDemo 应用程序

BitmapDemo 是用 AppWizard 创建的非文档/视图类型的应用程序，用来说明加载位图资源和将其位块传送到屏幕的方法。它还说明了如何有效地利用 BitBlts 和光栅操作码，通过在位图中指定一种颜色作为透明颜色来位块传送不规则形状的图像。程序的输出由位图矩形阵列组成，它们被绘制在从蓝色到黑色连续变化的背景上。当 Options 菜单中的 Draw Opaque 被选中时，对按块传送到屏幕上的位图不做修改，产生如图 15-5 所示的结果。如果选中 Draw Transparent，在位图按块传送到屏幕时，其中的红色像素就会被删除，结果如图 15-6 所示。

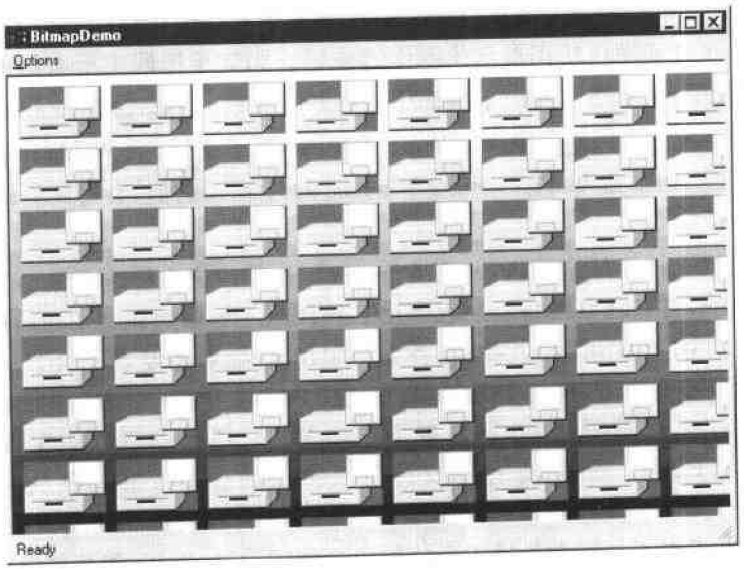

图 15-5 透明显示失效时的 BitmapDemo 窗口

BitmapDemo 使用了 CBitmap 的派生类 CMaskedBitmap 来代表位图。CMaskedBitmap 包含了两个 CBitmap 不具有的成员函数：一个 Draw 函数用来将位图按块传送到 DC，以及一个 DrawTransparent 函数用来把位图在按块传送到 DC 的同时滤掉指定颜色的像素。有了 CMaskedBitmap 的帮助，下列语句

```
CMaskedBitmap bitmap;
bitmap.LoadBitmap(IDB_BITMAP);
bitmap.Draw(pDC, x, y);
```

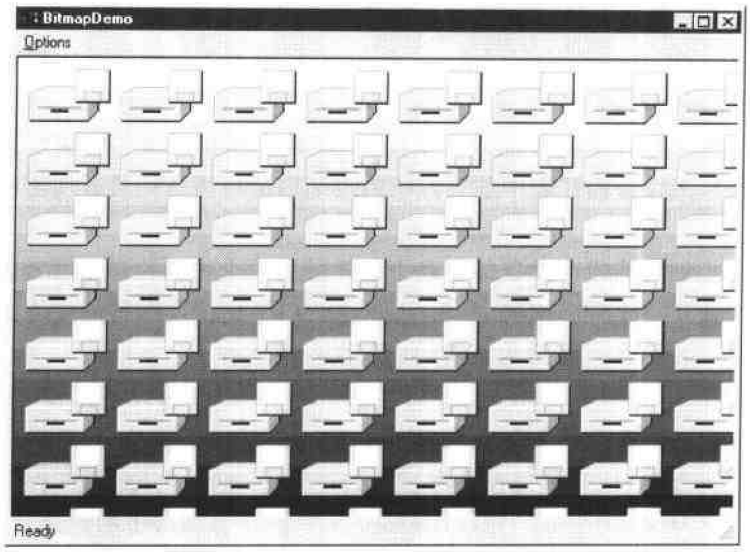

图 15-6　透明显示有效时的 BitmapDemo 窗口

就可完成创建一个位图对象、加载位图资源并在 pDC 表示的设备上输出的工作。x 和 y 参数指定了位图左上角的位置。语句

```
CMaskedBitmap bitmap;
bitmap.LoadBitmap (IDB_BITMAP);
bitmap.DrawTrarsparent (pDC, x, y, RGB (255, 0, 255));
```

完成的工作基本相同，就是没有对位图中红色 RGB (255, 0, 255)的像素执行位块传送。在 CMaskedBitmap 的帮助下，很容易绘制带有“孔洞”或具有非矩形轮廓的位图：只要赋给位图中所有透明像素一个公用的颜色，并将该颜色传递给 DrawTransparent 即可。DrawTransparent 就不会将透明像素与其他像素一起位块传送出去。

您应该对 CMaskedBitmap::Draw 的源程序比较熟悉：它与以前讲述的 DrawBitmap 函数相同。而 CMaskedBitmap::DrawTransparent 稍微有点复杂。但源程序中的注释可以帮助您理解它的实现过程。为防止注释解释得不够明白，我将涉及到将位图位块传送到屏幕的同时忽略某种颜色像素的处理过程归纳为下面几个步骤：

1. 创建内存设备描述表，并将位图选入其中。
2. 创建第 2 个内存设备描述表，并选入一个单色位图，该位图的大小与初始位图相同。创建一个“与”掩码：将第一步中创建的内存设备描述表的背景颜色设置为透明颜色，并将位图位块传送到设备描述表。得到的“与”掩码在初始位图中像素颜色与透明颜色相同的地方具有 1，其他地方都为 0。
3. 创建第 3 个内存 DC，选入一个大小和颜色属性与初始位图匹配的位图。在这个设

备描述表上创建"异或"掩码：首先用 SRCCOPY 光栅操作码将位图从第一步中创建的内存 DC 中位块传送到这个 DC 上，然后再用光栅操作码 0x220326 将"与"掩码位块传送到 DC。

4. 创建第 4 个内存 DC，并选入一个位图，其大小和颜色属性与初始位图匹配。将像素从在输出设备描述表中放置位图的矩形内位块传送到最近创建的内存设备描述表中。
5. 在第 4 步创建的内存设备描述表中创建最后一个图像：首先用 SRCAND 光栅操作码位块传送一个"与"掩码，然后用 SRCINVERT 光栅操作码位块传送一个掩码。
6. 从内存设备描述表将图像复制到输出设备描述表。

注意第 2 步是如何使用 BitBlt 生成"与"掩码的。由于目标设备描述表是单色的，所以 GDI 就在目标设备描述表上将其颜色等于背景颜色的像素转换为 1，而将其他像素转换为 0。首先将源设备描述表的背景色设置为位图的透明色，这很重要，它能保证变换的顺利进行。如果看一下 CMaskedBitmap::DrawTransparent 中相应于第二步的程序代码，您将看到目标设备描述表的大小和颜色属性是用 CBitmap::CreateBitmap 设置的，它生成一个单色位图，其尺寸等于初始位图的尺寸，然后将位图选入设备描述表。通过选取位图，您可以控制内存设备描述表显示面的大小和它支持的颜色种类。这就是在 DrawTransparent 中看到如此众多的对 CreateBitmap 和 CreateCompatibleBitmap 调用的原因。

在 DrawTransparent 中另一个有趣的内容是在第 3 步中使用的光栅操作码 0x220326，它对源设备描述表和目标设备描述表中涉及到的像素执行了以下光栅操作：

```
dest = (NOT src) AND dest
```

可以通过使用"标准"的光栅操作码调用 BitBlt 两次来实现相同的任务：一次用光栅操作码 NOTSRCCOPY 将源设备描述表中的图像反转，另一次用 SRCAND 将反转后的图像与目标设备描述表中的像素进行"与"操作。调用 BitBlt 一次明显要比调用两次效率高，但是不要奇怪，在某些 PC 上使用 0x220326 光栅操作码并不比使用 NOTSRCCOPY 和 SRCAND 组合执行速度快。大多数显示驱动程序都对某种光栅操作进行了优化，因而它的执行速度要比其他操作执行快，通常 NOTSRCCOPY 或 SRCAND 光栅操作的执行速度非常快，而 0x220326 则较慢。

在运行 BitmapDemo 进行实验的时候(在图 15-7 中给出了该程序的源代码)，可以注意到当 BitmapDemo 绘制透明像素时窗口会用更多的时间进行重绘。这是因为对于将单个图像输出到屏幕的操作，DrawTransparent 要比 Draw 做更多的工作。特别是当 DrawTransparent 反复生成相同的"与"和"异或"掩码时，执行效率会达到最低点。如果想在应用程序中使用 DrawTransparent 的功能，而对于该应用程序执行效率又是关键(例如：如果想用透明位图创建一个可以在屏幕上到处移动的类似小图画的对象)，那么就应该修改 CMaskedBitmap 类使它只生成一次屏蔽码位图，以后在需要时重复使用。通过直接对目标 DC 而不是包含目标 DC 上像素拷贝的内存 DC 执行"与"和"异或"屏蔽码运算可以改善运行效率，但是使用掩码会

产生短暂延迟，如果您正在动画程序中使用位图，那么延迟造成的小量闪烁就可能会很明显。

MainFrm.h

```
// MainFrm.h : interface of the CMainFrame class
//
/////////////////////////////////////////////////////////////////////////////

#if !defined(
    AFX_MAINFRM_H__D71EF549_A6FE_11D2_8E53_006008A82731__INCLUDED_ )
#define AFX_MAINFRM_H__D71EF549_A6FE_11D2_8E53_006008A82731__INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "ChildView.h"

class CMainFrame : public CFrameWnd
{
public:
    CMainFrame();
protected:
    DECLARE_DYNAMIC(CMainFrame)

// Attributes
public:

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CMainFrame)
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    virtual BOOL OnCmdMsg(UINT nID, int nCode, void* pExtra,
        AFX_CMDHANDLERINFO* pHandlerInfo);
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CMainFrame();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected: // control bar embedded members
    CStatusBar  m_wndStatusBar;
    CChildView    m_wndView;
```

```
// Generated message map functions
protected:
    //{{AFX_MSG(CMainFrame)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnSetFocus(CWnd pOldWnd);
    afx_msg BOOL OnQueryNewPalette();
    afx_msg void OnPaletteChanged(CWnd * pFocusWnd);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////

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

#endif
// !defined(
//    AFX_MAINFRM_H__D71EF549_A6FE_11D2_8E53_006008A82731__INCLUDED_)
```

MainFrm.cpp

```
// MainFrm.cpp : implementation of the CMainFrame class
//

#include "stdafx.h"
#include "BitmapDemo.h"
#include "MaskedBitmap.h"
#include "MainFrm.h"

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

/////////////////////////////////////////////////////////////////////////////
// CMainFrame

IMPLEMENT_DYNAMIC(CMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    //{{AFX_MSG_MAP(CMainFrame)
    ON_WM_CREATE()
    ON_WM_SETFOCUS()
    ON_WM_QUERYNEWPALETTE()
    ON_WM_PALETTECHANGED()
```

```
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

static UINT indicators[] =
{
    ID_SEPARATOR
};

/////////////////////////////////////////////////////////////////////////////
// CMainFrame construction/destruction
CMainFrame::CMainFrame()
{
}

CMainFrame::~CMainFrame()
{
}

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;
    // create a view to occupy the client area of the frame
    if (!m_wndView.Create(NULL, NULL, AFX_WS_DEFAULT_VIEW,
        CRect(0, 0, 0, 0), this, AFX_IDW_PANE_FIRST, NULL))
    {
        TRACE0("Failed to create view window\n");
        return -1;
    }

    if (!m_wndStatusBar.Create(this) ||
        !m_wndStatusBar.SetIndicators(indicators,
            sizeof(indicators)/sizeof(UINT)))
    {
        TRACE0("Failed to create status bar\n");
        return -1; // fail to create
    }

    return 0;
}

BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    if( !CFrameWnd::PreCreateWindow(cs) )
        return FALSE;
    // TODO: Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs

    cs.dwExStyle &= ~WS_EX_CLIENTEDGE;
```

```
    cs.lpszClass = AfxRegisterWndClass(0);
    return TRUE;
}

/////////////////////////////////////////////////////////////////////////////
// CMainFrame diagnostics
#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
    CFrameWnd::AssertValid();
}

void CMainFrame::Dump(CDumpContext& dc) const
{
    CFrameWnd::Dump(dc);
}

#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CMainFrame message handlers
void CMainFrame::OnSetFocus(CWnd * pOldWnd)
{
    // forward focus to the view window
    m_wndView.SetFocus();
}

BOOL CMainFrame::OnCmdMsg(UINT nID, int nCode, void * pExtra,
    AFX_CMDHANDLERINFO * pHandlerInfo)
{
    // let the view have first crack at the command
    if (m_wndView.OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
        return TRUE;

    // otherwise, do default handling
    return CFrameWnd::OnCmdMsg(nID, nCode, pExtra, pHandlerInfo);
}

BOOL CMainFrame::OnQueryNewPalette()
{
    m_wndView.Invalidate();
    return TRUE;
}

void CMainFrame::OnPaletteChanged(CWnd * pFocusWnd)
{
    m_wndView.Invalidate();
}
```

ChildView.h

```
// ChildView.h : interface of the CChildView class
//
/////////////////////////////////////////////////////////////////////////////

#if !defined(
    AFX_CHILDVIEW_H__D71EF54B_A6FE_11D2_8E53_006008A82731__INCLUDED_)
#define AFX_CHILDVIEW_H__D71EF54B_A6FE_11D2_8E53_006008A82731__INCLUDED_

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

/////////////////////////////////////////////////////////////////////////////
// CChildView window

class CChildView : public CWnd
{
// Construction
public:
    CChildView();

// Attributes
public:

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CChildView)
    protected:
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CChildView();

    // Generated message map functions
protected:
    void DoGradientFill (CDC* pDC, LPRECT pRect);
    CPalette m_palette;
    CMaskedBitmap m_bitmap;
    BOOL m_bDrawOpaque;
    //{{AFX_MSG(CChildView)
    afx_msg void OnPaint();
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
```

```
    afx_msg BOOL OnEraseBkgnd(CDC * pDC);
    afx_msg void OnOptionsDrawOpaque();
    afx_msg void OnOptionsDrawTransparent();
    afx_msg void OnUpdateOptionsDrawOpaque(CCmdUI * pCmdUI);
    afx_msg void OnUpdateOptionsDrawTransparent(CCmdUI * pCmdUI);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////

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

#endif
// !defined(
//    AFX_CHILDVIEW_H__D71EF54B_A6FE_11D2_8E53_006008A82731__INCLUDED_)
```

ChildView.cpp

```
// ChildView.cpp : implementation of the CChildView class
//

#include "stdafx.h"
#include "BitmapDemo.h"
#include "MaskedBitmap.h"
#include "ChildView.h"

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

/////////////////////////////////////////////////////////////////////////////
// CChildView

CChildView::CChildView()
{
    m_bDrawOpaque = TRUE;
}
CChildView::~CChildView()
{
}

BEGIN_MESSAGE_MAP(CChildView,CWnd )
```

```
    //{{AFX_MSG_MAP(CChildView)
    ON_WM_PAINT()
    ON_WM_CREATE()
    ON_WM_ERASEBKGND()
    ON_COMMAND(ID_OPTIONS_DRAW_OPAQUE, OnOptionsDrawOpaque)
    ON_COMMAND(ID_OPTIONS_DRAW_TRANSPARENT, OnOptionsDrawTransparent)
    ON_UPDATE_COMMAND_UI(ID_OPTIONS_DRAW_OPAQUE, OnUpdateOptionsDrawOpaque)
    ON_UPDATE_COMMAND_UI(ID_OPTIONS_DRAW_TRANSPARENT,
        OnUpdateOptionsDrawTransparent)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CChildView message handlers

BOOL CChildView::PreCreateWindow(CREATESTRUCT& cs)
{
    if (!CWnd::PreCreateWindow(cs))
        return FALSE;

    cs.dwExStyle |= WS_EX_CLIENTEDGE;
    cs.style &= ~WS_BORDER;
    cs.lpszClass = AfxRegisterWndClass(CS_HREDRAW|CS_VREDRAW|CS_DBLCLKS,
        ::LoadCursor(NULL, IDC_ARROW), HBRUSH(COLOR_WINDOW+1), NULL);

    return TRUE;
}

void CChildView::OnPaint()
{
    CRect rect;
    GetClientRect(&rect);
    CPaintDC dc(this);

    BITMAP bm;
    m_bitmap.GetBitmap(&bm);
    int cx = (rect.Width() / (bm.bmWidth + 8)) + 1;
    int cy = (rect.Height() / (bm.bmHeight + 8)) + 1;

    int i, j, x, y;
    for (i = 0; i < cx; i++) {
        for (j = 0; j < cy; j++) {
            x = 8 + (i * (bm.bmWidth + 8));
            y = 8 + (j * (bm.bmHeight + 8));
            if (m_bDrawOpaque)
                m_bitmap.Draw(&dc, x, y);
            else
                m_bitmap.DrawTransparent(&dc, x, y, RGB(255, 0, 0));
        }
    }
```

```
}

int CChildView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CWnd::OnCreate(lpCreateStruct) == -1)
        return -1;

    //
    // Load the bitmap.
    //
    m_bitmap.LoadBitmap(IDB_BITMAP);

    //
    // Create a palette for a gradient fill if this is a palettized device.
    //
    CClientDC dc(this);
    if (dc.GetDeviceCaps(RASTERCAPS) & RC_PALETTE) {
        struct {
            LOGPALETTE lp;
            PALETTEENTRY ape[63];
        } pal;

        LOGPALETTE* pLP = (LOGPALETTE*) &pal;
        pLP->palVersion = 0x300;
        pLP->palNumEntries = 64;

        for (int i=0; i<64; i++) {
            pLP->palPalEntry[i].peRed = 0;
            pLP->palPalEntry[i].peGreen = 0;
            pLP->palPalEntry[i].peBlue = 255 - (i * 4);
            pLP->palPalEntry[i].peFlags = 0;
        }
        m_palette.CreatePalette(pLP);
    }
    return 0;
}
BOOL CChildView::OnEraseBkgnd(CDC* pDC)
{
    CRect rect;
    GetClientRect(&rect);

    CPalette* pOldPalette;
    if ((HPALETTE) m_palette != NULL) {
        pOldPalette = pDC->SelectPalette(&m_palette, FALSE);
        pDC->RealizePalette();
    }

    DoGradientFill(pDC, &rect);

    if ((HPALETTE) m_palette != NULL)
        pDC->SelectPalette(pOldPalette, FALSE);
```

```
    return TRUE;
}

void CChildView::DoGradientFill(CDC * pDC, LPRECT pRect)
{
    CBrush * pBrush[64];
    for (int i = 0; i < 64; i ++ )
        pBrush[i] = new CBrush (PALETTERGB (0, 0, 255 - (i * 4)));

    int nWidth = pRect -> right - pRect -> left;
    int nHeight = pRect -> bottom - pRect -> top;
    CRect rect;

    for (i = 0; i < nHeight; i ++ ) {
        rect.SetRect (0, i, nWidth, i + 1);
        pDC -> FillRect (&rect, pBrush[(i * 63) / nHeight]);
    }

    for (i = 0; i < 64; i ++ )
        delete pBrush[i];
}

void CChildView::OnOptionsDrawOpaque()
{
    m_bDrawOpaque = TRUE;
    Invalidate ();
}

void CChildView::OnOptionsDrawTransparent()
{
    m_bDrawOpaque = FALSE;
    Invalidate ();
}

void CChildView::OnUpdateOptionsDrawOpaque(CCmdUI * pCmdUI)
{
    pCmdUI -> SetCheck (m_bDrawOpaque ? 1 : 0);
}

void CChildView::OnUpdateOptionsDrawTransparent(CCmdUI * pCmdUI)
{
    pCmdUI -> SetCheck (m_bDrawOpaque ? 0 : 1);
}
```

MaskedBitmap.h

```
// MaskedBitmap.h: interface for the CMaskedBitmap class.
//
```

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

#if !defined(
    AFX_MASKEDBITMAP_H__D71EF554_A6FE_11D2_8E53_006008A82731__INCLUDED_)
#define AFX_MASKEDBITMAP_H__D71EF554_A6FE_11D2_8E53_006008A82731__INCLUDED_

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

class CMaskedBitmap : public CBitmap
{
public:
    void DrawTransparent (CDC * pDC, int x, int y,
        COLORREF clrTransparency);
    void Draw (CDC * pDC, int x, int y);
};

#endif
// !defined(
//    AFX_MASKEDBITMAP_H__D71EF554_A6FE_11D2_8E53_006008A82731__INCLUDED_)
```

MaskedBitmap.cpp

```
// MaskedBitmap.cpp: implementation of the CMaskedBitmap class.
//

////////////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "BitmapDemo.h"
#include "MaskedBitmap.h"

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

void CMaskedBitmap::Draw(CDC pDC, int x, int y)
{
    BITMAP bm;
    GetBitmap (&bm);
    CPoint size (bm.bmWidth, bm.bmHeight);
    pDC->DPtoLP (&size);

    CPoint org (0, 0);
```