第八章　计时器

Microsoft Windows 计时器是一种输入设备，它周期性地在每经过一个指定的时间间隔後就通知应用程式一次。您的程式将时间间隔告诉 Windows，例如「每 10 秒钟通知我一声」，然後 Windows 给您的程式发送周期性发生的 WM_TIMER 讯息以表示时间到了。

初看之下，Windows 计时器似乎不如键盘和滑鼠设备重要，而且对许多应用程式来说确实如此。但是，计时器比您可能认为的要重要得多，它不只用於计时程式，比如出现在工具列中的 Windows 时钟和这一章中的两个时钟程式。下面是 Windows 计时器的其他应用，有些可能并不那么明显：

多工 虽然 Windows 98 是一个优先权式的多工环境，但有时候如果程式尽快将控制传回给 Windows 效率会更高。如果一个程式必须进行大量的处理，那么它可以将作业分成小块，每接收到一个 WM_TIMER 讯息处理一块（我将在第二十章中对此做更多的讨论）。

维护更新过的状态报告 程式可以利用计时器来显示持续变化资讯的「即时」更新，比如关於系统资源的变化或某个任务的进展情况。

实作「自动储存」功能 计时器提示 Windows 程式在指定的时间过去後把使用者的工作储存到磁片上。

终止程式展示版本的执行 一些程式的展示版本被设计成在其开始後，多长时间结束，比如说，30 分钟。如果时间已到，那么计时器就会通知应用程式。

步进移动 游戏中的图形物件或电脑辅助教学程式中的连续显示，需要按指定的速率来处理。利用计时器可以消除由於微处理器速度不同而造成的不一致。

多媒体 播放 CD 声音、声音或音乐的程式通常在背景播放声音资料。一个程式可以使用计时器来周期性地检查已播放了多少声音资料，并据此协调萤幕上的视觉资讯。

另一项应用可以保证程式在退出视窗讯息处理程式後，能够重新得到控制。在大多数时情况下，程式不能够知道何时下一个讯息会到来。

计时器入门

您可以通过呼叫 SetTimer 函式为您的 Windows 程式分配一个计时器。SetTimer 有一个时间间隔范围为 1 毫秒到 4,294,967,295 毫秒（将近 50 天）的整数型态参数，这个值指示 Windows 每隔多久时间给您的程式发送 WM_TIMER 讯息。例如，如果间隔为 1000 毫秒，那么 Windows 将每秒给程式发送一个 WM_TIMER

讯息。

当您的程式用完计时器时，它呼叫 KillTimer 函式来停止计时器讯息。在处理 WM_TIMER 讯息时，您可以通过呼叫 KillTimer 函式来编写一个「限用一次」的计时器。KillTimer 呼叫清除讯息伫列中尚未被处理的 WM_TIMER 讯息，从而使程式在呼叫 KillTimer 之後就不会再接收到 WM_TIMER 讯息。

系统和计时器

Windows 计时器是 PC 硬体和 ROM BIOS 架构下之计时器一种相对简单的扩充。回到 Windows 以前的 MS-DOS 程式写作环境下，应用程式能够通过拦截者称为 timer tick 的 BIOS 中断来实作时钟或计时器。一些为 MS-DOS 编写的程式自己拦截这个硬体中断以实作时钟和计时器。这些中断每 54.915 毫秒产生一次，或者大约每秒 18.2 次。这是原始的 IBM PC 的微处理器时脉值 4.772720 MHz 被 218 所除而得出的结果。

Windows 应用程式不拦截 BIOS 中断，相反地，Windows 本身处理硬体中断，这样应用程式就不必进行处理。对於目前拥有计时器的每个程式，Windows 储存一个每次硬体 timer tick 减少的计数。当这个计数减到 0 时，Windows 在应用程式讯息伫列中放置一个 WM_TIMER 讯息，并将计数重置为其最初值。

因为 Windows 应用程式从正常的讯息伫列中取得 WM_TIMER 讯息，所以您的程式在进行其他处理时不必担心 WM_TIMER 讯息会意外中断了程式。在这方面，计时器类似於键盘和滑鼠。驱动程式处理非同步硬体中断事件，Windows 把这些事件・译为规律、结构化和顺序化的讯息。

在 Windows 98 中，计时器与其下的 PC 计时器一样具有 55 毫秒的解析度。在 Microsoft Windows NT 中，计时器的解析度为 10 毫秒。

Windows 应用程式不能以高於这些解析度的频率（在 Windows 98 下，每秒 18.2 次，在 Windows NT 下，每秒大约 100 次）接收 WM_TIMER 讯息。在 SetTimer 呼叫中指定的时间间隔总是截尾後 tick 数的整数倍。例如，1000 毫秒的间隔除以 54.925 毫秒，得到 18.207 个 tick，截尾後是 18 个 tick，它实际上是 989 毫秒。对每个小於 55 毫秒的间隔，每个 tick 都会产生一个 WM_TIMER 讯息。

计时器讯息不是非同步的

因为计时器使用硬体计时器中断，程式写作者有时会误解，认为他们的程式会非同步地被中断来处理 WM_TIMER 讯息。

然而，WM_TIMER 讯息并不是非同步的。WM_TIMER 讯息放在正常的讯息伫列之中，和其他讯息排列在一起，因此，如果在 SetTimer 呼叫中指定间隔为 1000

毫秒，那么不能保证程式每 1000 毫秒或者 989 毫秒就会收到一个 WM_TIMER 讯息。如果其他程式的执行事件超过一秒，在此期间内，您的程式将收不到任何 WM_TIMER 讯息。您可以使用本章的程式来展示这一点。事实上， Windows 对 WM_TIMER 讯息的处理非常类似於对 WM_PAINT 讯息的处理，这两个讯息都是低优先顺序的，程式只有在讯息伫列中没有其他讯息时才接收它们。

WM_TIMER 还在另一方面和 WM_PAINT 相似：Windows 不能持续向讯息伫列中放入多个 WM_TIMER 讯息，而是将多余的 WM_TIMER 讯息组合成一个讯息。因此，应用程式不会一次收到多个这样的讯息，尽管可能在短时间内得到两个 WM_TIMER 讯息。应用程式不能确定这种处理方式所导致的 WM_TIMER 讯息「遗漏」的数目。

这样，WM_TIMER 讯息仅仅在需要更新时才提示程式，程式本身不能经由统计 WM_TIMER 讯息的数目来计时（在本章後面，我们将编写两个每秒更新一次的时钟程式，并可以看到如何做到这一点）。

为了方便起见，下面在讨论时钟时，我将使用「每秒得到一次 WM_TIMER 讯息」这样的叙述，但是请记住，这些讯息并非精确的 tick 中断。

计时器的使用：三种方法

如果您需要在整个程式执行期间都使用计时器，那么您将得从 WinMain 函式中或者在处理 WM_CREATE 讯息时呼叫 SetTimer，并在退出 WinMain 或回应 WM_DESTROY 讯息时呼叫 KillTimer。根据呼叫 SetTimer 时使用的参数，可以下列三种方法之一使用计时器。

方法一

这是最方便的一种方法，它让 Windows 把 WM_TIMER 讯息发送到应用程式的正常视窗讯息处理程式中，SetTimer 呼叫如下所示：

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

第一个参数是其视窗讯息处理程式将接收 WM_TIMER 讯息的视窗代号。第二个参数是计时器 ID，它是一个非 0 数值，在整个例子中假定为 1。第三个参数是一个 32 位元无正负号整数，以毫秒为单位指定一个时间间隔，一个 60,000 的值将使 Windows 每分钟发送一次 WM_TIMER 讯息。

您可以通过呼叫

```
KillTimer (hwnd, 1) ;
```

在任何时刻停止 WM_TIMER 讯息（即使正在处理 WM_TIMER 讯息）。此函式的第二个参数是 SetTimer 呼叫中所用的同一个计时器 ID。在终止程式之前，您

应该回应 WM_DESTROY 讯息停止任何活动的计时器。

当您的视窗讯息处理程式收到一个 WM_TIMER 讯息时，wParam 参数等於计时器的 ID 值（上述情形为 1），lParam 参数为 0。如果需要设定多个计时器，那么对每个计时器都使用不同的计时器 ID。wParam 的值将随传递到视窗讯息处理程式的 WM_TIMER 讯息的不同而不同。为了使程式更具有可读性，您可以使用 #define 叙述定义不同的计时器 ID：

```
#define TIMER_SEC 1
#define TIMER_MIN 2
```

然後您可以使用两个 SetTimer 呼叫来设定两个计时器：

```
SetTimer (hwnd, TIMER_SEC, 1000, NULL) ;
SetTimer (hwnd, TIMER_MIN, 60000, NULL) ;
```

WM_TIMER 的处理如下所示：

```
case  WM_TIMER:
      switch (wParam)
      {
      case TIMER_SEC:
            //每秒一次的处理
            break ;
      case TIMER_MIN:
            //每分钟一次的处理
            break ;
      }
return 0 ;
```

如果您想将一个已经存在的计时器设定为不同的时间间隔，您可以简单地用不同的时间值再次呼叫 SetTimer。在时钟程式里，如果显示秒或不显示秒是可以选择的，您就可以这样做，只需简单地将时间间隔在 1000 毫秒和 60 000 毫秒间切换就可以了。

程式 8-1 显示了一个使用计时器的简单程式，名为 BEEPER1，计时器的时间间隔设定为 1 秒。当它收到 WM_TIMER 讯息时，它将显示区域的颜色由蓝色变为红色或由红色变为蓝色，并通过呼叫 MessageBeep 函式发出响声。（虽然 MessageBeep 通常用於 MessageBox，但它确实是一个全功能的鸣叫函式。在有音效卡的 PC 机上，一般可以使用不同的 MB_ICON 参数作为 MessageBeep 的一个参数以用於 MessageBox，来播放使用者在「控制台」的「声音」程式中选择的不同声音）。

BEEPER1 在视窗讯息处理程式处理 WM_CREATE 讯息时设定计时器。在处理 WM_TIMER 讯息处理期间，BEEPER1 呼叫 MessageBeep，•转 bFlipFlop 的值并使视窗无效以产生 WM_PAINT 讯息。在处理 WM_PAINT 讯息处理期间，BEEPER1 通过呼叫 GetClientRect 获得视窗大小的 RECT 结构，并通过呼叫 FillRect 改变视

窗的颜色。

程式 8-1 BEEPER1

```
BEEPER1.C
/*------------------------------------------------------------------------
      BEEPER1.C  -- Timer Demo Program No. 1
                                  (c) Charles Petzold, 1998
------------------------------------------------------------------------*/

#include <windows.h>

#define ID_TIMER    1

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                                  PSTR szCmdLine, int iCmdShow)
{
      static TCHAR     szAppName[] = TEXT ("Beeper1") ;
      HWND                  hwnd ;
      MSG                        msg ;
      WNDCLASS                   wndclass ;

      wndclass.style                          = CS_HREDRAW | CS_VREDRAW ;
      wndclass.lpfnWndProc              = WndProc ;
      wndclass.cbClsExtra               = 0 ;
      wndclass.cbWndExtra               = 0 ;
      wndclass.hInstance                = hInstance ;
      wndclass.hIcon                    = LoadIcon (NULL, IDI_APPLICATION) ;
      wndclass.hCursor                  = LoadCursor (NULL, IDC_ARROW) ;
      wndclass.hbrBackground            = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
      wndclass.lpszMenuName             = NULL ;
      wndclass.lpszClassName            = szAppName ;

      if (!RegisterClass (&wndclass))
      {
            MessageBox (     NULL, TEXT ("Program requires Windows NT!"),
                                    szAppName, MB_ICONERROR) ;
            return 0 ;
      }

      hwnd = CreateWindow (  szAppName, TEXT ("Beeper1 Timer Demo"),
                        WS_OVERLAPPEDWINDOW,
                  CW_USEDEFAULT, CW_USEDEFAULT,
                  CW_USEDEFAULT, CW_USEDEFAULT,
                        NULL, NULL, hInstance, NULL) ;

      ShowWindow (hwnd, iCmdShow) ;
      UpdateWindow (hwnd) ;
```

```
     while (GetMessage (&msg, NULL, 0, 0))
     {
               TranslateMessage (&msg) ;
               DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam,LPARAM lParam)
{
     static BOOL      fFlipFlop = FALSE ;
     HBRUSH                hBrush ;
     HDC                   hdc ;
     PAINTSTRUCT ps ;
     RECT       rc ;

     switch (message)
    {
     case  WM_CREATE:
           SetTimer (hwnd, ID_TIMER, 1000, NULL) ;
                 return 0 ;

     case  WM_TIMER :
                 MessageBeep (-1) ;
                 fFlipFlop = !fFlipFlop ;
                 InvalidateRect (hwnd, NULL, FALSE) ;
                 return 0 ;

     case  WM_PAINT :
                 hdc = BeginPaint (hwnd, &ps) ;

                 GetClientRect (hwnd, &rc) ;
                 hBrush  =  CreateSolidBrush  (fFlipFlop  ?  RGB(255,0,0)  :
RGB(0,0,255)) ;
                 FillRect (hdc, &rc, hBrush) ;

           EndPaint (hwnd, &ps) ;
           DeleteObject (hBrush) ;
                 return 0 ;

     case  WM_DESTROY :
                 KillTimer (hwnd, ID_TIMER) ;
                 PostQuitMessage (0) ;
           return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

因为 BEEPER1 每次收到 WM_TIMER 讯息时，都用颜色的变换显示出来，所以您可以通过呼叫 BEEPER1 来查看 WM_TIMER 讯息的性质，并完成 Windows 内部的一些其他操作。

例如，首先呼叫 控制台 的 显示器 程式，选择 效果 ，确定 拖曳时显示视窗内容 核取方块没有被选中。现在，试著移动或者缩放 BEEPER1 视窗，这将导致程式进入「模态讯息回圈」。Windows 通过在内部讯息而非您程式的讯息回圈中拦截所有讯息，来禁止对移动或者缩放操作的任何干扰。通过此回圈到达程式视窗的大多数讯息都被丢弃，这就是 BEEPER1 停止蜂鸣的原因。当完成了移动与缩放之後，您将会注意到 BEEPER1 不能取得它所丢弃的所有 WM_TIMER 讯息，尽管前两个讯息的间隔可能少於 1 秒。

在「拖曳时显示视窗内容」核取方块被选中时，Windows 中，的模态讯息回圈会试图给您的视窗讯息处理程式传递一些丢失的讯息。这样做有时工作得很好，有时却不行。

方法二

设定计时器的第一种方法是把 WM_TIMER 讯息发送到通常的视窗讯息处理程式，而第二种方法是让 Windows 直接将计时器讯息发送给您程式的另一个函式。

接收这些计时器讯息的函式被称为「callback」函式，这是一个在您的程式之中但是由 Windows 呼叫的函式。您先告诉 Windows 此函式的位址，然後 Windows 呼叫此函式。这看起来也很熟悉，因为程式的视窗讯息处理程式实际上也是一种 callback 函式。当注册视窗类别时，要将函式的位址告诉 Windows，当发送讯息给程式时，Windows 会呼叫此函式。

SetTimer 并非是唯一使用 callback 函式的 Windows 函式。CreateDialog 和 DialogBox 函式（将在第十一章中介绍）使用 callback 函式处理对话方块中的讯息；有几个 Windows 函式（EnumChildWindow、EnumFonts、EnumObjects、EnumProps 和 EnumWindow）把列举资讯传递给 callback 函式；还有几个不那么常用的函式（GrayString、LineDDA 和 SetWindowHookEx）也要求 callback 函式。

像视窗讯息处理程式一样，callback 函式也必须定义为 CALLBACK，因为它是由 Windows 从程式的程式码段呼叫的。callback 函式的参数和 callback 函式的传回值取决於 callback 函式的目的。跟计时器有关的 callback 函式中，输入参数与视窗讯息处理程式的输入参数一样。计时器 callback 函式不向 Windows 传回值。

我们把以下的 callback 函式称为 TimerProc（您能够选择与其他一些用语

不会发生冲突的任何名称），它只处理 WM_TIMER 讯息：

```
VOID CALLBACK TimerProc (    HWND hwnd, UINT message, UINT iTimerID, DWORD dwTime)
{
      处理 WM_TIMER 讯息
}
```

TimerProc 的参数 hwnd 是在呼叫 SetTimer 时指定的视窗代号。Windows 只把 WM_TIMER 讯息送给 TimerProc，因此讯息参数总是等於 WM_TIMER。iTimerID 值是计时器 ID，dwTimer 值是与从 GetTickCount 函式的传回值相容的值。这是自 Windows 启动後所经过的毫秒数。

在 BEEPER1 中已经看到过，用第一种方法设定计时器时要求下面格式的 SetTimer 呼叫：

```
SetTimer (hwnd, iTimerID, iMsecInterval, NULL) ;
```

您使用 callback 函式处理 WM_TIMER 讯息时，SetTimer 的第四个参数由 callback 函式的位址取代，如下所示：

```
SetTimer (hwnd, iTimerID, iMsecInterval, TimerProc) ;
```

我们来看看一些范例程式码，这样您就会了解这些东西是如何组合在一起的。在功能上，除了 Windows 发送一个计时器讯息给 TimerProc 而非 WndProc 之外，程式 8-2 所示的 BEEPER2 程式与 BEEPER1 是相同的。注意，TimerProc 和 WndProc 一起被宣告在程式的开始处。

程式 8-2　BEEPER2

```
BEEPER2.C
/*-------------------------------------------------------------------------
-
      BEEPER2.C --      Timer Demo Program No. 2
                              (c) Charles Petzold, 1998
-------------------------------------------------------------------------*
/

#include <windows.h>
#define ID_TIMER    1

LRESULT     CALLBACK         WndProc    (HWND, UINT, WPARAM, LPARAM) ;
VOID    CALLBACK  TimerProc (HWND, UINT, UINT,    DWORD ) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                                              PSTR szCmdLine, int iCmdShow)
{
      static char szAppName[]             = "Beeper2" ;
      HWND                          hwnd ;
      MSG                           msg ;
      WNDCLASS                wndclass ;

      wndclass.style                            = CS_HREDRAW | CS_VREDRAW ;
```

```
     wndclass.lpfnWndProc                    = WndProc ;
     wndclass.cbClsExtra                     = 0 ;
     wndclass.cbWndExtra                     = 0 ;
     wndclass.hInstance                      = hInstance ;
     wndclass.hIcon                          =        LoadIcon        (NULL,
IDI_APPLICATION) ;
     wndclass.hCursor                        = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground            =       (HBRUSH)       GetStockObject
(WHITE_BRUSH) ;
     wndclass.lpszMenuName             = NULL ;
     wndclass.lpszClassName            = szAppName ;

     if (!RegisterClass (&wndclass))
     {
               MessageBox (     NULL, TEXT ("Program requires Windows NT!"),
                                                        szAppName,
MB_ICONERROR) ;
               return 0 ;
     }

     hwnd = CreateWindow (  szAppName, "Beeper2 Timer Demo",
                                          WS_OVERLAPPEDWINDOW,
                           CW_USEDEFAULT, CW_USEDEFAULT,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
               TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     switch (message)
    {
     case  WM_CREATE:
               SetTimer (hwnd, ID_TIMER, 1000, TimerProc) ;
               return 0 ;

     case  WM_DESTROY:
               KillTimer (hwnd, ID_TIMER) ;
               PostQuitMessage (0) ;
```

```
                return 0 ;
        }
        return DefWindowProc (hwnd, message, wParam, lParam) ;
}

VOID CALLBACK TimerProc (HWND hwnd, UINT message, UINT iTimerID, DWORD dwTime)
{
        static BOOL fFlipFlop = FALSE ;
        HBRUSH                  hBrush ;
        HDC                           hdc ;
        RECT                          rc ;

        MessageBeep (-1) ;
        fFlipFlop = !fFlipFlop ;

        GetClientRect (hwnd, &rc) ;
        hdc = GetDC (hwnd) ;
        hBrush = CreateSolidBrush (fFlipFlop ? RGB(255,0,0) : RGB(0,0,255)) ;

        FillRect (hdc, &rc, hBrush) ;
        ReleaseDC (hwnd, hdc) ;
        DeleteObject (hBrush) ;
}
```

方法三

设定计时器的第三种方法类似於第二种方法，只是传递给 SetTimer 的 hwnd 参数被设定为 NULL，并且第二个参数（通常为计时器 ID）被忽略了，最後，此函式传回计时器 ID：

```
iTimerID = SetTimer (NULL, 0, wMsecInterval, TimerProc) ;
```

如果没有可用的计时器，那么从 SetTimer 传回的 iTimerID 值将为 NULL。

KillTimer 的第一个参数（通常是视窗代号）也必须为 NULL，计时器 ID 必须是 SetTimer 的传回值：

```
KillTimer (NULL, iTimerID) ;
```

传递给 TimerProc 计时器函式的 hwnd 参数也必须是 NULL。这种设定计时器的方法很少被使用。如果在您的程式在不同时刻有一系列的 SetTimer 呼叫，而又不希望追踪您已经用过了那些计时器 ID，那么使用此方法是很方便的。

既然您已经知道了如何使用 Windows 计时器，就可以开始讨论一些有用的计时器程式了。

计时器用於时钟

时钟是计时器最明显的应用，因此让我们来看看两个时钟，一个数位时钟，一个类比时钟。

建立数位时钟

程式 8-3 所示的 DIGCLOCK 程式，使用类似 LED 的 7 个显示方块显示了目前的时间。

程式 8-3 DIGCLOCK

```
DIGCLOCK.C
/*------------------------------------------------------------------------
--
      DIGCLOCK.C --          Digital Clock
                                   (c) Charles Petzold, 1998
------------------------------------------------------------------------
*/

#include <windows.h>
#define ID_TIMER    1
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                                              PSTR szCmdLine, int iCmdShow)
{
      static TCHAR     szAppName[] = TEXT ("DigClock") ;
      HWND                                hwnd ;
      MSG                           msg ;
      WNDCLASS                      wndclass ;

      wndclass.style                            = CS_HREDRAW | CS_VREDRAW ;
      wndclass.lpfnWndProc                      = WndProc ;
      wndclass.cbClsExtra                       = 0 ;
      wndclass.cbWndExtra                       = 0 ;
      wndclass.hInstance                        = hInstance ;
      wndclass.hIcon                            =        LoadIcon        (NULL,
IDI_APPLICATION) ;
      wndclass.hCursor                          = LoadCursor (NULL, IDC_ARROW) ;
      wndclass.hbrBackground              =       (HBRUSH)       GetStockObject
(WHITE_BRUSH) ;
      wndclass.lpszMenuName               = NULL ;
      wndclass.lpszClassName              = szAppName ;

      if (!RegisterClass (&wndclass))
      {
                  MessageBox (     NULL, TEXT ("Program requires Windows NT!"),
                                                          szAppName,
```

```
MB_ICONERROR) ;
                return 0 ;
     }

     hwnd = CreateWindow (  szAppName, TEXT ("Digital Clock"),
                WS_OVERLAPPEDWINDOW,
                CW_USEDEFAULT, CW_USEDEFAULT,
                CW_USEDEFAULT, CW_USEDEFAULT,
                NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
                {
                     TranslateMessage (&msg) ;
                     DispatchMessage (&msg) ;
                }
     return msg.wParam ;
     }

void DisplayDigit (HDC hdc, int iNumber)
{
     static BOOL  fSevenSegment [10][7] = {
                    1, 1,   1,    0,    1,    1,    1,               // 0
                    0, 0,   1,    0,    0,    1,    0,               // 1
                    1, 0,   1,    1,    1,    0,    1,               // 2
                    1, 0,   1,    1,    0,    1,    1,               // 3
                    0, 1,   1,    1,    0,    1,    0,               // 4
                    1, 1,   0,    1,    0,    1,    1,               // 5
                    1, 1,   0,    1,    1,    1,    1,               // 6
                    1, 0,   1,    0,    0,    1,    0,               // 7
                    1, 1,   1,    1,    1,    1,    1,               // 8
                    1, 1,   1,    1,    0,    1,    1 } ;       // 9
     static POINT ptSegment [7][6] = {
                 7,  6,  11, 2,  31,  2, 35,  6,  31, 10, 11, 10,
                6,  7,  10, 11, 10, 31, 6,   35, 2,  31, 2,  11,
                36, 7,  40, 11, 40, 31, 36,  35, 32, 31, 32, 11,
                7 , 36, 11, 32, 31, 32, 35,  36, 31, 40, 11, 40,
                6 , 37, 10, 41, 10, 61, 6,   65, 2,  61, 2,  41,
                36, 37, 40, 41, 40, 61, 36,  65, 32, 61, 32, 41,
                7 , 66, 11, 62, 31, 62, 35,  66, 31, 70, 11, 70 } ;
     int         iSeg ;
     for (iSeg = 0 ; iSeg < 7 ; iSeg++)
                if (fSevenSegment [iNumber][iSeg])
                                Polygon (hdc, ptSegment [iSeg], 6) ;
}
```

```
void DisplayTwoDigits (HDC hdc, int iNumber, BOOL fSuppress)
{
      if (!fSuppress || (iNumber / 10 != 0))
                  DisplayDigit (hdc, iNumber / 10) ;
      OffsetWindowOrgEx (hdc, -42, 0, NULL) ;
      DisplayDigit (hdc, iNumber % 10) ;
      OffsetWindowOrgEx (hdc, -42, 0, NULL) ;
}

void DisplayColon (HDC hdc)
{
      POINT ptColon [2][4] = {     2,    21,   6,    17,   10,   21,   6,
                  25, 2,      51,   6,    47,   10,   51,   6,    55 } ;

      Polygon (hdc, ptColon [0], 4) ;
      Polygon (hdc, ptColon [1], 4) ;

      OffsetWindowOrgEx (hdc, -12, 0, NULL) ;
}

void DisplayTime (HDC hdc, BOOL f24Hour, BOOL fSuppress)
{
      SYSTEMTIME st ;
      GetLocalTime (&st) ;
      if (f24Hour)
            DisplayTwoDigits (hdc, st.wHour, fSuppress) ;
      else
      DisplayTwoDigits (hdc, (st.wHour %= 12) ? st.wHour : 12, fSuppress) ;
      DisplayColon (hdc) ;
      DisplayTwoDigits (hdc, st.wMinute, FALSE) ;
      DisplayColon (hdc) ;
      DisplayTwoDigits (hdc, st.wSecond, FALSE) ;
}
LRESULT CALLBACK WndProc (   HWND  hwnd,  UINT  message,  WPARAM  wParam,LPARAM
lParam)
{
      static BOOL            f24Hour, fSuppress ;
      static HBRUSH          hBrushRed ;
      static int                   cxClient, cyClient ;
      HDC                                 hdc ;
      PAINTSTRUCT   ps ;
      TCHAR                        szBuffer [2] ;

      switch (message)
      {
      case  WM_CREATE:
                  hBrushRed = CreateSolidBrush (RGB (255, 0, 0)) ;
                  SetTimer (hwnd, ID_TIMER, 1000, NULL) ;// fall through
```

```
     case  WM_SETTINGCHANGE:
           GetLocaleInfo (LOCALE_USER_DEFAULT, LOCALE_ITIME, szBuffer, 2) ;
           f24Hour = (szBuffer[0] == '1') ;

           GetLocaleInfo (LOCALE_USER_DEFAULT, LOCALE_ITLZERO, szBuffer, 2) ;
           fSuppress = (szBuffer[0] == '0') ;

           InvalidateRect (hwnd, NULL, TRUE) ;
           return 0 ;

     case  WM_SIZE:
           cxClient = LOWORD (lParam) ;
                 cyClient = HIWORD (lParam) ;
                 return 0 ;

     case  WM_TIMER:
                 InvalidateRect (hwnd, NULL, TRUE) ;
                 return 0 ;

     case  WM_PAINT:
                 hdc = BeginPaint (hwnd, &ps) ;

                 SetMapMode (hdc, MM_ISOTROPIC) ;
                 SetWindowExtEx (hdc, 276, 72, NULL) ;
                 SetViewportExtEx (hdc, cxClient, cyClient, NULL) ;

                 SetWindowOrgEx (hdc, 138, 36, NULL) ;
                 SetViewportOrgEx (hdc, cxClient / 2, cyClient / 2, NULL) ;
                 SelectObject (hdc, GetStockObject (NULL_PEN)) ;
                 SelectObject (hdc, hBrushRed) ;

                 DisplayTime (hdc, f24Hour, fSuppress) ;

                 EndPaint (hwnd, &ps) ;
                 return 0 ;

     case  WM_DESTROY:
                 KillTimer (hwnd, ID_TIMER) ;
                 DeleteObject (hBrushRed) ;
                 PostQuitMessage (0) ;
                 return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

DIGCLOCK 视窗如图 8-1 所示。

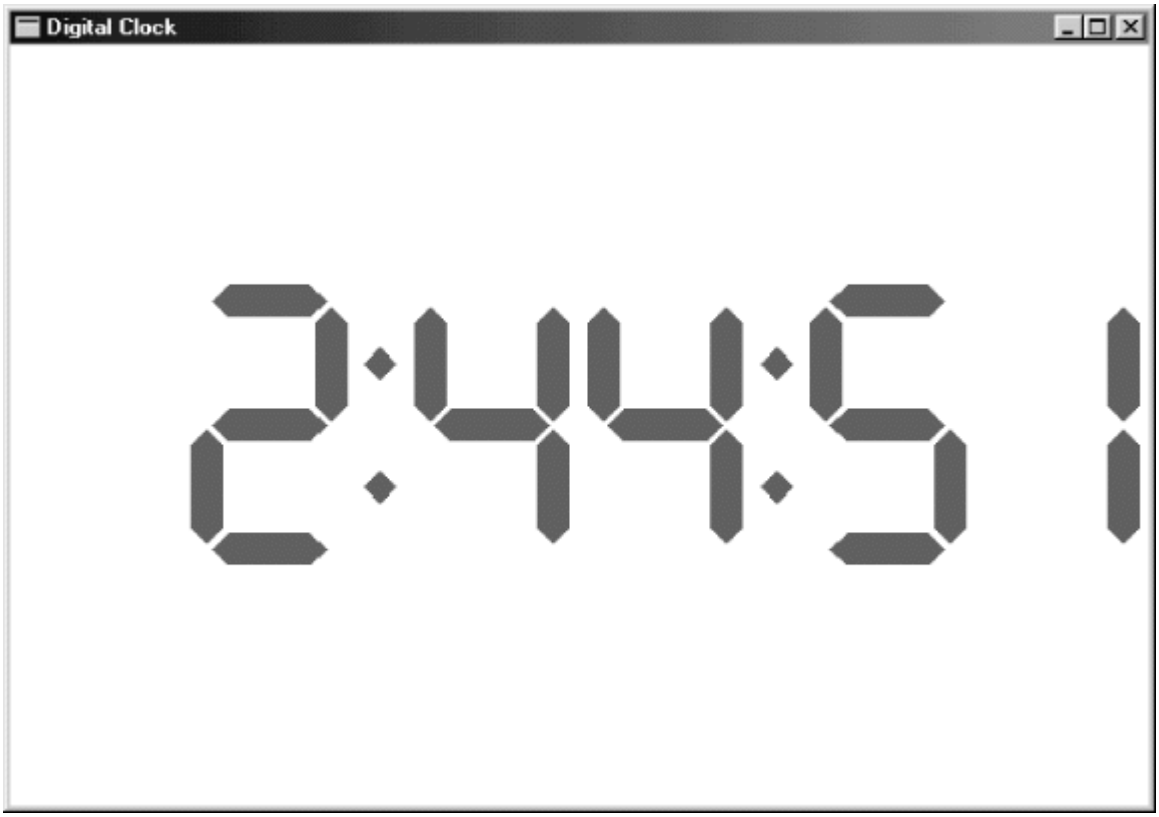

图 8-1 DIGCLOCK 的萤幕显示

虽然，在图 8-1 中您看不到时钟的数字是红色的。DIGCLOCK 的视窗讯息处理程式在处理 WM_CREATE 讯息处理期间建立了一个红色的画刷并在处理 WM_DESTROY 讯息处理期间清除它。WM_CREATE 讯息也为 DIGCLOCK 设定了一个一秒的计时器，该计时器在处理 WM_DESTROY 讯息处理期间被终止（待会将讨论对 GetLocaleInfo 的呼叫）。

在收到 WM_TIMER 讯息後，DIGCLOCK 的视窗程序呼叫 InvalidateRect 简单地使整个视窗无效。这不是最佳方法，因为每秒整个视窗都要被擦除和重画，有时会引起显示器的闪烁。依据目前的时间使视窗需要更新的部分无效是最好的解决方法。然而，在逻辑上这样做的确很复杂。

在处理 WM_TIMER 讯息处理期间使视窗无效会迫使所有程式的真正活动转入 WM_PAINT。DIGCLOCK 在 WM_PAINT 讯息一开始将映射方式设定为 MM_ISOTROPIC。这样，DIGCLOCK 将使用水平方向和垂直方向相等的轴。这些轴（由 SetWindowExtEx 呼叫设定）是水平 276 个单位，垂直 72 个单位。当然，这些轴定得有点太随意了，但它们是按照时钟数位元的大小和间距安排的。

DIGCLOCK 将视窗原点设定为（138,36），这是视窗范围的中心；将视埠原点设定为（cxClient / 2,cyClient / 2）。这意味著时钟的显示位於 DIGCLOCK 显示区域的中心，但是该 DIGCLOCK 也可以使用在显示幕左上角的原点（0, 0）的轴。

然後 WM_PAINT 将目前画刷设定为之前建立的红画刷，将目前画笔设定为

NULL_PEN， 并呼叫 DIGCLOCK 中的函式 DisplayTime。

取得目前时间

DisplayTime 函式开始呼叫 Windows 函式 GetLocalTime，它带有一个的 SYSTEMTIME 结构的参数，在 WINBASE.H 中定义为：

```
typedef struct _SYSTEMTIME
{
      WORD  wYear ;
      WORD  wMonth ;
      WORD  wDayOfWeek ;
      WORD  wDay ;
      WORD  wHour ;
      WORD  wMinute ;
      WORD  wSecond ;
      WORD  wMilliseconds ;
}
SYSTEMTIME, * PSYSTEMTIME ;
```

很明显，SYSTEMTIME 结构包含日期和时间。月份由 1 开始递增（也就是说，一月是 1），星期由 0 开始递增（星期天是 0）。wDay 成员是本月目前的日子，也是由 1 开始递增的。

SYSTEMTIME 主要用於 GetLocalTime 和 GetSystemTime 函式。GetSystemTime 函式传回目前的世界时间（Coordinated Universal Time，UTC），大概与英国格林威治时间相同。GetLocalTime 函式传回当地时间，依据电脑所在的时区。这些值的精确度完全决定於使用者所调整的时间精确度以及是否指定了正确的时区。可以双击工作列的时间显示来检查电脑上的时区设定。第二十三章会有一个程式，能够通过 Internet 精确地设定时间。

Windows 还有 SetLocalTime 和 SetSystemTime 函式，以及在/Platform SDK/Windows Base Services/General Library/Time 中说明的其他与时间有关的函式。

显示数字和冒号

如果 DIGCLOCK 使用一种模拟 7 段显示的字体将会简单一些。否则，它就得使用 Polygon 函式做所有的工作。

DIGCLOCK 中的 DisplayDigit 函式定义了两个阵列。fSevenSegment 阵列有 7 个 BOOL 值，用於从 0 到 9 的每个十进位数字。这些值指出了哪一段需要显示（为 1），哪一段不需要显示（为 0）。在这个阵列中，7 段由上到下、由左到右排序。7 段中的每个段都是一个 6 边的多边形。ptSegment 阵列是一个 POINT

结构的阵列，指出了 7 个段中每个点的图形座标。每个数字由下列程式码画出：

```
for   (iSeg = 0 ; iSeg < 7 ; iSeg++)
      if (  fSevenSegment [iNumber][iSeg])
                  Polygon (hdc, ptSegment [iSeg], 6) ;
```

类似地（但更简单），DisplayColon 函式在小时与分钟、分钟与秒之间画一个冒号。数字是 42 个单位宽，冒号是 12 个单位宽，因此 6 个数字与 2 个冒号，总宽度是 276 个单位，SetWindowExtEx 呼叫中使用了这个大小。

回到 DisplayTime 函式，原点位於最左数字位置的左上角。DisplayTime 呼叫 DisplayTwoDigits，DisplayTwoDigits 呼叫 DisplayDigit 两次，并且在每次呼叫 OffsetWindowOrgEx 後，将视窗原点向右移动 42 个单位。类似地，DisplayColon 函式在画完冒号後，将视窗原点向右移动 12 个单位。用这种方法，不管物件出现在视窗内的哪个地方，函式对数字和冒号都使用同样的座标。

这个程式的其他技巧是以 12 小时或 24 小时的格式显示时间以及当最左边的小时数字为 0 时不显示它。

国际化

尽管像 DIGCLOCK 这样显示时间是非常简单的，但是要显示复杂的日期和时间还是要依赖 Windows 的国际化支援。格式化日期和时间的最简单的方法是呼叫 GetDateFormat 和 GetTimeFormat 函式。这些函式在/Platform SDK/Windows Base Services/General Library/String Manipulation/String Manipulation Reference/String Manipulation Functions 中有记载，但是它们在/Platform SDK/Windows Base Services/International Features/National Language Support 中进行了说明。这些函式接受 SYSTEMTIME 结构并且依据使用者在「控制台」的「区域设定」 程式中所做的选择而将日期和时间格式化。

DIGCLOCK 不能使用 GetDateFormat 函式，因为它只知道显示数字和冒号，然而，DIGCLOCK 应该能够根据使用者的参数选择来显示 12 小时或 24 小时的格式，并禁止（或不禁止）开头的小时数字。您可以从 GetLocaleInfo 函式中取得这种资讯。虽然 GetLocaleInfo 在 /Platform SDK/Windows Base Services/General Library/String Manipulation/String Manipulation Reference/String Manipulation Functions 中有记载，但是这个函式使用的识别字在 /Platform SDK/Windows Base Services/International Features/National Language Support/National Language Support Constants 中有说明。

DIGCLOCK 在处理 WM_CREATE 讯息时，最初呼叫 GetLocaleInfo 两次，第一次使用 LOCALE_ITIME 识别字（确定使用的是 12 小时还是 24 小时格式），然後

使用 LOCALE_ITLZERO 识别字（在小时显示中禁止前面显示 0）。GetLocaleInfo 函式在字串中传回所有的资讯，但是在大多数情况下把字串转变为整数并不是非常容易。DIGCLOCK 把字串储存在两个静态变数中并把它们传递给 DisplayTime 函式。

如果使用者更改了任何系统设定，则会将 WM_SETTINGCHANGE 讯息传送给所有的应用程式。DIGCLOCK 通过再次呼叫 GetLocaleInfo 处理这个讯息。以这种方式，您可以在「控制台」的「区域设定」 程式中进行不同的设定来实验一下。

在理论上,DIGCLOCK 也应该使用 LOCALE_STIME 识别字呼叫 GetLocaleInfo。这会传回使用者为时间的小时、分钟和秒等单个部分选择的字元。因为 DIGCLOCK 被设定为仅显示冒号，所以不管选择了什么，都会得到冒号。要指出时间是 A.M. 或 P.M.，应用程式可以使用带有 LOCALE_S1159 和 LOCALE_S2359 识别字的 GetLocaleInfo 函式。这些识别字使程式获得适合於使用者国家/地区和语言的字串。

我们也可以让 DIGCLOCK 处理 WM_TIMECHANGE 讯息，这样它将系统时间与日期发生变化的讯息通知应用程式。DIGCLOCK 因 WM_TIMER 讯息而每秒更新一次，实际上没有必要这样作，对 WM_TIMECHANGE 讯息的处理使得每分钟更新一次的时钟变得更为合理。

建立类比时钟

类比时钟不必关心国际化问题，但是由於图形所引起的复杂性却抵消了这种简化。为了正确地产生时钟，您需要知道一些三角函数。CLOCK 如程式 8-4 所示。

程式 8-4　CLOCK

```
CLOCK.C
/*------------------------------------------------------------------------
-
   CLOCK.C --     Analog Clock Program
                              (c) Charles Petzold, 1998
------------------------------------------------------------------------*
/

#include <windows.h>
#include <math.h>

#define ID_TIMER                    1
#define TWOPI                             (2 * 3.14159)

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (   HINSTANCE hInstance, HINSTANCE hPrevInstance,
```

```
                                                    PSTR    szCmdLine,    int
iCmdShow)
{
      static TCHAR     szAppName[] = TEXT ("Clock") ;
      HWND                              hwnd;
      MSG                         msg;
      WNDCLASS                    wndclass ;

      wndclass.style                          = CS_HREDRAW | CS_VREDRAW ;
      wndclass.lpfnWndProc              = WndProc ;
      wndclass.cbClsExtra               = 0 ;
      wndclass.cbWndExtra               = 0 ;
      wndclass.hInstance                = hInstance ;
      wndclass.hIcon                    = NULL ;
      wndclass.hCursor                  = LoadCursor (NULL, IDC_ARROW) ;
      wndclass.hbrBackground            =       (HBRUSH)       GetStockObject
(WHITE_BRUSH) ;
      wndclass.lpszMenuName             = NULL ;
      wndclass.lpszClassName            = szAppName ;

      if (!RegisterClass (&wndclass))
      {
                MessageBox (     NULL, TEXT ("Program requires Windows NT!"),
                                                          szAppName,
MB_ICONERROR) ;
                return 0 ;
      }
      hwnd = CreateWindow (  szAppName, TEXT ("Analog Clock"),
                     WS_OVERLAPPEDWINDOW,
                     CW_USEDEFAULT, CW_USEDEFAULT,
                     CW_USEDEFAULT, CW_USEDEFAULT,
                     NULL, NULL, hInstance, NULL) ;

      ShowWindow (hwnd, iCmdShow) ;
      UpdateWindow (hwnd) ;

      while (GetMessage (&msg, NULL, 0, 0))
      {
                       TranslateMessage (&msg) ;
                       DispatchMessage (&msg) ;
     }
      return msg.wParam ;
}

void SetIsotropic (HDC hdc, int cxClient, int cyClient)
{
      SetMapMode (hdc, MM_ISOTROPIC) ;
      SetWindowExtEx (hdc, 1000, 1000, NULL) ;
```

```
      SetViewportExtEx (hdc, cxClient / 2, -cyClient / 2, NULL) ;
      SetViewportOrgEx (hdc, cxClient / 2,  cyClient / 2, NULL) ;
}

void RotatePoint (POINT pt[], int iNum, int iAngle)
{
      int   i ;
      POINT ptTemp ;

      for (i = 0 ; i < iNum ; i++)
      {
                ptTemp.x = (int) (pt[i].x * cos (TWOPI * iAngle / 360) +
                                  pt[i].y * sin (TWOPI * iAngle / 360)) ;

                ptTemp.y = (int) (pt[i].y * cos (TWOPI * iAngle / 360) -
                                  pt[i].x * sin (TWOPI * iAngle / 360)) ;

                pt[i] = ptTemp ;
     }
}

void DrawClock (HDC hdc)
{
      int   iAngle ;
      POINT pt[3] ;
      for (iAngle = 0 ; iAngle < 360 ; iAngle += 6)
     {
                pt[0].x    =   0 ;
                pt[0].y    = 900 ;

                RotatePoint (pt, 1, iAngle) ;

                pt[2].x    = pt[2].y = iAngle % 5 ? 33 : 100 ;

                pt[0].x -  = pt[2].x / 2 ;
                pt[0].y -  = pt[2].y / 2 ;

                pt[1].x    = pt[0].x + pt[2].x ;
                pt[1].y    = pt[0].y + pt[2].y ;

                SelectObject (hdc, GetStockObject (BLACK_BRUSH)) ;

                Ellipse (hdc, pt[0].x, pt[0].y, pt[1].x, pt[1].y) ;
      }
}

void DrawHands (HDC hdc, SYSTEMTIME * pst, BOOL fChange)
{
```

```
    static POINT pt[3][5] ={0, -150, 100, 0, 0, 600, -100, 0, 0, -150,
                            0, -200,      50,   0, 0, 800, -50, 0, 0,  -200,
                            0,0,    0,    0, 0,  0,  0,  0, 0, 800 } ;
     int                    i, iAngle[3] ;
     POINT                        ptTemp[3][5] ;

     iAngle[0]  = (pst->wHour * 30) % 360 + pst->wMinute / 2 ;
     iAngle[1]  =  pst->wMinute  *  6 ;
     iAngle[2]  =  pst->wSecond  *  6 ;

     memcpy (ptTemp, pt, sizeof (pt)) ;
     for (i = fChange ? 0 : 2 ; i < 3 ; i++)
     {
               RotatePoint (ptTemp[i], 5, iAngle[i]) ;
               Polyline (hdc, ptTemp[i], 5) ;
     }
}

LRESULT CALLBACK WndProc (  HWND  hwnd,  UINT  message,  WPARAM  wParam,LPARAM
lParam)
{
     static int       cxClient, cyClient ;
     static SYSTEMTIME stPrevious ;
     BOOL                              fChange ;
     HDC                               hdc ;
     PAINTSTRUCT                       ps ;
     SYSTEMTIME                        st ;

     switch (message)
     {
     case  WM_CREATE :
               SetTimer (hwnd, ID_TIMER, 1000, NULL) ;
               GetLocalTime (&st) ;
               stPrevious = st ;
               return 0 ;

     case  WM_SIZE :
               cxClient = LOWORD (lParam) ;
               cyClient = HIWORD (lParam) ;
               return 0 ;

     case  WM_TIMER :
               GetLocalTime (&st) ;

               fChange =   st.wHour      !   = stPrevious.wHour ||
                                             st.wMinute !          =
stPrevious.wMinute ;
```

```
                hdc = GetDC (hwnd) ;

                SetIsotropic (hdc, cxClient, cyClient) ;

                SelectObject (hdc, GetStockObject (WHITE_PEN)) ;
                DrawHands (hdc, &stPrevious, fChange) ;

                SelectObject (hdc, GetStockObject (BLACK_PEN)) ;
                DrawHands (hdc, &st, TRUE) ;

                ReleaseDC (hwnd, hdc) ;

                stPrevious = st ;
                return 0 ;

     case  WM_PAINT :
                hdc = BeginPaint (hwnd, &ps) ;

                SetIsotropic (hdc, cxClient, cyClient) ;
                DrawClock    (hdc) ;
                DrawHands    (hdc, &stPrevious, TRUE) ;

                EndPaint (hwnd, &ps) ;
                return 0 ;

     case  WM_DESTROY :
                KillTimer (hwnd, ID_TIMER) ;
                PostQuitMessage (0) ;
                return 0 ;
    }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

CLOCK 萤幕显示如图 8-2。

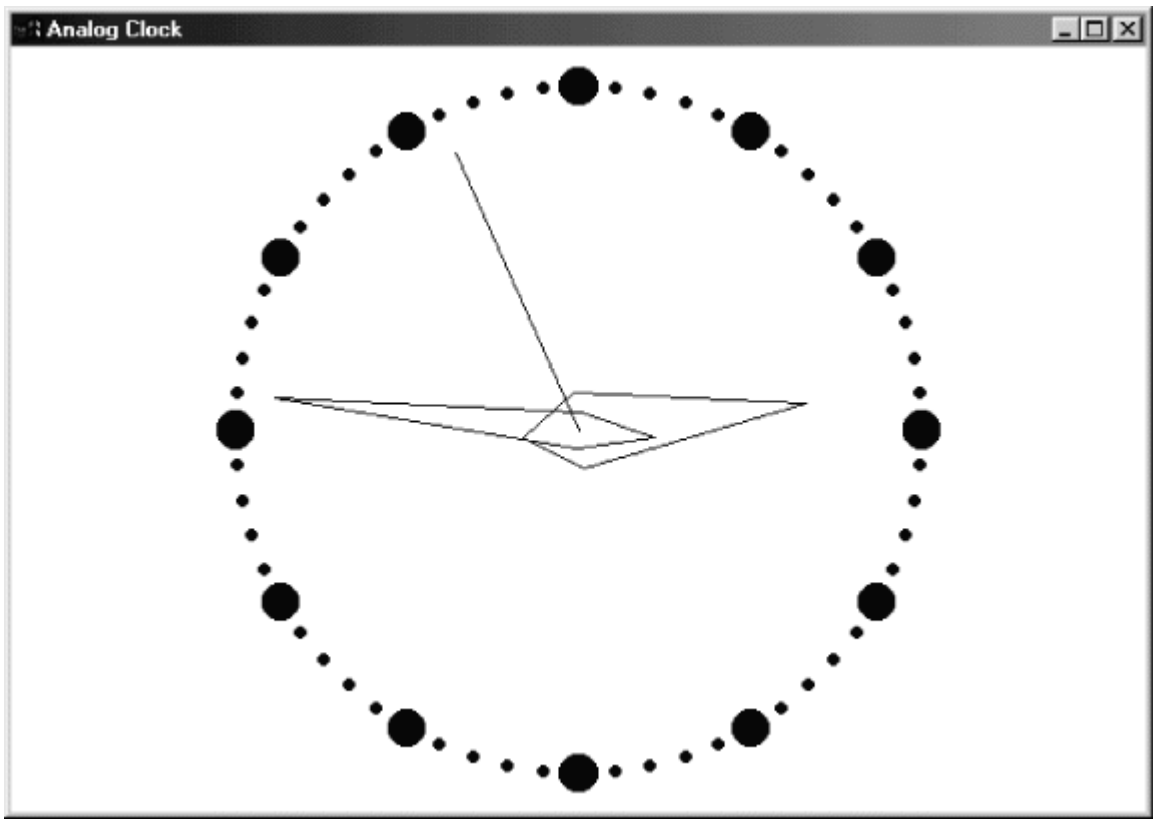

图 8-2 CLOCK 的萤幕显示

等方向性(isotropic)映射对於这样的应用来说是理想的，CLOCK.C 中的 SetIsotropic 函式负责设定此模式。在呼叫 SetMapMode 之後，SetIsotropic 将视窗范围设定为 1000，并将视埠范围设定为显示区域的一半宽度和显示区域的负的一半高度。视埠原点被设定为显示区域的中心。我在第五章中讨论过，这将建立一个笛卡儿座标系，其点(0,0)位於显示区域的中心，在所有方向上的范围都是 1000。

RotatePoint 函式是用到三角函数的地方，此函式的三个参数分别是一个或者多个点的阵列、阵列中点的个数以及以度为单位的旋转角度。函式以原点为中心按顺时针方向（这对一个时钟正合适）旋转这些点。例如，如果传给函式的点是(0,100)——即 12:00 的位置——而角度为 90 度，那么该点将被变换为(100,0)——即 3:00。它使用下列公式来做到这一点：

```
x' = x * cos (a) + y * sin (a)
y' = y * cos (a) - x * sin (a)
```

RotatePoint 函式在绘制时钟表面的点和表针时都是有用的，我们将马上看到这一点。

DrawClock 函式绘制 60 个时钟表面的点，从顶部(12:00)开始，其中每个点离原点 900 单位，因此第一个点位於(0,900)，此後的每个点按顺时针依次增加 6 度。这些点中的 12 个直径为 100 个单位；其余的为 33 个单位。使用 Ellipse 函式来画点。

DrawHands 函式绘制时钟的时针、分针和秒针。定义表针轮廓（当它们垂直向上时的形状）的座标存放在一个 POINT 结构的阵列中。根据时间，这些座标使用 RotatePoint 函式进行旋转，并用 Windows 的 Polyline 函式进行显示。注意时针和分针只有当传递给 DrawHands 的 bChange 参数为 TRUE 时才被显示。当程式更新时钟的表针时，大多数情况下时针和分针不需要重画。

现在让我们将注意力转到视窗讯息处理程式。在 WM_CREATE 讯息处理期间，视窗讯息处理程式取得目前时间并将它存放在名为 dtPrevious 的变数中，这个变数将在以後被用於确定时针或者分针从上次更新以来是否改变过。

第一次绘制时钟是在第一个 WM_PAINT 讯息处理期间，这只不过是依次呼叫 SetIsotropic、 DrawClock 和 DrawHands，後者的 bChange 参数被设定为 TRUE。

在 WM_TIMER 讯息处理期间，WndProc 首先取得新的时间并确定是否需要重新绘制时针和分针。如果需要，则使用一个白色画笔和上一次时间绘制所有的表针，从而有效地擦除它们。否则，只对秒针使用白色画笔进行擦除，然後，再使用一个黑色画笔绘制所有的表针。

以计时器进行状态报告

本章的最後一个程式是我在第五章提到过的。它是一个使用 GetPixel 函式的好例子。

WHATCLR （见程式 8-5）显示了滑鼠游标下目前图素的 RGB 颜色。

程式 8-5 WHATCLR

```
WHATCLR.C
/*-----------------------------------------------------------------------------
      WHATCLR.C -- Displays Color Under Cursor
                           (c) Charles Petzold, 1998
-----------------------------------------------------------------------------*
/

#include <windows.h>
#define ID_TIMER    1
void FindWindowSize (int *, int *) ;
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                                              PSTR szCmdLine, int iCmdShow)
{
      static TCHAR     szAppName[] = TEXT ("WhatClr") ;
      HWND                                hwnd ;
      int                           cxWindow, cyWindow ;
      MSG                           msg ;
      WNDCLASS                      wndclass ;
```

```
     wndclass.style                    = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc              = WndProc ;
     wndclass.cbClsExtra               = 0 ;
     wndclass.cbWndExtra               = 0 ;
     wndclass.hInstance                = hInstance ;
     wndclass.hIcon                    = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor                  = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground          = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName          = szAppName ;

     if (!RegisterClass (&wndclass))
     {
               MessageBox (     NULL, TEXT ("This program requires Windows
NT!"),
                                                        szAppName,
MB_ICONERROR) ;
               return 0 ;
     }

     FindWindowSize (&cxWindow, &cyWindow) ;
     hwnd = CreateWindow (szAppName, TEXT ("What Color"),
            WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_BORDER,
            CW_USEDEFAULT, CW_USEDEFAULT,
            cxWindow, cyWindow,
            NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
                    TranslateMessage (&msg) ;
                    DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

void FindWindowSize (int * pcxWindow, int * pcyWindow)
{
     HDC                   hdcScreen ;
     TEXTMETRIC  tm ;

     hdcScreen = CreateIC (TEXT ("DISPLAY"), NULL, NULL, NULL) ;
     GetTextMetrics (hdcScreen, &tm) ;
     DeleteDC (hdcScreen) ;
```

```
     * pcxWindow = 2 *      GetSystemMetrics (SM_CXBORDER)  +
                            12 * tm.tmAveCharWidth ;
     * pcyWindow = 2 *      GetSystemMetrics (SM_CYBORDER)  +
                         GetSystemMetrics (SM_CYCAPTION) +
                            2 * tm.tmHeight ;
}

LRESULT CALLBACK WndProc (   HWND  hwnd,  UINT  message,  WPARAM  wParam,LPARAM
lParam)
{
     static COLORREF        cr, crLast ;
     static HDC                   hdcScreen ;
     HDC                    hdc ;
     PAINTSTRUCT            ps ;
     POINT                  pt ;
     RECT                   rc ;
     TCHAR                  szBuffer [16] ;

     switch (message)
     {
     case  WM_CREATE:
               hdcScreen = CreateDC (TEXT ("DISPLAY"), NULL, NULL, NULL) ;
               SetTimer (hwnd, ID_TIMER, 100, NULL) ;
               return 0 ;

     case  WM_TIMER:
               GetCursorPos (&pt) ;
               cr = GetPixel (hdcScreen, pt.x, pt.y) ;

               SetPixel (hdcScreen, pt.x, pt.y, 0) ;

               if (cr != crLast)
               {
                         crLast = cr ;
                         InvalidateRect (hwnd, NULL, FALSE) ;
               }
               return 0 ;

     case  WM_PAINT:
               hdc = BeginPaint (hwnd, &ps) ;

               GetClientRect (hwnd, &rc) ;

               wsprintf (szBuffer, TEXT ("  %02X %02X %02X  "),
                     GetRValue (cr), GetGValue (cr), GetBValue (cr)) ;

               DrawText (hdc, szBuffer, -1, &rc,
                  DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;
```

```
                EndPaint (hwnd, &ps) ;
                return 0 ;

     case  WM_DESTROY:
                DeleteDC (hdcScreen) ;
                KillTimer (hwnd, ID_TIMER) ;
                PostQuitMessage (0) ;
                return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

WHATCLR 在 WinMain 中做了一点与以往不同的事。因为 WHATCLR 的视窗只需要显示十六进位 RGB 值那么大，所以它在 CreateWindow 函式中使用 WS_BORDER 视窗样式建立了一个不能改变大小的视窗。要计算视窗的大小，WHATCLR 通过先呼叫 CreateIC 再呼叫 GetSystemMetrics 以取得用於视讯显示的装置内容资讯。计算好的视窗宽度和高度值被传递给 CreateWindow。

WHATCLR 的视窗讯息处理程式在处理 WM_CREATE 讯息处理期间，呼叫 CreateDC 建立了用於整个视讯显示的装置内容。这个装置内容在程式的生命周期内都有效。在处理 WM_TIMER 讯息处理期间，程式取得目前滑鼠游标位置的图素。在处理 WM_PAINT 讯息处理期间显示 RGB 颜色。

您可能想知道，从 CreateDC 函式中取得的装置内容代号是否能让您在萤幕的任意位置显示一些东西，而不光只是取得图素颜色。答案是可以的，一般而言，让一个应用程式在另一个程式控制的画面区域上画图是不好的，但在某些特殊情况下，这可能会非常有用。