第四章　输出文字

在前一章，您看到了一个简单的 Windows 98 程式，它在视窗中央，或者更准确地说，在显示区域中央显示一行文字。正如我们学到的，显示区域是整个应用程式视窗中未被标题列、视窗边框，以及可选的功能表列、工具列、状态列和卷动列占据的部分。简而言之，显示区域是视窗中可以由程式任意书写和传递视觉资讯的部分。

对於程式的显示区域，您几乎可以为所欲为，只不过您不能假定视窗大小是某一特定尺寸，或者在程式执行时其大小会保持不变。如果您不熟悉图形视窗环境的程式设计，这些限制可能会使您感到惊讶：不能再假设萤幕上的一行文字一定有 80 个字元了。您的程式必须与其他 Windows 程式共用视讯显示器。Windows 使用者控制程式视窗在萤幕上显示的方式。尽管可以建立固定大小的视窗（这对於计算器之类的应用是合理的），但在大多数情况下，使用者应该能够改变应用程式视窗的大小。您的程式必须能够接受指定给它的大小，并且合理地利用这一空间。

这有两种可能的情况。一种可能是，程式只有仅能显示「hello」的显示区域；还有另一种可能，即程式在一个大萤幕、高解析度的系统上执行，其显示区域大得足以显示两整页文字。灵活地处理这两种极端是 Windows 程式设计的要点之一。

这一章，我们将讲述程式在显示区域显示资讯的方式，但比上一章说明的显示方式更加复杂。当程式在显示区域显示文字或图形时，它经常要「绘制」它的显示区域。本章著重讲述绘制的方法。

尽管 Windows 为显示图形提供了强大的图形装置介面（GDI）函式，但在这一章中，我只介绍简单文字行的显示。我也将忽略 Windows 能够使用的不同字体外形及字体大小，仅使用 Windows 的内定系统字体。这看起来似乎是一种限制，其实不然，本章涉及和解决的问题适用於所有 Windows 程式设计。在混合显示文字和图形时，Windows 内定字体的字元大小通常决定了图形的尺寸。

本章表面上是讨论绘图的方法，实际上是讨论与装置无关的程式设计基础。Windows 程式只能对显示区域大小甚至字元的大小做很少的假定，相反地，必须使用 Windows 提供的功能来取得关於程式执行环境的资讯。

绘制和更新

在文字模式环境下，程式可以在显示器的任意部分输出，程式输出到萤幕

上的内容会停留在原处，不会神秘地消失。因此，程式可以丢掉重新生成萤幕显示时所需的资讯。

在 Windows 中，只能在视窗的显示区域绘制文字和图形，而且不能确保在显示区域内显示的内容会一直保留到程式下一次有意地改写它时还保留在那里。例如，使用者可能会在萤幕上移动另一个程式的视窗，这样就可能覆盖您的应用程式视窗的一部分。Windows 不会保存您的视窗中被其他程式覆盖的区域，当程式移开後，Windows 会要求您的程式更新显示区域的这个部分。

Windows 是一个讯息驱动系统。它通过把讯息投入应用程式讯息伫列中或者把讯息发送给合适的视窗讯息处理程式，将发生的各种事件通知给应用程式。Windows 通过发送 WM_PAINT 讯息通知视窗讯息处理程式，视窗的部分显示区域需要绘制。

WM_PAINT 讯息

大多数 Windows 程式在 WinMain 中进入讯息回圈之前的初始化期间都要呼叫函式 UpdateWindow。Windows 利用这个机会给视窗讯息处理程式发送第一个 WM_PAINT 讯息。这个讯息通知视窗讯息处理程式：必须绘制显示区域。此後，视窗讯息处理程式应在任何时刻都准备好处理其他 WM_PAINT 讯息，必要的话，甚至重新绘制视窗的整个显示区域。在发生下面几种事件之一时，视窗讯息处理程式会接收到一个 WM_PAINT 讯息：

在使用者移动视窗或显示视窗时，视窗中先前被隐藏的区域重新可见。

使用者改变视窗的大小（如果视窗类别样式有著 CS_HREDRAW 和 CS_VREDRAW 位元旗标的设定）。

程式使用 ScrollWindow 或 ScrollDC 函式滚动显示区域的一部分。

程式使用 InvalidateRect 或 InvalidateRgn 函式刻意产生 WM_PAINT 讯息。

在某些情况下，显示区域的一部分被临时覆盖，Windows 试图保存一个显示区域，并在以後恢复它，但这不一定能成功。在以下情况下，Windows 可能发送 WM_PAINT 讯息：

Windows 擦除覆盖了部分视窗的对话方块或讯息方块。

功能表下拉出来，然後被释放。

显示工具提示讯息。

在某些情况下，Windows 总是保存它所覆盖的显示区域，然後恢复它。这些情况是：

滑鼠游标穿越显示区域。

图示拖过显示区域。

处理 WM_PAINT 讯息要求程式写作者改变自己向显示器输出的思维方式。程式应该组织成可以保留绘制显示区域需要的所有资讯，并且仅当「回应要求」——即 Windows 给视窗讯息处理程式发送 WM_PAINT 讯息时才进行绘制。如果程式在其他时间需要更新其显示区域，它可以强制 Windows 产生一个 WM_PAINT 讯息。这看来似乎是在萤幕上显示内容的一种舍近求远的方法。但您的程式结构可以从中受益。

有效矩形和无效矩形

尽管视窗讯息处理程式一旦接收到 WM_PAINT 讯息之後，就准备更新整个显示区域，但它经常只需要更新一个较小的区域（最常见的是显示区域中的矩形区域）。显然，当对话方块覆盖了部分显示区域时，情况即是如此。在擦除对话方块之後，需要重画的只是先前被对话方块遮住的矩形区域。

这个区域称为「无效区域」或「更新区域」。正是显示区域内无效区域的存在，才会让 Windows 将一个 WM_PAINT 讯息放在应用程式的讯息伫列中。只有在显示区域的某一部分失效时，视窗才会接受 WM_PAINT 讯息。

Windows 内部为每个视窗保存一个「绘图资讯结构」，这个结构包含了包围无效区域的最小矩形的座标以及其他资讯，这个矩形就叫做「无效矩形」，有时也称为「无效区域」。如果在视窗讯息处理程式处理 WM_PAINT 讯息之前显示区域中的另一个区域变为无效，则 Windows 计算出一个包围两个区域的新的无效区域（以及一个新的无效矩形），并将这种变化後的资讯放在绘制资讯结构中。Windows 不会将多个 WM_PAINT 讯息都放在讯息伫列中。

视窗讯息处理程式可以通过呼叫 InvalidateRect 使显示区域内的矩形无效。如果讯息伫列中已经包含一个 WM_PAINT 讯息，Windows 将计算出新的无效矩形。否则，它将一个新的 WM_PAINT 讯息放入讯息伫列中。在接收到 WM_PAINT 讯息时，视窗讯息处理程式可以取得无效矩形的座标（我们马上就会看到这一点）。通过呼叫 GetUpdateRect，可以在任何时候取得这些座标。

在处理 WM_PAINT 讯息处理期间，视窗讯息处理程式在呼叫了 BeginPaint 之後，整个显示区域即变为有效。程式也可以通过呼叫 ValidateRect 函式使显示区域内的任意矩形区域变为有效。如果这呼叫具有令整个无效区域变为有效的效果，则目前伫列中的任何 WM_PAINT 讯息都将被删除。

GDI 简介

要在视窗的显示区域绘图，可以使用 Windows 的图形装置介面（GDI）函式。Windows 提供了几个 GDI 函式，用於将字串输出到视窗的显示区域内。我们已经

在上一章看过 DrawText 函式，但是目前使用最为普遍的文字输出函式是 TextOut。该函式的格式如下：

TextOut (hdc, x, y, psText, iLength) ;

TextOut 向视窗的显示区域写入字串。psText 参数是指向字串的指标，iLength 是字串的长度。x 和 y 参数定义了字串在显示区域的开始位置（不久会讲述关於它们的详细情况）。hdc 参数是「装置内容代号」，它是 GDI 的重要部分。实际上，每个 GDI 函式都需要将这个代号作为函式的第一个参数。

装置内容

读者可能还记得，代号只不过是一个数值，Windows 以它在内部使用物件。程式写作者从 Windows 取得代号，然後在其他函式中使用该代号。装置内容代号是 GDI 函式的视窗「通行证」，有了这种装置内容代号，程式写作者就能自如地在显示区域上绘图，使图形如自己所愿地变得好看或者难看。

装置内容（简称为「DC」）实际上是 GDI 内部保存的资料结构。装置内容与特定的显示设备（如视讯显示器或印表机）相关。对於视讯显示器，装置内容总是与显示器上的特定视窗相关。

装置内容中的有些值是图形「属性」，这些属性定义了 GDI 绘图函式工作的细节。例如，对於 TextOut，装置内容的属性确定了文字的颜色、文字的背景色、x 座标和 y 座标映射到视窗的显示区域的方式，以及显示文字时 Windows 使用的字体。

当程式需要绘图时，它必须先取得装置内容代号。在取得了该代号後，Windows 用内定的属性值填入内部装置内容结构。在後面的章节中您会看到，可以通过呼叫不同的 GDI 函式改变这些预设值。利用其他的 GDI 函式可以取得这些属性的目前值。当然，还有其他的 GDI 函式能够在视窗的显示区域真正地绘图。

当程式在显示区域绘图完毕後，它必须释放装置内容代号。代号被程式释放後就不再有效，且不能再被使用。程式必须在处理单个讯息处理期间取得和释放代号。除了呼叫 CreateDC（函式，在本章暂不讲述）建立的装置内容之外，程式不能在两个讯息之间保存其他装置内容代号。

Windows 应用程式一般使用两种方法来取得装置内容代号，以备在萤幕上绘图。

取得装置内容代号：方法一

在处理 WM_PAINT 讯息时，使用这种方法。它涉及 BeginPaint 和 EndPaint

两个函式，这两个函式需要视窗代号（作为参数传给视窗讯息处理程式）和 PAINTSTRUCT 结构的变数(在 WINUSER.H 表头档案中定义)的地址为参数。Windows 程式写作者通常把这一结构变数命名为 ps 并且在视窗讯息处理程式中定义它：

```
PAINTSTRUCT ps ;
```

在处理 WM_PAINT 讯息时，视窗讯息处理程式首先呼叫 BeginPaint。BeginPaint 函式一般在准备绘制时导致无效区域的背景被擦除。该函式也填入 ps 结构的栏位。BeginPaint 传回的值是装置内容代号，这一传回值通常被保存在叫做 hdc 的变数中。它在视窗讯息处理程式中的定义如下：

```
HDC hdc ;
```

HDC 资料型态定义为 32 位元的无正负号整数。然後，程式就可以使用需要装置内容代号的 TextOut 等 GDI 函式。呼叫 EndPaint 即可释放装置内容代号。

一般地，处理 WM_PAINT 讯息的形式如下：

```
case  WM_PAINT:
      hdc = BeginPaint (hwnd, &ps) ;
         使用 GDI 函式
      EndPaint (hwnd, &ps) ;
      return 0 ;
```

在处理 WM_PAINT 讯息时，必须成对地呼叫 BeginPaint 和 EndPaint。如果视窗讯息处理程式不处理 WM_PAINT 讯息，则它必须将 WM_PAINT 讯息传递给 Windows 中 DefWindowProc（内定视窗讯息处理程式）。DefWindowProc 以下列代码处理 WM_PAINT 讯息：

```
case WM_PAINT:
      BeginPaint (hwnd, &ps) ;
      EndPaint (hwnd, &ps) ;
      return 0 ;
```

这两个 BeginPaint 和 EndPaint 呼叫之间中没有任何叙述，仅仅使先前无效区域变为有效。但以下方法是错误的：

```
case WM_PAINT:
      return 0 ;   // WRONG !!!
```

Windows 将一个 WM_PAINT 讯息放到讯息伫列中，是因为显示区域的一部分无效。如果不呼叫 BeginPaint 和 EndPaint（或者 ValidateRect），则 Windows 不会使该区域变为有效。相反，Windows 将发送另一个 WM_PAINT 讯息，且一直发送下去。

绘图资讯结构

前面提到过，Windows 为每个视窗保存一个「绘图资讯结构」，这就是 PAINTSTRUCT，定义如下：

```
typedef struct tagPAINTSTRUCT
```

```
{
      HDC         hdc ;
      BOOL           fErase ;
      RECT           rcPaint ;
      BOOL        fRestore ;
      BOOL        fIncUpdate ;
      BYTE        rgbReserved[32] ;
} PAINTSTRUCT ;
```

在程式呼叫 BeginPaint 时，Windows 会适当填入该结构的各个栏位值。使用者程式只使用前三个栏位，其他栏位由 Windows 内部使用。hdc 栏位是装置内容代号。在旧版本的 Windows 中，BeginPaint 的传回值也曾是这个装置内容代号。在大多数情况下， fErase 被标志为 FALSE(0)，这意味著 Windows 已经擦除了无效矩形的背景。这最早在 BeginPaint 函式中发生（如果要在视窗讯息处理程式中自己定义一些背景擦除行为，可以自行处理 WM_ERASEBKGND 讯息）。Windows 使用 WNDCLASS 结构的 hbrBackground 栏位指定的画刷来擦除背景，这个 WNDCLASS 结构是程式在 WinMain 初始化期间登录视窗类别时使用的。许多 Windows 程式使用白色画刷。以下叙述设定视窗类别结构栏位值：

```
wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
```

不过，如果程式通过呼叫 Windows 函式 InvalidateRect 使显示区域中的矩形失效，则该函式的最後一个参数会指定是否擦除背景。如果这个参数为 FALSE（即 0),则 Windows 将不会擦除背景,并且在呼叫完 BeginPaint 後 PAINTSTRUCT 结构的 fErase 栏位将为 TRUE（非零）。

PAINTSTRUCT 结构的 rcPaint 栏位是 RECT 型态的结构。您已经在第三章中看到，RECT 结构定义了一个矩形，其四个栏位为 left、top、right 和 bottom。PAINTSTRUCT 结构的 rcPaint 栏位定义了无效矩形的边界，如图 4-1 所示。这些值均以图素为单位，并相对於显示区域的左上角。无效矩形是应该重画的区域。

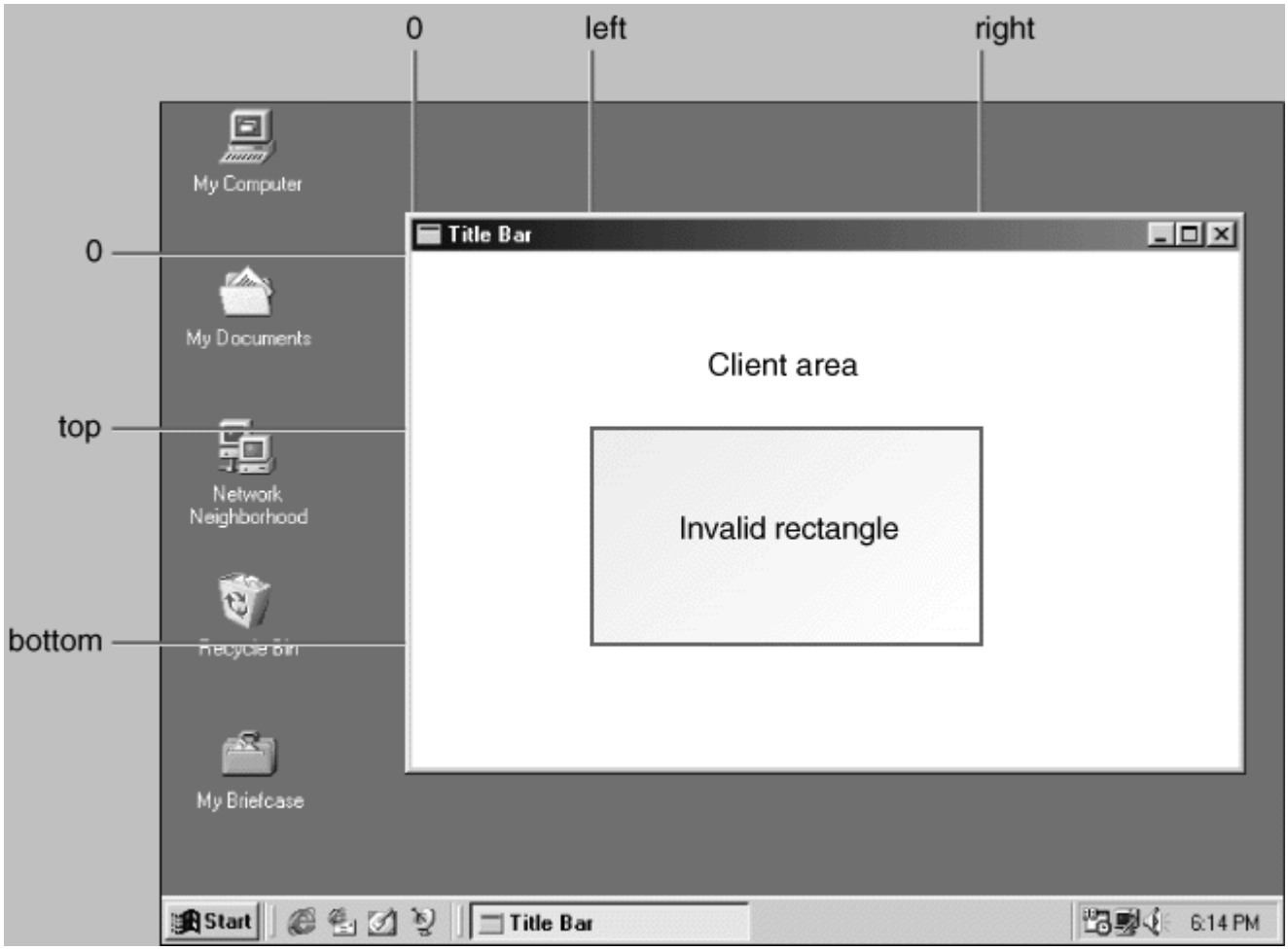

图 4-1 无效矩形的边界

PAINTSTRUCT 中的 rcPaint 矩形不仅是无效矩形，它还是一个「剪取」矩形。这意味著 Windows 将绘图操作限制在剪取矩形内（更确切地说，如果无效矩形区域不为矩形，则 Windows 将绘图操作限制在这个区域内）。

在处理 WM_PAINT 讯息时，为了在更新的矩形外绘图，可以使用如下呼叫：

```
InvalidateRect (hwnd, NULL, TRUE) ;
```

该呼叫在 BeginPaint 呼叫之前进行，它使整个显示区域变为无效，并擦除背景。但是，如果最後一个参数等於 FALSE，则不擦除背景，原有的东西将保留在原处。

通常这是 Windows 程式在无论何时收到 WM_PAINT 讯息而不考虑 rcPaint 结构的情况下简单地重画整个显示区域最方便的方法。例如，如果在显示区域的显示输出中包括了一个圆，但是只有圆的一部分落到了无效矩形中，它就使仅绘制圆的无效部分变得没有意义。这需要画整个圆。在您使用从 BeginPaint 传回的装置内容代号时，Windows 不会绘制 rcPaint 矩形外的任何部分。

在第三章的 HELLOWIN 程式中，我们并不关心处理 WM_PAINT 讯息时的无效矩形。如果文字显示区域恰巧在无效矩形内，则由 DrawText 恢复之。否则，在处理 DrawText 呼叫的某个时刻，Windows 会确定它无须向显示器上输出。不过，

这一决定需要时间。关心程式性能和速度的程式写作者希望在处理 WM_PAINT 期间使用无效矩形范围，以避免不必要的 GDI 呼叫。如果绘制时需要存取例如点阵图这样的磁片档案，则这就显得尤其重要。

取得装置内容代号：方法二

虽然最好是在处理 WM_PAINT 讯息处理期间更新整个显示区域，但是您也会发现在处理非 WM_PAINT 讯息处理期间绘制显示区域的某个部分也是非常有用的。或者您需要将装置内容代号用於其他目的，如取得装置内容的资讯。

要得到视窗显示区域的装置内容代号，可以呼叫 GetDC 来取得代号，在使用完後呼叫 ReleaseDC：

```
hdc = GetDC (hwnd) ;
使用 GDI 函式
ReleaseDC (hwnd, hdc) ;
```

与 BeginPaint 和 EndPaint 一样，GetDC 和 ReleaseDC 函式必须成对地使用。如果在处理某讯息时呼叫 GetDC，则必须在退出视窗讯息处理程式之前呼叫 ReleaseDC。不要在一个讯息中呼叫 GetDC 却在另一个讯息呼叫 ReleaseDC。

与从 BeginPaint 传回装置内容代号不同，GetDC 传回的装置内容代号具有一个剪取矩形，它等於整个显示区域。可以在显示区域的某一部分绘图，而不只是在无效矩形上绘图（如果确实存在无效矩形）。与 BeginPaint 不同，GetDC 不会使任何无效区域变为有效。如果需要使整个显示区域有效，可以呼叫

```
ValidateRect (hwnd, NULL) ;
```

一般可以呼叫 GetDC 和 ReleaseDC 来对键盘讯息（如在字处理程式中）和滑鼠讯息（如在画图程式中）作出反应。此时，程式可以立刻根据使用者的键盘或滑鼠输入来更新显示区域，而不需要考虑为了视窗的无效区域而使用 WM_PAINT 讯息。不过，一旦确实收到了 WM_PAINT 讯息，程式就必须要收集足够的资讯後才能更新显示。

与 GetDC 相似的函式是 GetWindowDC。GetDC 传回用於写入视窗显示区域的装置内容代号，而 GetWindowDC 传回写入整个视窗的装置内容代号。例如，您的程式可以使用从 GetWindowDC 传回的装置内容代号在视窗的标题列上写入文字。然而，程式同样也应该处理 WM_NCPAINT （「非显示区域绘制」）讯息。

TextOut：细节

TextOut 是用於显示文字的最常用的 GDI 函式。语法是：

```
TextOut (hdc, x, y, psText, iLength) ;
```

以下将详细地讨论这个函式。

第一个参数是装置内容代号，它既可以是 GetDC 的传回值，也可以是在处理 WM_PAINT 讯息时 BeginPaint 的传回值。

装置内容的属性控制了被显示的字串的特徵。例如，装置内容中有一个属性指定文字颜色，内定颜色为黑色；内定装置内容还定义了白色的背景。在程式向显示器输出文字时，Windows 使用这个背景色来填入字元周围的矩形空间（称为「字元框」）。

该文字背景色与定义视窗类别时设置的背景并不相同。视窗类别中的背景是一个画刷，它是一种纯色或者非纯色组成的画刷，Windows 用它来擦除显示区域，它不是装置内容结构的一部分。在定义视窗类别结构时，大多数 Windows 应用程式使用 WHITE_BRUSH，以便内定装置内容中的内定文字背景颜色与 Windows 用以擦除显示区域背景的画刷颜色相同。

psText 参数是指向字串的指标，iLength 是字串中字元的个数。如果 psText 指向 Unicode 字串，则字串中的位元组数就是 iLength 值的两倍。字串中不能包含任何 ASCII 控制字元（如回车、换行、制表或退格），Windows 会将这些控制字元显示为实心块。TextOut 不识别作为字串结束标志的内容为零的位元组（对於 Unicode，是一个短整数型态的 0），而需要由 nLength 参数指明长度。

TextOut 中的 x 和 y 定义显示区域内字串的开始位置，x 是水平位置，y 是垂直位置。字串中第一个字元的左上角位於座标点(x,y)。在内定的装置内容中，原点（x 和 y 均为 0 的点）是显示区域的左上角。如果在 TextOut 中将 x 和 y 设为 0，则将从显示区域左上角开始输出字串。

当您阅读 GDI 绘图函式（例如 TextOut）的文件时，就会发现传递给函式的座标常常被称为「逻辑座标」。在第五章会详细地解释这种情况。现在请注意，Windows 有许多「座标映射方式」，它们用来控制 GDI 函式指定的逻辑座标转换为显示器的实际图素座标的方式。映射方式在装置内容中定义，内定映射方式是 MM_TEXT（使用 WINGDI.H 中定义的识别字）。在 MM_TEXT 映射方式下，逻辑单位与实际单位相同，都是图素；x 的值从左向右递增，y 的值从上向下递增（参看图 4-2）。MM_TEXT 座标系与 Windows 在 PAINTSTRUCT 结构中定义无效矩形时使用的座标系相同，这为我们带来了很多方便（但是，其他映射方式并非如此）。

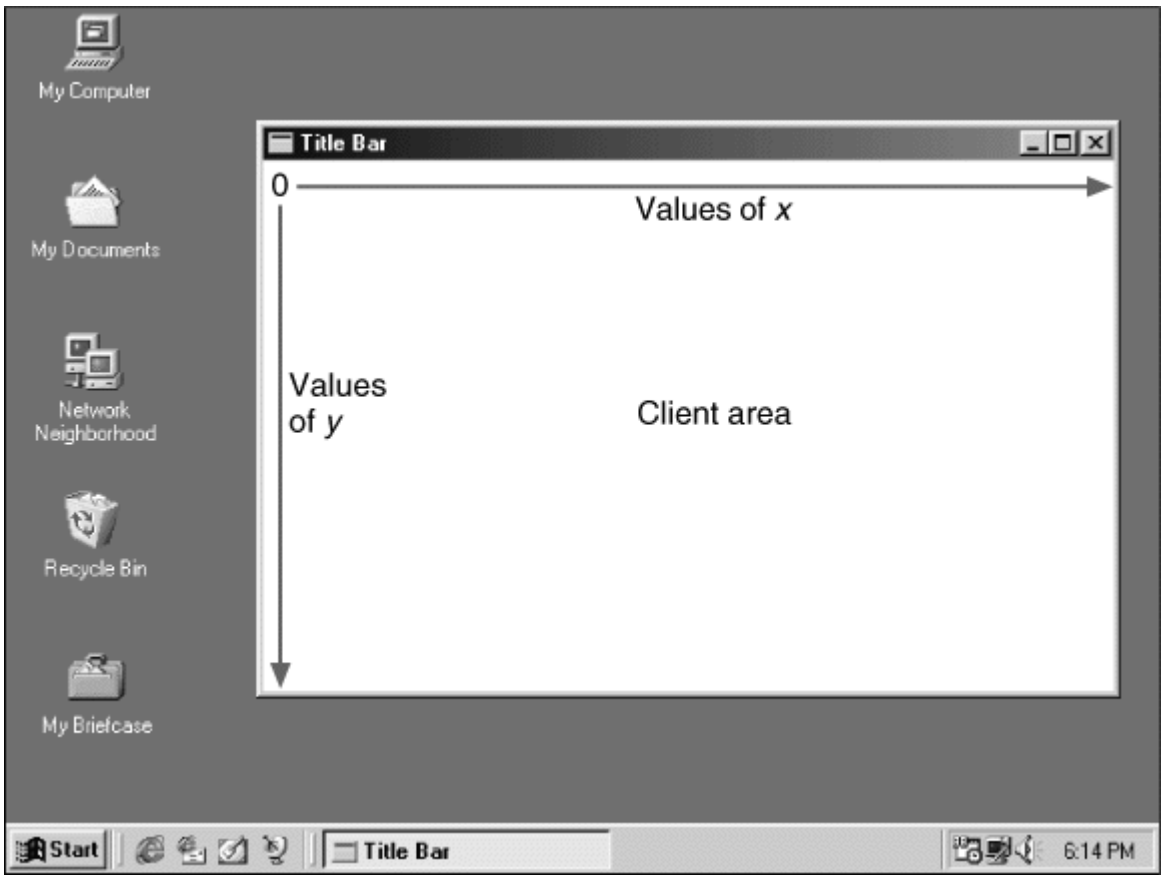

图 4-2 MM_TEXT 映射方式下的 x 座标和 y 座标

装置内容也定义了一个剪裁区域。您已经看到，对於从 GetDC 取得的装置内容代号，内定剪裁区域是整个显示区域；而对於从 BeginPaint 取得的装置内容代号，则为无效区域。Windows 不会在剪裁区域之外的任何位置显示字串。如果一个字元有一部分在剪裁区域外，则 Windows 将只显示此区域内的那部分。要想将输出写到视窗的显示区域之外不是那么容易的，所以不用担心会无意间出现这种事情。

系统字体

装置内容还定义了在您呼叫 TextOut 显示文字时 Windows 使用的字体。内定字体为「系统字体」，或用 Windows 表头档案中的识别字，即 SYSTEM_FONT。系统字体是 Windows 用来在标题列、功能表和对话方块中显示字串的内定字体。

在 Windows 的早期版本中，系统字体是等宽(fixed-pitch)字体，这意味著所有字元均具有同样的宽度，非常类似於打字机。然而，从 Windows 3.0 开始，系统字体成为一种变宽(variable-pitch)字体，这意味著不同的字元具有不同的大小，比如，「W」要比「i」宽。变宽字体比等宽字体好读，这已经是公认的事实。不过，可以想见，这一转变使很多原来的 Windows 程式码不再适用，从而要求程式写作者学习一些使用字体的新技术。

系统字体是一种「点阵字体」，这意味著字元被定义为图素块（在第十七

章，将讨论 TrueType 字体，它是由轮廓定义的）。至於确切的大小，系统字体的字元大小取决於视讯显示器的大小。系统字体设计为至少能在显示器上显示 25 行 80 列文字。

字元大小

要用 TextOut 显示多行文字，就必须确定字体的字元大小，可以根据字元的高度来定位字元的後续行，以及根据字元的宽度来定位字元的後续列。

系统字体的字元高度和平均宽度是多少？这个问题取决於视讯显示器的图素大小。Windows 需要的最小显示大小是 640 480，但是许多使用者更喜欢 800 600 或 1024 768 的显示大小。另外，对於这些较大的显示尺寸，Windows 允许使用者选择不同大小的系统字体。

程式可以呼叫 GetSystemMetrics 函式以取使用者介面上各类视觉元件大小的资讯，呼叫 GetTextMetrics 取得字体大小。GetTextMetrics 传回装置内容中目前选取的字体资讯，因此它需要装置内容代号。Windows 将文字大小的不同值复制到在 WINGDI.H 中定义的 TEXTMETRIC 型态的结构中。TEXTMETRIC 结构有 20 个栏位，我们只使用前七个：

```
typedef struct tagTEXTMETRIC
{
      LONG tmHeight ;
      LONG tmAscent ;
      LONG tmDescent ;
      LONG tmInternalLeading ;
      LONG tmExternalLeading ;
      LONG tmAveCharWidth ;
   LONG tmMaxCharWidth ;
            其他结构栏位
}
TEXTMETRIC, * PTEXTMETRIC ;
```

这些栏位值的单位取决於选定的装置内容映射方式。在内定装置内容下，映射方式是 MM_TEXT，因此值的大小是以图素为单位。

要使用 GetTextMetrics 函式，需要先定义一个结构变数（通常称为 tm）：

```
TEXTMETRIC tm ;
```

在需要确定文字大小时，先取得装置内容代号，再呼叫 GetTextMetrics：

```
hdc = GetDC (hwnd) ;
GetTextMetrics (hdc, &tm) ;
ReleaseDC (hwnd, hdc) ;
```

此後，您就可以查看文字尺寸结构中的值，并有可能保存其中的一些以备将来使用。

文字大小：细节

TEXTMETRIC 结构提供了关於目前装置内容中选用的字体的丰富资讯。但是，字体的纵向大小只由 5 个值确定，其中 4 个值如图 4-3 所示。

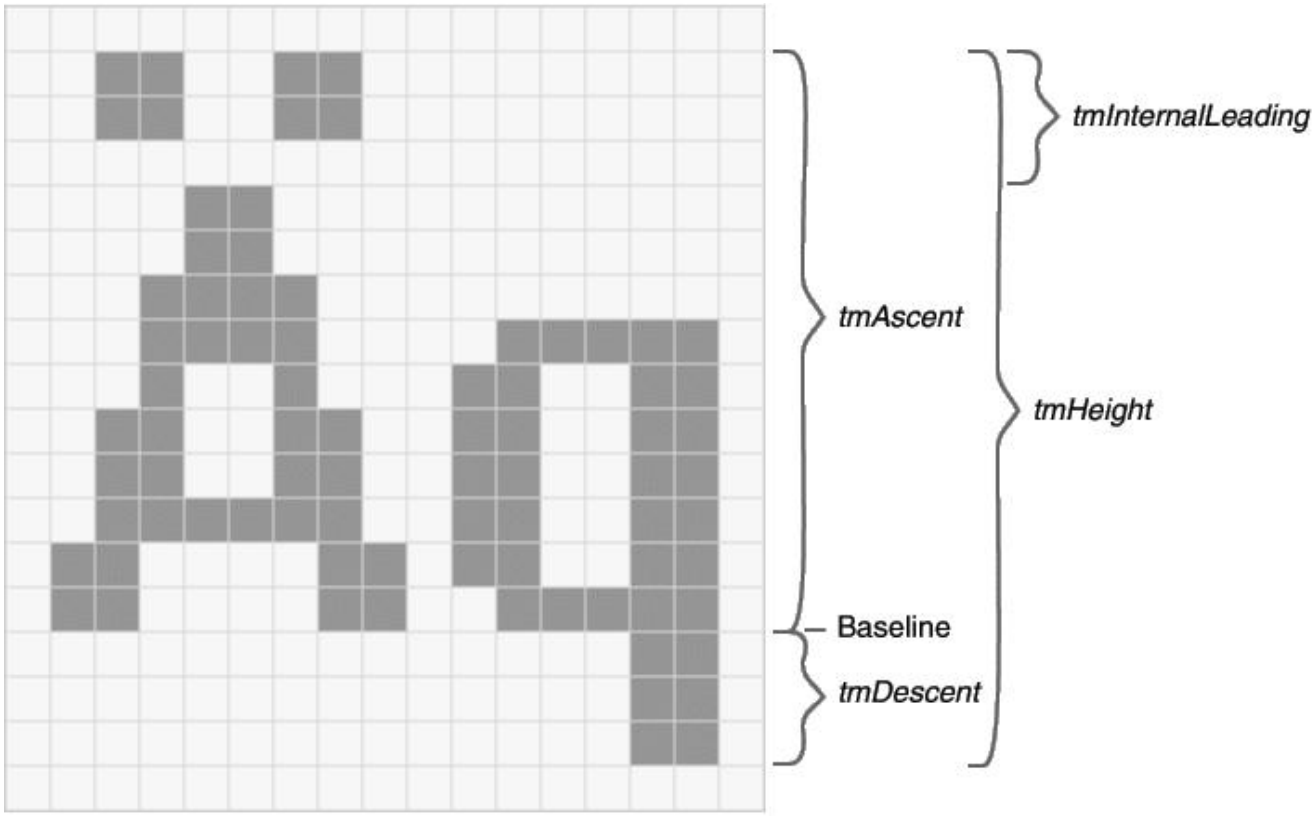

图 4-3 定义字体中纵向字元大小的 4 个值

最重要的值是 tmHeight，它是 tmAscent 和 tmDescent 的和。这两个值表示了基准线上下字元的最大纵向高度。「间距」（leading）指印表机在两行文字间插入的空间。在 TEXTMETRIC 结构中，内部的间距包括在 tmAscent 中（因此也在 tmHeight 中），并且它经常是重音符号出现的地方。tmInternalLeading 栏位可被设成 0，在这种情况下，加重音的字母会稍稍缩短以便容纳重音符号。

TEXTMETRIC 结构还包括一个不包含在 tmHeight 值中的栏位 tmExternalLeading。它是字体设计者建议加在横向字元之间的空间大小。在安排文字行之间的空隙时，您可以接受设计者建议的值，也可以拒绝它。在系统字体中 tmExternalLeading 可以为 0，因此我没有在图 4-3 中显示它。（尽管我不想告诉你们，图 4-3 确实就是 Windows 在 640 480 的显示解析度中使用的系统字体。）

TEXTMETRICS 结构包含有描述字元宽度的两个栏位，即 tmAveCharWidth（小写字母加权平均宽度）和 tmMaxCharWidth（字体中最宽字元的宽度）。对於定宽字体，这两个值是相等的（图 4-3 中这些值分别为 7 和 14）。

本章的范例程式还需要另一种字元宽度，即大写字母的平均宽度，这可以

用 tmAveCharWidth 乘以 150%大致计算出来。

必须认识到，系统字体的大小取决於 Windows 所执行的视讯显示器的解析度，在某些情况下，取决於使用者选取的系统字体的大小。Windows 提供了一个与装置无关的图形介面，但程式写作者还是有事情要处理的。不要想当然耳地猜测字体大小来写作 Windows 程式，也不要把值定死，您可以使用 GetTextMetrics 函式取得这一资讯。

格式化文字

Windows 启动後，系统字体的大小就不会发生改变，所以在程式执行过程中，程式写作者只需要呼叫一次 GetTexMetrics。最好是在视窗讯息处理程式中处理 WM_CREATE 讯息时进行此呼叫，WM_CREATE 讯息是视窗讯息处理程式接收的第一个讯息。在 WinMain 中呼叫 CreateWindow 时，Windows 会以一个 WM_CREATE 讯息呼叫视窗讯息处理程式。

假设要编写一个 Windows 程式，在显示区域显示几行文字，这需要先取得字元宽度和高度。您可以在视窗讯息处理程式内定义两个变数来保存平均字元宽度(cxChar)和总的字元高度(cyChar)：

```
static int cxChar, cyChar ;
```

变数名的字首 c 代表「count」，在这里指图素数，与 x 和 y 结合，分别指宽和高。这些变数定义为 static 静态变数，因为它们在视窗讯息处理程式中处理其他讯息（如 WM_PAINT）时也应该是有效的。如果变数在函式外面定义，则不需要定义为 static。

下面是取得系统字体的字元宽度和高度的 WM_CREATE 程式码：

```
case WM_CREATE:
     hdc = GetDC (hwnd) ;
     GetTextMetrics (hdc, &tm) ;

     cxChar = tm.tmAveCharWidth ;
     cyChar = tm.tmHeight + tm.tmExternalLeading ;

     ReleaseDC (hwnd, hdc) ;
     return 0 ;
```

注意我在计算 cyChar 时包括了 tmExternalLeading 栏位，虽然该栏位在系统字体中为 0，但是因为它使得文字的可读性更好，所以还是应该把它包括进去。沿著视窗向下每隔 cyChar 图素就会显示一行文字。

您会发现常常需要显示格式化的数字跟简单的字串。我在第二章讲到过，您不能使惯用的工具（可爱的 printf 函式）来完成这项工作，但是可以使用 sprintf 和 Windows 版的 sprintf——wsprintf。这些函式与 printf 相似，只

是把格式化字串放到字串中。然後，可以用 TextOut 将字串输出到显示器上。非常方便的是，从 sprintf 和 wsprintf 传回的值就是字串的长度。您可以将这个值传递给 TextOut 作为 iLength 参数。下面的程式码显示了 wsprintf 与 TextOut 的典型组合：

```
int   iLength ;
TCHAR szBuffer [40] ;
```

其他行程式

```
iLength = wsprintf (szBuffer, TEXT ("The sum of %i and %i is %i"),
                        iA, iB, iA + iB) ;
TextOut (hdc, x, y, szBuffer, iLength) ;
```

对於这样简单的情况，可以将 nLength 的定义值与 TextOut 放在同一条叙述中，从而无需定义 iLength：

```
TextOut (hdc, x, y, szBuffer,
      wsprintf (szBuffer, TEXT ("The sum of %i and %i is %i"),
               iA, iB, iA + iB)) ;
```

虽然这样子写起来不好看，但是功能与前者是一样的。

综合使用

现在，我们似乎已经具备了在萤幕上显示多行文字所需要的所有知识。我们知道如何在 WM_PAINT 讯息处理期间取得一个装置内容代号，如何使用 TextOut 函式以及如何根据字元大小来安排字距，剩下的就是显示一点有意义的东西了。

在前一章里，我们大概知道从 Windows 的 GetSystemMetrics 函式中取得的资讯是很有意义的，该函式传回 Windows 中不同视觉元件的大小资讯，如图示、游标、标题列和卷动列等。它们的大小因显示卡和驱动程式的不同而有所不同。GetSystemMetrics 是在程式中完成与装置无关图形输出的重要函式。

该函式需要一个参数，叫做「索引」，在 Windows 表头档案定义了 75 个整数索引识别字（识别字的数量随著每个版本的 Windows 的发布而不断地增加，在 Windows 1.0 的程式写作者文件中仅列出了 26 个）。GetSystemMetrics 传回一个整数，这个整数通常就是参数中指定的图形元件大小。

让我们来编写一个程式，显示一些可以从 GetSystemMetrics 呼叫中取得的资讯，显示格式为每种视觉元件一行。如果我们建立一个表头档案，在表头档案中定义一个结构阵列，此结构包含 GetSystemMetrics 索引对应的 Windows 表头档案识别字和呼叫所传回的每个值对应的字串，这样处理起来要容易一些。表头档案名为 SYSMETS.H，如程式 4-1 所示。

程式 4-1　SYSMETS.H

```
/*-------------------------------------------------------
 SYSMETS.H -- System metrics display structure
```

```
------------------------------------------------------------*/
#define NUMLINES ((int) (sizeof sysmetrics / sizeof sysmetrics [0]))
struct
{
      int   Index ;
      TCHAR *     szLabel ;
      TCHAR *     szDesc ;
}
sysmetrics [] =
{
      SM_CXSCREEN,    TEXT ("SM_CXSCREEN"),
                  TEXT ("Screen width in pixels"),
      SM_CYSCREEN,      TEXT ("SM_CYSCREEN"),
                        TEXT ("Screen height in pixels"),
      SM_CXVSCROLL,     TEXT ("SM_CXVSCROLL"),
                        TEXT ("Vertical scroll width"),
      SM_CYHSCROLL,     TEXT ("SM_CYHSCROLL"),
                        TEXT ("Horizontal scroll height"),
      SM_CYCAPTION,     TEXT ("SM_CYCAPTION"),
                     TEXT ("Caption bar height"),
      SM_CXBORDER,      TEXT ("SM_CXBORDER"),
                        TEXT ("Window border width"),
      SM_CYBORDER,      TEXT ("SM_CYBORDER"),
                        TEXT ("Window border height"),
      SM_CXFIXEDFRAME,TEXT ("SM_CXFIXEDFRAME"),
                        TEXT ("Dialog window frame width"),
      SM_CYFIXEDFRAME,TEXT ("SM_CYFIXEDFRAME"),
                        TEXT ("Dialog window frame height"),
      SM_CYVTHUMB,      TEXT ("SM_CYVTHUMB"),
                        TEXT ("Vertical scroll thumb height"),
      SM_CXHTHUMB,    TEXT ("SM_CXHTHUMB"),
                           TEXT ("Horizontal scroll thumb width"),
      SM_CXICON,      TEXT ("SM_CXICON"),
                        TEXT ("Icon width"),
      SM_CYICON,      TEXT ("SM_CYICON"),
                        TEXT ("Icon height"),
      SM_CXCURSOR,      TEXT ("SM_CXCURSOR"),
                        TEXT ("Cursor width"),
      SM_CYCURSOR,    TEXT ("SM_CYCURSOR"),
                        TEXT ("Cursor height"),
      SM_CYMENU,      TEXT ("SM_CYMENU"),
                        TEXT ("Menu bar height"),
      SM_CXFULLSCREEN,TEXT ("SM_CXFULLSCREEN"),
                        TEXT ("Full screen client area width"),
      SM_CYFULLSCREEN,TEXT ("SM_CYFULLSCREEN"),
                        TEXT ("Full screen client area height"),
      SM_CYKANJIWINDOW,TEXT ("SM_CYKANJIWINDOW"),
                         TEXT ("Kanji window height"),
```

```
SM_MOUSEPRESENT, TEXT ("SM_MOUSEPRESENT"),
                 TEXT ("Mouse present flag"),
SM_CYVSCROLL,    TEXT ("SM_CYVSCROLL"),
                 TEXT ("Vertical scroll arrow height"),
SM_CXHSCROLL,    TEXT ("SM_CXHSCROLL"),
                 TEXT ("Horizontal scroll arrow width"),
SM_DEBUG,        TEXT ("SM_DEBUG"),
                 TEXT ("Debug version flag"),
SM_SWAPBUTTON,   TEXT ("SM_SWAPBUTTON"),
                 TEXT ("Mouse buttons swapped flag"),
SM_CXMIN,        TEXT ("SM_CXMIN"),
                 TEXT ("Minimum window width"),
SM_CYMIN,        TEXT ("SM_CYMIN"),
                 TEXT ("Minimum window height"),
SM_CXSIZE,       TEXT ("SM_CXSIZE"),
                 TEXT ("Min/Max/Close button width"),
SM_CYSIZE,       TEXT ("SM_CYSIZE"),
                 TEXT ("Min/Max/Close button height"),
SM_CXSIZEFRAME,  TEXT ("SM_CXSIZEFRAME"),
                 TEXT ("Window sizing frame width"),
SM_CYSIZEFRAME,  TEXT ("SM_CYSIZEFRAME"),
                 TEXT ("Window sizing frame height"),
SM_CXMINTRACK,   TEXT ("SM_CXMINTRACK"),
                 TEXT ("Minimum window tracking width"),
SM_CYMINTRACK,   TEXT ("SM_CYMINTRACK"),
                 TEXT ("Minimum window tracking height"),
SM_CXDOUBLECLK,  TEXT ("SM_CXDOUBLECLK"),
                 TEXT ("Double click x tolerance"),
SM_CYDOUBLECLK,  TEXT ("SM_CYDOUBLECLK"),
                 TEXT ("Double click y tolerance"),
SM_CXICONSPACING,TEXT ("SM_CXICONSPACING"),
                 TEXT ("Horizontal icon spacing"),
SM_CYICONSPACING,TEXT ("SM_CYICONSPACING"),
                 TEXT ("Vertical icon spacing"),
SM_MENUDROPALIGNMENT,  TEXT ("SM_MENUDROPALIGNMENT"),
                       TEXT ("Left or right menu drop"),
SM_PENWINDOWS,         TEXT ("SM_PENWINDOWS"),
                       TEXT ("Pen extensions installed"),
SM_DBCSENABLED,        TEXT ("SM_DBCSENABLED"),
                       TEXT ("Double-Byte Char Set enabled"),
SM_CMOUSEBUTTONS,   TEXT ("SM_CMOUSEBUTTONS"),
                       TEXT ("Number of mouse buttons"),
SM_SECURE,          TEXT ("SM_SECURE"),
                       TEXT ("Security present flag"),
SM_CXEDGE,          TEXT ("SM_CXEDGE"),
                       TEXT ("3-D border width"),
SM_CYEDGE,          TEXT ("SM_CYEDGE"),
                       TEXT ("3-D border height"),
```

```
    SM_CXMINSPACING,     TEXT ("SM_CXMINSPACING"),
                             TEXT ("Minimized window spacing width"),
    SM_CYMINSPACING,     TEXT ("SM_CYMINSPACING"),
                             TEXT ("Minimized window spacing height"),
    SM_CXSMICON,           TEXT ("SM_CXSMICON"),
                             TEXT ("Small icon width"),
    SM_CYSMICON,           TEXT ("SM_CYSMICON"),
                             TEXT ("Small icon height"),
    SM_CYSMCAPTION,        TEXT ("SM_CYSMCAPTION"),
                             TEXT ("Small caption height"),
    SM_CXSMSIZE,           TEXT ("SM_CXSMSIZE"),
                             TEXT ("Small caption button width"),
    SM_CYSMSIZE,           TEXT ("SM_CYSMSIZE"),
                             TEXT ("Small caption button height"),
    SM_CXMENUSIZE,         TEXT ("SM_CXMENUSIZE"),
                             TEXT ("Menu bar button width"),
    SM_CYMENUSIZE,         TEXT ("SM_CYMENUSIZE"),
                             TEXT ("Menu bar button height"),
    SM_ARRANGE,          TEXT ("SM_ARRANGE"),
                             TEXT ("How minimized windows arranged"),
    SM_CXMINIMIZED,        TEXT ("SM_CXMINIMIZED"),
                             TEXT ("Minimized window width"),
    SM_CYMINIMIZED,        TEXT ("SM_CYMINIMIZED"),
                             TEXT ("Minimized window height"),
    SM_CXMAXTRACK,         TEXT ("SM_CXMAXTRACK"),
                             TEXT ("Maximum draggable width"),
    SM_CYMAXTRACK,         TEXT ("SM_CYMAXTRACK"),
                             TEXT ("Maximum draggable height"),
    SM_CXMAXIMIZED,        TEXT ("SM_CXMAXIMIZED"),
                             TEXT ("Width of maximized window"),
    SM_CYMAXIMIZED,        TEXT ("SM_CYMAXIMIZED"),
                             TEXT ("Height of maximized window"),
    SM_NETWORK,            TEXT ("SM_NETWORK"),
                             TEXT ("Network present flag"),
    SM_CLEANBOOT,         TEXT ("SM_CLEANBOOT"),
                             TEXT ("How system was booted"),
    SM_CXDRAG,           TEXT ("SM_CXDRAG"),
                             TEXT ("Avoid drag x tolerance"),
    SM_CYDRAG,           TEXT ("SM_CYDRAG"),
                             TEXT ("Avoid drag y tolerance"),
    SM_SHOWSOUNDS,         TEXT ("SM_SHOWSOUNDS"),
                             TEXT ("Present sounds visually"),
    SM_CXMENUCHECK,        TEXT ("SM_CXMENUCHECK"),
                             TEXT ("Menu check-mark width"),
    SM_CYMENUCHECK,        TEXT ("SM_CYMENUCHECK"),
                           TEXT ("Menu check-mark height"),
    SM_SLOWMACHINE,        TEXT ("SM_SLOWMACHINE"),
                             TEXT ("Slow processor flag"),
```

```
      SM_MIDEASTENABLED,        TEXT ("SM_MIDEASTENABLED"),
                                TEXT ("Hebrew and Arabic enabled flag"),
      SM_MOUSEWHEELPRESENT,  TEXT ("SM_MOUSEWHEELPRESENT"),
                                TEXT ("Mouse wheel present flag"),
      SM_XVIRTUALSCREEN,        TEXT ("SM_XVIRTUALSCREEN"),
                                TEXT ("Virtual screen x origin"),
      SM_YVIRTUALSCREEN,        TEXT ("SM_YVIRTUALSCREEN"),
                                TEXT ("Virtual screen y origin"),
      SM_CXVIRTUALSCREEN,       TEXT ("SM_CXVIRTUALSCREEN"),
                                TEXT ("Virtual screen width"),
      SM_CYVIRTUALSCREEN,       TEXT ("SM_CYVIRTUALSCREEN"),
                                TEXT ("Virtual screen height"),
      SM_CMONITORS,           TEXT ("SM_CMONITORS"),
                                TEXT ("Number of monitors"),
      SM_SAMEDISPLAYFORMAT,  TEXT ("SM_SAMEDISPLAYFORMAT"),
                                TEXT ("Same color format flag")
} ;
```

显示资讯的程式命名为 SYSMETS1。SYSMETS1.C 的原始码如程式 4-2 所示。现在大多数程式码看起来都很熟悉。WinMain 中的程式码实际上与 HELLOWIN 中的程式码相同，并且 WndProc 中的大部分程式码都已经讨论过了。

程式 4-2　SYSMETS1.C

```
/*------------------------------------------------------------------
 SYSMETS1.C -- System Metrics Display Program No. 1
                 (c) Charles Petzold, 1998
  ----------------------------------------------------------------*/
#include <windows.h>
#include "sysmets.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                PSTR szCmdLine, int iCmdShow)
{
      static TCHAR szAppName[] = TEXT ("SysMets1") ;
      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 ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;

        return 0 ;
    }
     hwnd = CreateWindow (szAppName, TEXT ("Get System Metrics No. 1"),
                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 int  cxChar, cxCaps, cyChar ;
     HDC              hdc ;
     int              i ;
     PAINTSTRUCT ps ;
     TCHAR            szBuffer [10] ;
     TEXTMETRIC  tm ;

     switch (message)
    {
     case  WM_CREATE:
          hdc = GetDC (hwnd) ;

          GetTextMetrics (hdc, &tm) ;
          cxChar = tm.tmAveCharWidth ;
          cxCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2 ;
          cyChar = tm.tmHeight + tm.tmExternalLeading ;

          ReleaseDC (hwnd, hdc) ;
          return 0 ;

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

          for (i = 0 ; i < NUMLINES ; i++)
```

```
                {
            TextOut     (hdc, 0, cyChar * i,
                        sysmetrics[i].szLabel,
                        lstrlen (sysmetrics[i].szLabel)) ;

            TextOut     (hdc, 22 * cxCaps, cyChar * i,
                        sysmetrics[i].szDesc,
                        lstrlen (sysmetrics[i].szDesc)) ;

                    SetTextAlign (hdc, TA_RIGHT | TA_TOP) ;
          TextOut (hdc, 22 * cxCaps + 40 * cxChar, cyChar * i, szBuffer,
      wsprintf (szBuffer, TEXT ("%5d"),
      GetSystemMetrics (sysmetrics[i].iIndex))) ;
                SetTextAlign (hdc, TA_LEFT | TA_TOP) ;
        }
            EndPaint (hwnd, &ps) ;
            return 0 ;
      case  WM_DESTROY :
            PostQuitMessage (0) ;
      return 0 ;
    }
      return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

图 4-4 显示了在标准 VGA 上执行的 SYSMETS1。在程式显示区域的前两行可以看到，萤幕宽度是 640 个图素，萤幕高度是 480 个图素，这两个值以及程式所显示的其他值可能会因视讯显示器型态的不同而不同。

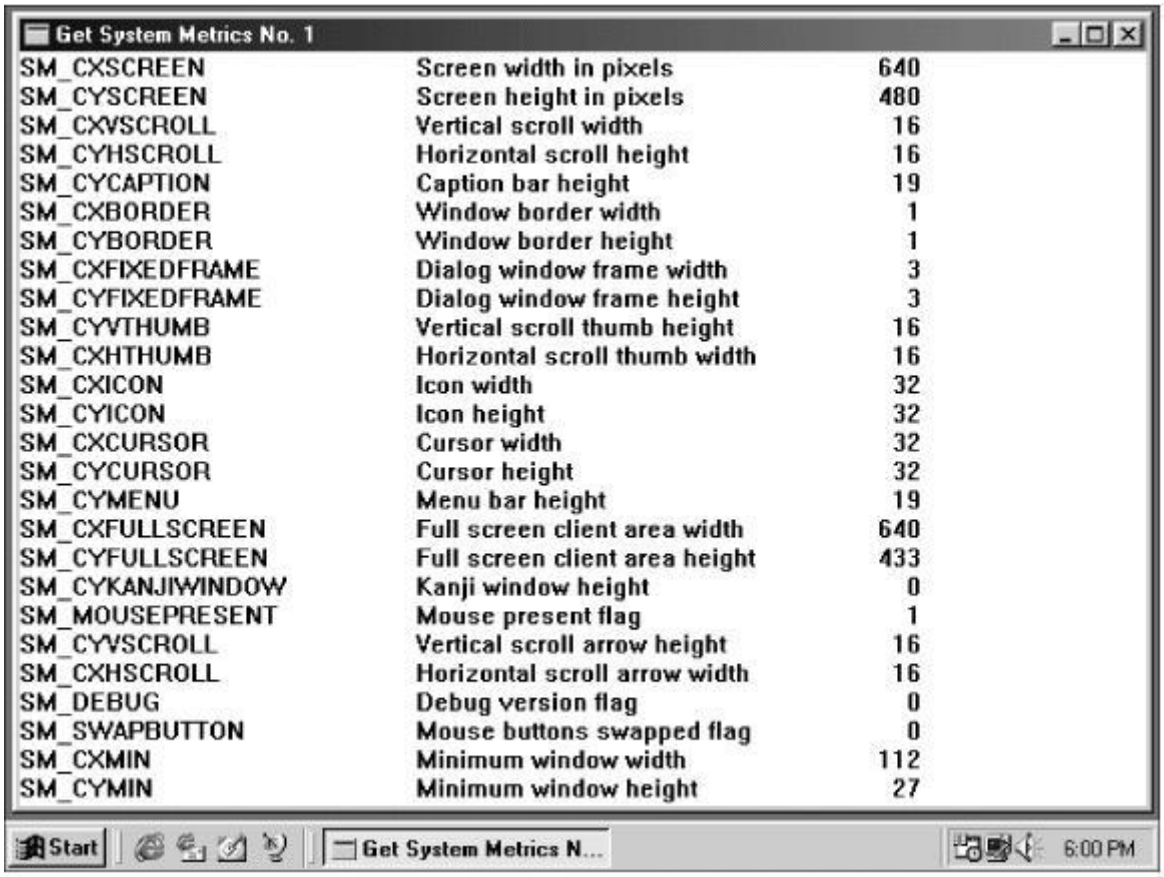

图 4-4 SYSMETS1 的显示

SYSMETS1.C 视窗讯息处理程式

SYSMETS1.C 程式中的 WndProc 视窗讯息处理程式处理三个讯息：WM_CREATE、WM_PAINT 和 WM_DESTROY。WM_DESTROY 讯息的处理方法与第三章的 HELLOWIN 程式相同。

WM_CREATE 讯息是视窗讯息处理程式接收到的第一个讯息。在 CreateWindow 函式建立视窗时，Windows 产生这个讯息。在处理 WM_CREATE 讯息时，SYSMETS1 呼叫 GetDC 取得视窗的装置内容，并呼叫 GetTextMetrics 取得内定系统字体的文字大小。SYSMETS1 将平均字元宽度保存在 cxChar 中，将字元的总高度（包括外部间距）保存在 cyChar 中。

SYSMETS1 还将大写字母的平均宽度保存在静态变数 cxCaps 中。对於固定宽度的字体， cxCaps 等於 cxChar。对於可变宽度字体，cxCaps 设定为 cxChar 乘以 150%。对於可变宽度字体，TEXTMETRIC 结构中的 tmPitchAndFamily 栏位的低位元为 1，对於固定宽度字体，该值为 0。 SYSMETS1 使用这个位元从 cxChar 计算 cxCaps：

```
cxCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2 ;
```

SYSMETS1 在处理 WM_PAINT 讯息处理期间完成所有视窗建立工作。通常，视窗讯息处理程式先呼叫 BeginPaint 取得装置内容代号，然後用一道 for 叙述对

SYSMETS.H 中定义的 sysmetrics 结构的每一行进行回圈。三列文字用三个 TextOut 函式显示，对於每一列，TextOut 的第三个参数都设定为：

```
cyChar * i
```

这个参数指示了字串顶端相对於显示区域顶部的图素位置。

第一条 TextOut 叙述在第一列显示了大写识别字。TextOut 的第二个参数是 0，这是说文字从显示区域的左边缘开始。文字的内容来自 sysmetrics 结构的 szLabel 栏位。我使用 Windows 函式 lstrlen 来计算字串的长度，它是 TextOut 需要的最後一个参数。

第二条 TextOut 叙述显示了对系统尺寸值的描述。这些描述存放在 sysmetrics 结构的 szDesc 栏位中。在这种情况下，TextOut 的第二个参数设定为：

```
22 * cxCaps
```

第一列显示的最长的大写识别字有 20 个字元，因此第二列必须在第一列文字开头向右 20 cxCaps 处开始。我使用 22，以在两列之间加一点多余的空间。

第三条 TextOut 叙述显示从 GetSystemMetrics 函式取得的数值。变宽字体使得格式化向右对齐的数值有些棘手。从 0 到 9 的数字具有相同的宽度，但是这个宽度比空格宽度大。数值可以比一个数字宽，所以不同的数值应该从不同的横向位置开始。

那么，如果我们指定字串结束的图素位置，而不是指定字串的开始位置，以此向右对齐数值，是否会容易一些呢？用 SetTextAlign 函式就可以做到这一点。在 SYSMETS1 呼叫：

```
SetTextAlign (hdc, TA_RIGHT | TA_TOP) ;
```

之後，传给後续 TextOut 函式的座标将指定字串的右上角，而不是左上角。

显示列数的 TextOut 函式的第二个参数设定为：

```
22 * cxCaps + 40 * cxChar
```

值 40*cxChar 包含了第二列的宽度和第三列的宽度。在 TextOut 函式之後，另一个对 SetTextAlign 的呼叫将对齐方式设定回普通方式，以进行下次回圈。

空间不够

在 SYSMETS1 程式中存在著一个很难处理的问题：除非您有一个大萤幕跟高解析度的显示卡，否则就无法看到系统尺度列表的最後几行。如果视窗太窄，甚至根本看不到值。

SYSMETS1 不知道这个问题。否则我们就会显示一个讯息方块说「抱歉！」程式甚至不知道它的显示区域有多大，它从视窗顶部开始输出文字，并仰赖 Windows 裁剪超出显示区域底部的内容。

显然，这很不理想。为了解决这个问题，我们的第一个任务是确定程式在显示区域内能输出多少内容。

显示区域的大小

如果您使用过现有的 Windows 应用程式，可能会发现视窗的尺寸变化极大。视窗最大化时（假定视窗只有标题列并且没有功能表），显示区域几乎占据了整个萤幕。这一最大化了的显示区域的尺寸可以通过以 SM_CXFULLSCREEN 和 SM_CYFULLSCREEN 为参数呼叫 GetSystemMetrics 来获得。视窗的最小尺寸可以很小，有时甚至不存在，更不用说显示区域了。

在最近一章，我们使用 GetClientRect 函式来取得显示区域的大小。使用这个函式没有什么不好，但是在您每次要使用资讯时就去呼叫它一遍是没有效率的。确定视窗显示区域大小的更好方法是在视窗讯息处理程式中处理 WM_SIZE 讯息。在视窗大小改变时，Windows 给视窗讯息处理程式发送一个 WM_SIZE 讯息。传给视窗讯息处理程式的 lParam 参数的低字组中包含显示区域的宽度，高字组中包含显示区域的高度。要保存这些尺寸，需要在视窗讯息处理程式中定义两个静态变数：

```
static int cxClient, cyClient ;
```

与 cxChar 和 cyChar 相似，这两个变数在视窗讯息处理程式内定义为静态变数，因为在以後处理其他讯息时会用到它们。处理 WM_SIZE 的方法如下：

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

实际上您会在每个 Windows 程式中看到类似的程式码。LOWORD 和 HIWORD 巨集在 Windows 表头档案 WINDEF.H 中定义。这些巨集的定义看起来像这样：

```
#define LOWORD(l) ((WORD)(l))
#define HIWORD(l) ((WORD)(((DWORD)(l) >> 16) & 0xFFFF))
```

这两个巨集传回 WORD 值（16 位元的无正负号整数，范围从 0 到 0xFFFF）。一般，将这些值保存在 32 位元有号整数中。这就不会牵扯到任何转换问题，并使得这些值在以後需要的任何计算中易於使用。

在许多 Windows 程式中，WM_SIZE 讯息必然跟著一个 WM_PAINT 讯息。为什么呢？因为在我们定义视窗类别时指定视窗类别样式为：

```
CS_HREDRAW | CS_VREDRAW
```

这种视窗类别样式告诉 Windows，如果水平或者垂直大小发生改变， 则强制更新显示区域。

用如下公式计算可以在显示区域内显示的文字的总行数：

```
cyClient / cyChar
```

如果显示区域的高度太小以至无法显示一个完整的字元，这个公式的结果可以为 0。类似地，在显示区域的水平方向可以显示的小写字元的近似数目为：

```
cxClient / cxChar
```

如果在处理 WM_CREATE 讯息处理期间取得 cxChar 和 cyChar，则不用担心在这两个计算公式中会出现被 0 除的情况。在 WinMain 呼叫 CreateWindow 时，视窗讯息处理程式接收一个 WM_CREATE 讯息。在 WinMain 呼叫 ShowWindow 之後接收到第一个 WM_CREATE 讯息，此时 cxChar 和 cyChar 已经被赋予正的非零值了。

如果显示区域的大小不足以容纳所有的内容，那么，知道视窗显示区域的大小只是为使用者提供了在显示区域内卷动文字的第一步。如果您对其他有类似需求的 Windows 应用程式很熟悉，就很可能知道，这种情况下，我们需要使用「卷动列」。

卷动列

卷动列是图形使用者介面中最好的功能之一，它很容易使用，而且提供了很好的视觉回馈效果。您可以使用卷动列显示任何东西--无论是文字、图形、表格、资料库记录、图像或是网页，只要它所需的空间超出了视窗的显示区域所能提供的空间，就可以使用卷动列。

卷动列既有垂直方向的（供上下移动），也有水平方向的（供左右移动）。使用者可以使用滑鼠在卷动列两端的箭头上或者在箭头之间的区域中点一下，这时，「卷动方块」在卷动列内的移动位置与所显示的资讯在整个文件中的近似相关位置成比例。使用者也可以用滑鼠拖动卷动方块到特定的位置。图 4-5 显示了垂直卷动列的建议用法。

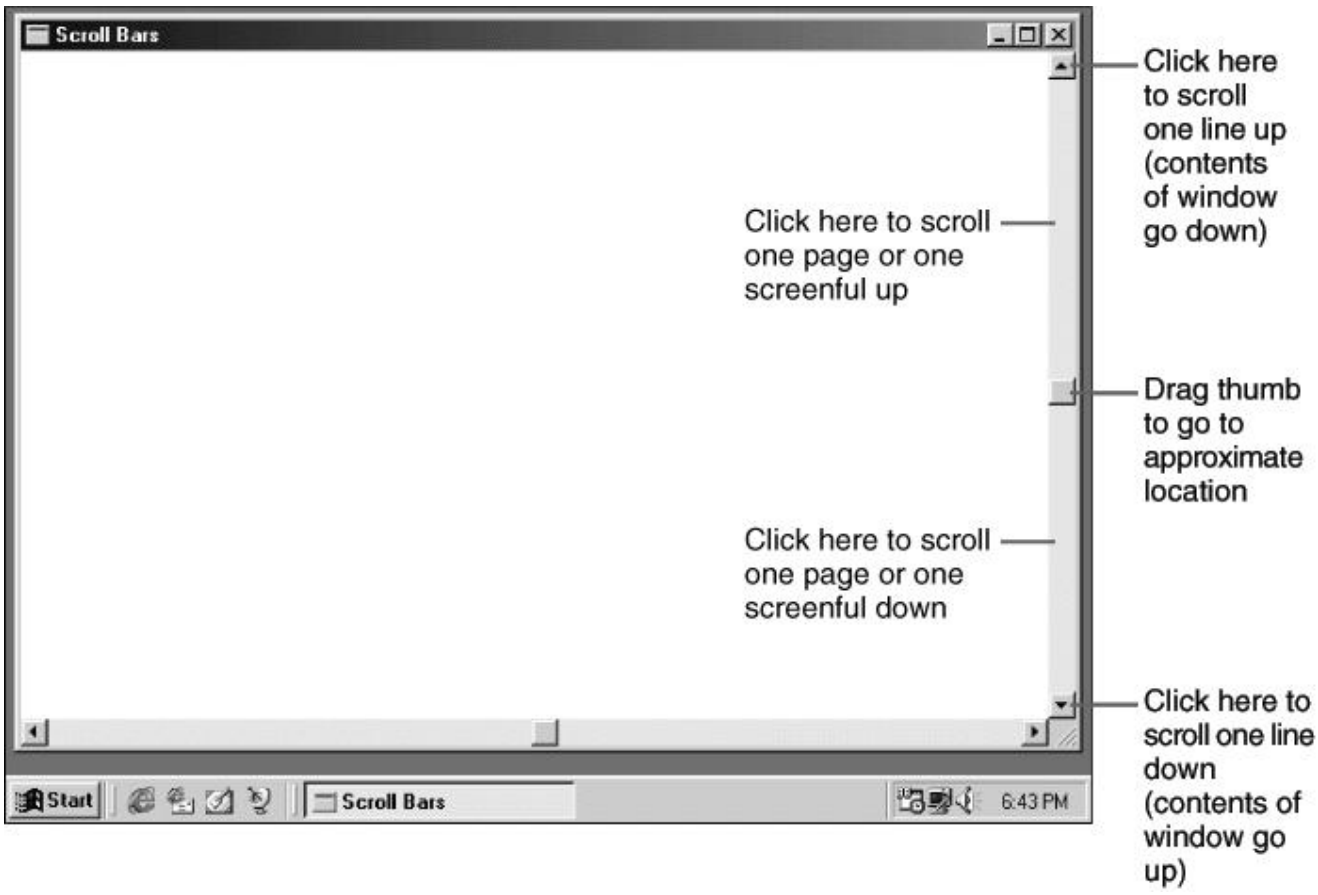

图 4-5 垂直卷动列

有时，程式写作者对卷动概念很难理解，因为他们的观点与使用者的观点不同：使用者向下卷动是想看到文件较下面的部分；但是，程式实际上是将文件相对於显示视窗向上移动。Windows 文件和表头档案识别字是依据使用者的观点：向上卷动意味著朝文件的开头移动；向下卷动意味著朝文件尾部移动。

很容易在应用程式中包含水平或者垂直的卷动列，程式写作者只需要在 CreateWindow 的第三个参数中包括视窗样式（WS）识别字 WS_VSCROLL（垂直卷动）和/或 WS_HSCROLL（水平卷动）即可。这些卷动列通常放在视窗的右部和底部，伸展为显示区域的整个长度或宽度。显示区域不包含卷动列所占据的空间。对於特定的显示驱动程式和显示解析度，垂直卷动列的宽度和水平卷动列的高度是恒定的。如果需要这些值，可以使用 GetSystemMetrics 呼叫来取得（如前面的程式那样）。

Windows 负责处理对卷动列的所有滑鼠操作，但是，视窗卷动列没有自动的键盘介面。如果想用游标键来完成卷动功能，则必须提供这方面的程式码（我们将在下一章另一个版本的 SYSMETS 程式中做到这一点）。

卷动列的范围和位置

每个卷动列均有一个相关的「范围」（这是一对整数，分别代表最小值和

最大值）和「位置」（它是卷动方块在此范围内的位置）。当卷动方块在卷动列的顶部（或左部）时，卷动方块的位置是范围的最小值；在卷动列的底部（或右部）时，卷动方块的位置是范围的最大值。

在内定情况下，卷动列的范围是从 0（顶部或左部）至 100（底部或右部），但将范围改变为更方便於程式的数值也是很容易的：

```
SetScrollRange (hwnd, iBar, iMin, iMax, bRedraw) ;
```

参数 iBar 为 SB_VERT 或者 SB_HORZ，iMin 和 iMax 分别是范围的最小值和最大值。如果想要 Windows 根据新范围重画卷动列，则设置 bRedraw 为 TRUE（如果在呼叫 SetScrollRange 後，呼叫了影响卷动列位置的其他函式，则应该将 bRedraw 设定为 FALSE 以避免过多的重画）。

卷动方块的位置总是离散的整数值。例如，范围为 0 至 4 的卷动列具有 5 个卷动方块位置，如图 4-6 所示。

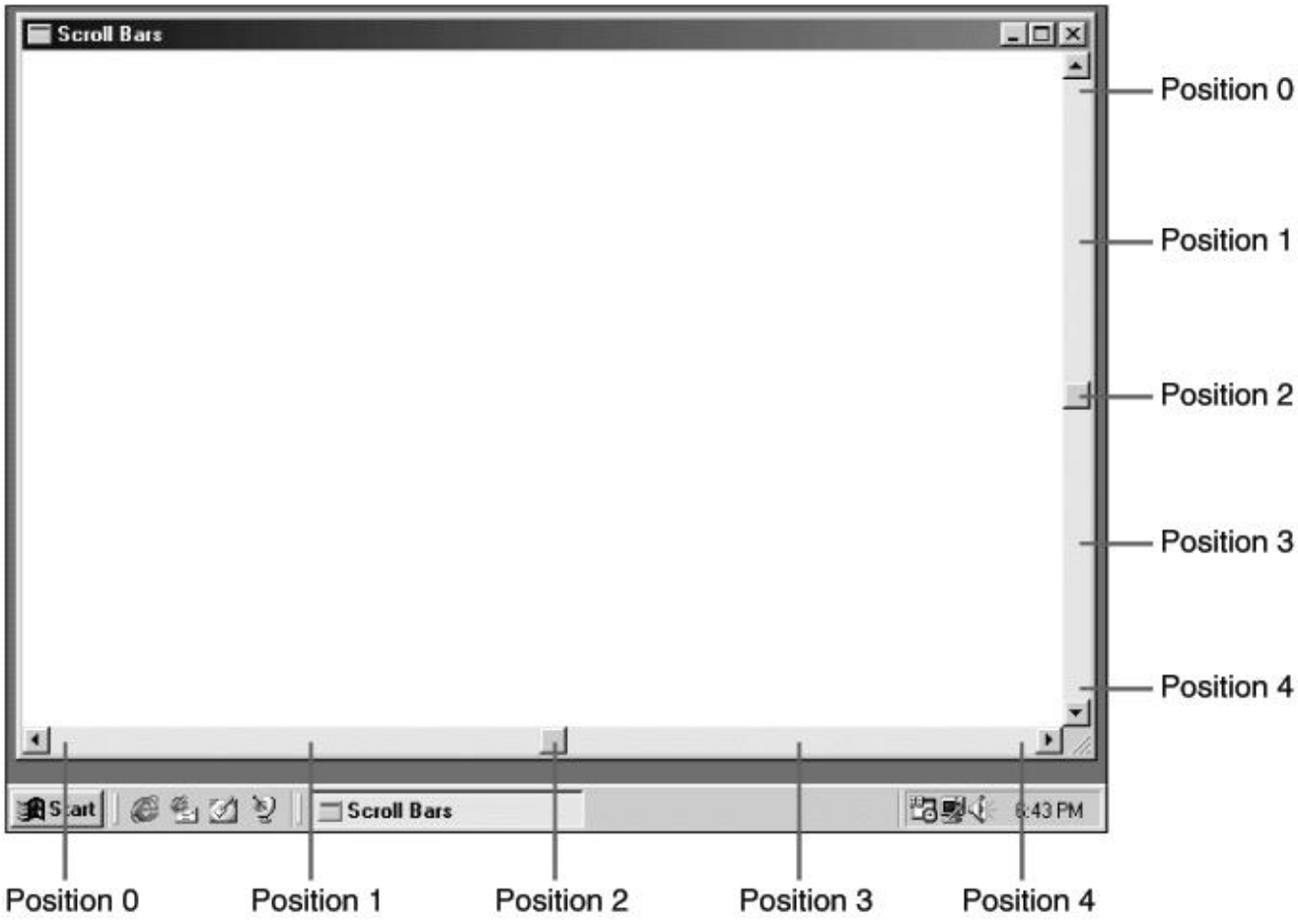

图 4-6 具有 5 个卷动方块位置的卷动列

您可以使用 SetScrollPos 在卷动列范围内设置新的卷动方块位置：

```
SetScrollPos (hwnd, iBar, iPos, bRedraw) ;
```

参数 iPos 是新位置，它必须在 iMin 至 iMax 的范围内。Windows 提供了类似的函式（GetScrollRange 和 GetScrollPos）来取得卷动列的目前范围和位置。

在程式内使用卷动列时，程式写作者与 Windows 共同负责维护卷动列以及

更新卷动方块的位置。下面是 Windows 对卷动列的处理：

- 处理所有卷动列滑鼠事件
- 当使用者在卷动列内单击滑鼠时，提供一种「反相显示」的闪烁
- 当使用者在卷动列内拖动卷动方块时，移动卷动方块
- 为包含卷动列视窗的视窗讯息处理程式发送卷动列讯息

以下是程式写作者应该完成的工作：

- 初始化卷动列的范围和位置
- 处理视窗讯息处理程式的卷动列讯息
- 更新卷动列内卷动方块的位置
- 更改显示区域的内容以回应对卷动列的更改

像生活中的大多数事情一样，在我们看一些程式码时这些会显得更加有意义。

卷动列讯息

在用滑鼠单击卷动列或者拖动卷动方块时，Windows 给视窗讯息处理程式发送 WM_VSCROLL（供上下移动）和 WM_HSCROLL（供左右移动）讯息。在卷动列上的每个滑鼠动作都至少产生两个讯息，一条在按下滑鼠按钮时产生，一条在释放按钮时产生。

和所有的讯息一样， WM_VSCROLL 和 WM_HSCROLL 也带有 wParam 和 lParam 讯息参数。对於来自作为视窗的一部分而建立的卷动列讯息，您可以忽略 lParam；它只用于作为子视窗而建立的卷动列（通常在对话方块内）。

wParam 讯息参数被分为一个低字组和一个高字组。wParam 的低字组是一个数值，它指出了滑鼠对卷动列进行的操作。这个数值被看作一个「通知码」。通知码的值由以 SB（代表「scroll bar（卷动列）」）开头的识别字定义。以下是在 WINUSER.H 中定义的通知码：

```
#define SB_LINEUP           0
#define SB_LINELEFT         0
#define SB_LINEDOWN         1
#define SB_LINERIGHT        1
#define SB_PAGEUP           2
#define SB_PAGELEFT         2
#define SB_PAGEDOWN         3
#define SB_PAGERIGHT        3
#define SB_THUMBPOSITION    4
#define SB_THUMBTRACK       5
#define SB_TOP              6
#define SB_LEFT             6
#define SB_BOTTOM           7
```

```
#define SB_RIGHT        7
#define SB_ENDSCROLL      8
```

包含 LEFT 和 RIGHT 的识别字用於水平卷动列，包含 UP、DOWN、TOP 和 BOTTOM 的识别字用於垂直卷动列。滑鼠在卷动列的不同区域单击所产生的通知码如图 4-7 所示。

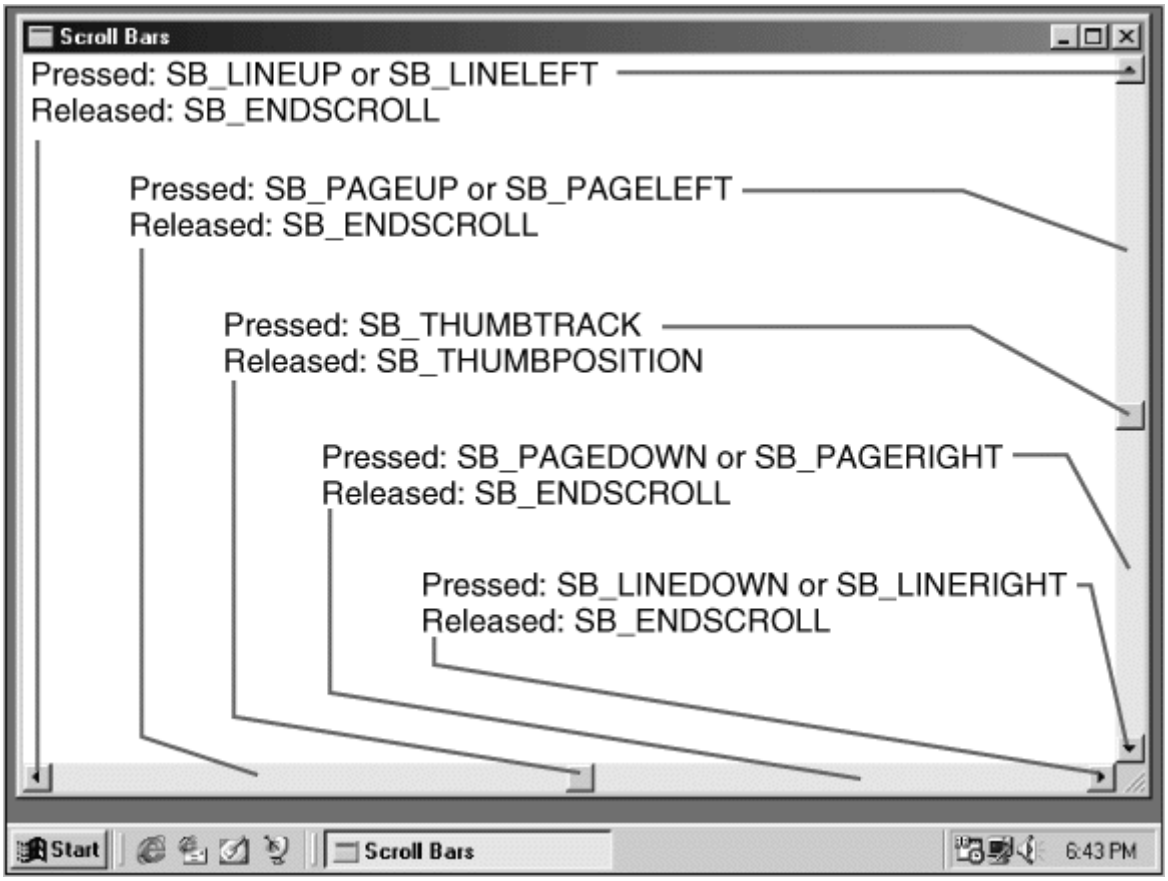

图 4-7 用於卷动列讯息的 wParam 值的识别字

如果在卷动列的各个部位按住滑鼠键，程式就能收到多个卷动列讯息。当释放滑鼠键後，程式会收到一个带有 SB_ENDSCROLL 通知码的讯息。一般可以忽略这个讯息，Windows 不会去改变卷动方块的位置，而您可以在程式中呼叫 SetScrollPos 来改变卷动方块的位置。

当把滑鼠的游标放在卷动方块上并按住滑鼠键时，您就可以移动卷动方块。这样就产生了带有 SB_THUMBTRACK 和 SB_THUMBPOSITION 通知码的卷动列讯息。在 wParam 的低字组是 SB_THUMBTRACK 时，wParam 的高字组是使用者在拖动卷动方块时的目前位置。该位置位於卷动列范围的最小值和最大值之间。在 wParam 的低字组是 SB_THUMBPOSITION 时，wParam 的高字组是使用者释放滑鼠键後卷动方块的最终位置。对於其他的卷动列操作，wParam 的高字组应该被忽略。

为了给使用者提供回馈，Windows 在您用滑鼠拖动卷动方块时移动它，同时您的程式会收到 SB_THUMBTRACK 讯息。然而，如果不通过呼叫 SetScrollPos 来处理 SB_THUMBTRACK 或 SB_THUMBPOSITION 讯息，在使用者释放滑鼠键後，卷动

方块会迅速跳回原来的位置。

程式能够处理 SB_THUMBTRACK 或 SB_THUMBPOSITION 讯息，但一般不同时处理两者。如果处理 SB_THUMBTRACK 讯息，在使用者拖动卷动方块时您需要移动显示区域的内容。而如果处理 SB_THUMBPOSITION 讯息，则只需在使用者停止拖动卷动方块时移动显示区域的内容。处理 SB_THUMBTRACK 讯息更好一些（但更困难），对於某些型态的资料，您的程式可能很难跟上产生的讯息。

WINUSER.H 表头档案还包括 SB_TOP、SB_BOTTOM、SB_LEFT 和 SB_RIGHT 通知码，指出卷动列已经被移到了它的最小或最大位置。然而，对於作为应用程式视窗一部分而建立的卷动列来说，永远不会接收到这些通知码。

在卷动列范围使用 32 位元的值也是有效的，尽管这不常见。然而，wParam 的高字组只有 16 位元的大小，它不能适当地指出 SB_THUMBTRACK 和 SB_THUMBPOSITION 操作的位置。在这种情况下，需要使用 GetScrollInfo 函式（在下面描述）来得到资讯。

在 SYSMETS 中加入卷动功能

前面的说明已经很详尽了，现在，要将那些东西动手做做看了。让我们开始时简单些，从垂直卷动著手，因为我们实在太需要垂直卷动了，而暂时还可以不用水平卷动。SYSMET2 如程式 4-3 所示。这个程式可能是卷动列的最简单的应用。

程式 4-3　SYSMETS2.C

```
/*------------------------------------------------------------------
      SYSMETS2.C -- System Metrics Display Program No. 2
              (c) Charles Petzold, 1998
 ------------------------------------------------------------------*/
#include <windows.h>
#include "sysmets.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
      static TCHAR szAppName[] = TEXT ("SysMets2") ;
      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 ("This program requires Windows NT!"),
            szAppName, MB_ICONERROR) ;
      return 0 ;
    }

     hwnd = CreateWindow (szAppName, TEXT ("Get System Metrics No. 2"),
                WS_OVERLAPPEDWINDOW | WS_VSCROLL,
                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 int  cxChar, cxCaps, cyChar, cyClient, iVscrollPos ;
     HDC       hdc ;
     int       i, y ;
     PAINTSTRUCT ps ;
     TCHAR     szBuffer[10] ;
     TEXTMETRIC  tm ;
     switch (message)
    {
case WM_CREATE:
     hdc = GetDC (hwnd) ;
     GetTextMetrics (hdc, &tm) ;
     cxChar = tm.tmAveCharWidth ;
     cxCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2 ;
     cyChar = tm.tmHeight + tm.tmExternalLeading ;

     ReleaseDC (hwnd, hdc) ;
     SetScrollRange (hwnd, SB_VERT, 0, NUMLINES - 1, FALSE) ;
     SetScrollPos   (hwnd, SB_VERT, iVscrollPos, TRUE) ;
```

```
          return 0 ;

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

     case WM_VSCROLL:
          switch (LOWORD (wParam))
        {
     case SB_LINEUP:
          iVscrollPos -= 1 ;
          break ;

     case SB_LINEDOWN:
          iVscrollPos += 1 ;
          break ;

     case SB_PAGEUP:
          iVscrollPos -= cyClient / cyChar ;
          break ;

     case SB_PAGEDOWN:
          iVscrollPos += cyClient / cyChar ;
          break ;

     case SB_THUMBPOSITION:
          iVscrollPos = HIWORD (wParam) ;
          break ;

     default :
          break ;
        }

     iVscrollPos = max (0, min (iVscrollPos, NUMLINES - 1)) ;
     if (iVscrollPos != GetScrollPos (hwnd, SB_VERT))
        {
          SetScrollPos (hwnd, SB_VERT, iVscrollPos, TRUE) ;
          InvalidateRect (hwnd, NULL, TRUE) ;
        }
          return 0 ;
     case WM_PAINT:
          hdc = BeginPaint (hwnd, &ps) ;
          for (i = 0 ; i < NUMLINES ; i++)
          {
                y = cyChar * (i - iVscrollPos) ;
                TextOut (hdc, 0, y,
                      sysmetrics[i].szLabel,
                      lstrlen (sysmetrics[i].szLabel)) ;
```

```
                TextOut (hdc, 22 * cxCaps, y,
                      sysmetrics[i].szDesc,
                      lstrlen (sysmetrics[i].szDesc)) ;

                SetTextAlign (hdc, TA_RIGHT | TA_TOP) ;
                TextOut (hdc, 22 * cxCaps + 40 * cxChar, y, szBuffer,
                      wsprintf (szBuffer, TEXT ("%5d"),
                                  GetSystemMetrics (sysmetrics[i].iIndex))) ;
                SetTextAlign (hdc, TA_LEFT | TA_TOP) ;
        }
           EndPaint (hwnd, &ps) ;
           return 0 ;

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

新的 CreateWindow 呼叫在第三个参数中包含了 WS_VSCROLL 视窗样式，从而在视窗中加入了垂直卷动列，其视窗样式为：

```
WS_OVERLAPPEDWINDOW | WS_VSCROLL
```

WndProc 视窗讯息处理程式在处理 WM_CREATE 讯息时增加了两条叙述，以设置垂直卷动列的范围和初始位置：

```
SetScrollRange (hwnd, SB_VERT, 0, NUMLINES - 1, FALSE) ;
SetScrollPos (hwnd, SB_VERT, iVscrollPos, TRUE) ;
```

sysmetrics 结构具有 NUMLINES 行文字，所以卷动列范围被设定为 0 至 NUMLINES-1。卷动列的每个位置对应于在显示区域顶部显示的一个文字行。如果卷动方块的位置为 0，则第一行会被放置在显示区域的顶部。如果位置大於 0，其他行就会出现在显示区域的顶部。当位置为 NUMLINES-1 时，则最後一行文字出现在显示区域的顶部。

为了有助於处理 WM_VSCROLL 讯息，在视窗讯息处理程式中定义了一个静态变数 iVscrollPos，这一变数是卷动列内卷动方块的目前位置。对於 SB_LINEUP 和 SB_LINEDOWN，只需要将卷动方块调整一个单位的位置。对於 SB_PAGEUP 和 SB_PAGEDOWN，我们想移动一整面的内容，或者移动 cyClient /cyChar 个单位的位置。对於 SB_THUMBPOSITION，新的卷动方块位置是 wParam 的高字组。SB_ENDSCROLL 和 SB_THUMBTRACK 讯息被忽略。

在程式依据收到的 WM_VSCROLL 讯息计算出新的 iVscrollPos 值後，用 min 和 max 巨集来调整 iVscrollPos，以确保它在最大值与最小值之间。程式然後将 iVscrollPos 与呼叫 GetScrollPos 取得的先前位置相比较，如果卷动位置发生

了变化，则使用 SetScrollPos 来进行更新，并且呼叫 InvalidateRect 使整个视窗无效。

InvalidateRect 呼叫产生一个 WM_PAINT 讯息。SYSMETS1 在处理 WM_PAINT 讯息时，每一行的 y 座标计算公式为：

```
cyChar * i
```

在 SYSMETS2 中，计算公式为：

```
cyChar * (i - iVscrollPos)
```

回圈仍然显示 NUMLINES 行文字，但是对於非零值的 iVscrollPos 是负数。程式实际上在显示区域以外显示这些文字行。当然，Windows 不会显示这些行，因此萤幕显得乾净和漂亮。

前面说过，我们一开始不想弄得太复杂，这样的程式码很浪费，效率很低。下面我们对此加以修改，但是先要考虑在 WM_VSCROLL 讯息之後更新显示区域的方法。

绘图程式的组织

在处理完卷动列讯息後，SYSMETS2 不更新显示区域，相反，它呼叫 InvalidateRect 使显示区域失效。这导致 Windows 将一个 WM_PAINT 讯息放入讯息伫列中。

最好能使 Windows 程式在回应 WM_PAINT 讯息时完成所有的显示区域绘制功能。因为程式必须在一接收到 WM_PAINT 讯息时就更新整个显示区域，如果在程式的其他部分也绘制的话，将很可能使程式码重复。

首先，您可能对这种拐弯抹角的方式感到厌烦。在 Windows 的早期，因为这种方式与文字模式的程式设计差别太大，程式写作者感到这种概念很难理解。并且，程式要不断地通过马上绘制画面来回应键盘和滑鼠。这样做既方便又有效，但是在很多情况下，这完全不必要。当您掌握了在回应 WM_PAINT 讯息时积累绘制显示区域所需要的全部资讯的原则之後，会对这种结果感到满意的。

如同 SYSMETS2 示范的，程式仍然需要在处理非 WM_PAINT 讯息时更新特定的显示区域，使用 InvalidateRect 就很方便，您可以用它使显示区域内的特定矩形或者整个显示区域失效。

只将视窗显示区域标记为无效以产生 WM_PAINT 讯息，对於某些应用程式来说也许不是完全令人满意的选择。在呼叫 InvalidateRect 之後，Windows 将 WM_PAINT 讯息放入讯息伫列中，最後由视窗讯息处理程式处理它。然而，Windows 将 WM_PAINT 讯息当成低优先顺序讯息，如果系统有许多其他的动作正在发生，那么也许会让您等待一会儿工夫。这时，当对话方块消失时，将会出现一些空白的「洞」，程式仍然等待更新它的视窗。

如果您希望立即更新无效区域，可以在呼叫 InvalidateRect 之後呼叫 UpdateWindow：

```
UpdateWindow (hwnd) ;
```

如果显示区域的任一部分无效，则 UpdateWindow 将导致 Windows 用 WM_PAINT 讯息呼叫视窗讯息处理程式（如果整个显示区域有效，则不呼叫视窗讯息处理程式）。这一 WM_PAINT 讯息不进入讯息伫列，直接由 Windows 呼叫视窗讯息处理程式。视窗讯息处理程式完成更新後立即退出，Windows 将控制传回给程式中 UpdateWindow 呼叫之後的叙述。

您可能注意到，UpdateWindow 与 WinMain 中用来产生第一个 WM_PAINT 讯息的函式相同。最初建立视窗时，整个显示区域内容变为无效，UpdateWindow 指示视窗讯息处理程式绘制显示区域。

建立更好的滚动

SYSMETS2 动作良好，但它只是模仿其他程式中的卷动列，并且效率很低。很快我将示范一个新的版本，改进它的不足。也许最有趣的是这个新版本不使用目前所讨论的四个卷动列函式。相反，它将使用 Win32 API 中才有的新函式。

卷动列资讯函式

卷动列文件（在/Platform SDK/User Interface Services/Controls/Scroll Bars 中）指出 SetScrollRange、SetScrollPos、GetScrollRange 和 GetScrollPos 函式是「过时的」，但这并不完全正确。这些函式在 Windows 1.0 中就出现了，在 Win32 API 中升级以处理 32 位元参数。它们仍然具有良好的功能。而且，它们不与 Windows 程式设计中新函式相冲突，这就是我在此书中仍使用它们的原因。

Win32 API 介绍的两个卷动列函式称作 SetScrollInfo 和 GetScrollInfo。这些函式可以完成以前函式的全部功能，并增加了两个新特性。

第一个功能涉及卷动方块的大小。您可能注意到，卷动方块大小在 SYSMETS2 程式中是固定的。然而，在您可能使用到的一些 Windows 应用程式中，卷动方块大小与在视窗中显示的文件大小成比例。显示的大小称作「页面大小」。演算法为：

$$\frac{\text{捲動方塊大小}}{\text{滾動長度}} \approx \frac{\text{頁面大小}}{\text{範圍}} \approx \frac{\text{顯示的文件數量}}{\text{文件的總大小}}$$

可以使用 SetScrollInfo 来设置页面大小（从而设置了卷动方块的大小），如将要看到的 SYSMETS3 程式所示。

GetScrollInfo 函式增加了第二个重要的功能，或者说它改进了目前 API 的不足。假设您要使用 65,536 或更大单位的范围，这在 16 位元 Windows 中是不可能的。当然在 Win32 中，函式被定义为可接受 32 位元参数，因此是没有问题的。（记住如果使用这样大的范围，卷动方块的实际物理位置数仍然由卷动列的图素大小限制）。然而，当使用 SB_THUMBTRACK 或 SB_THUMBPOSITION 通知码得到 WM_VSCROLL 或 WM_HSCROLL 讯息时，只提供了 16 位元资料来指出卷动方块的目前位置。通过 GetScrollInfo 函式可以取得真实的 32 位元值。

SetScrollInfo 和 GetScrollInfo 函式的语法是

```
SetScrollInfo (hwnd, iBar, &si, bRedraw) ;
GetScrollInfo (hwnd, iBar, &si) ;
```

像在其他卷动列函式中那样，iBar 参数是 SB_VERT 或 SB_HORZ，它还可以是用於卷动列控制的 SB_CTL。SetScrollInfo 的最後一个参数可以是 TRUE 或 FALSE，指出了是否要 Windows 重新绘制计算了新资讯後的卷动列。

两个函式的第三个参数是 SCROLLINFO 结构，定义为：

```
typedef struct tagSCROLLINFO
{
      UINT cbSize ;     // set to sizeof (SCROLLINFO)
      UINT fMask ;      // values to set or get
      int  nMin ;     // minimum range value
      int  nMax ;       // maximum range value
      UINT nPage ;      // page size
      int  nPos ;       // current position
      int  nTrackPos ;// current tracking position
}
SCROLLINFO, * PSCROLLINFO ;
```

在程式中，可以定义如下的 SCROLLINFO 结构型态：

```
SCROLLINFO si ;
```

在呼叫 SetScrollInfo 或 GetScrollInfo 之前，必须将 cbSize 栏位设定为结构的大小：

```
si.cbSize = sizeof (si) ;
```

或

```
si.cbSize = sizeof (SCROLLINFO) ;
```

逐渐熟悉 Windows 後，您就会发现另外几个结构像这个结构一样，第一个栏位指出了结构大小。这个栏位使将来的 Windows 版本可以扩充结构并添加新的功能，并且仍然与以前编译的版本相容。

把 fMask 栏位设定为一个以上以 SIF 字首开头的旗标，并且可以使用 C 的位元操作 OR 运算子(|)组合这些旗标。

SetScrollInfo 函式使用 SIF_RANGE 旗标时，必须把 nMin 和 nMax 栏位设定为所需的卷动列范围。GetScrollInfo 函式使用 SIF_RANGE 旗标时，应把 nMin

和 nMax 栏位设定为从函式传回的目前范围。

SIF_POS 旗标也一样。当通过 SetScrollInfo 使用它时，必须把结构的 nPos 栏位设定为所需的位置。可以通过 GetScrollInfo 使用 SIF_POS 旗标来取得目前位置。

使用 SIF_PAGE 旗标能够取得页面大小。用 SetScrollInfo 函式把 nPage 设定为所需的页面大小。GetScrollInfo 使用 SIF_PAGE 旗标可以取得目前页面的大小。如果不想得到比例化的卷动列，就不要使用该旗标。

当处理带有 SB_THUMBTRACK 或 SB_THUMBPOSITION 通知码的 WM_VSCROLL 或 WM_HSCROLL 讯息时，通过 GetScrollInfo 只使用 SIF_TRACKPOS 旗标。从函式的传回中，SCROLLINFO 结构的 nTrackPos 栏位将指出目前的 32 位元的卷动方块位置。

在 SetScrollInfo 函式中仅使用 SIF_DISABLENOSCROLL 旗标。如果指定了此旗标，而且新的卷动列参数使卷动列消失，则该卷动列就不能使用了（下面会有更多的解释）。

SIF_ALL 旗标是 SIF_RANGE、SIF_POS、SIF_PAGE 和 SIF_TRACKPOS 的组合。在 WM_SIZE 讯息处理期间设置卷动列参数时，这是很方便的（在 SetScrollInfo 函式中指定 SIF_TRACKPOS 後，它会被忽略）。这在处理卷动列讯息时也是很方便的。

卷动范围

在 SYSMETS2 中，卷动范围设置最小为 0，最大为 NUMLINES-1。当卷动列位置是 0 时，第一行资讯显示在显示区域的顶部；当卷动列的位置是 NUMLINES-1 时，最後一行显示在显示区域的顶部，并且看不见其他行。

可以说 SYSMETS2 卷动范围太大。事实上只需把资讯最後一行显示在显示区域的底部而不是顶部即可。我们可以对 SYSMETS2 作出一些修改以达到此点。当处理 WM_CREATE 讯息时不设置卷动列范围，而是等到接收到 WM_SIZE 讯息後再做此工作：

```
iVscrollMax = max (0, NUMLINES - cyClient / cyChar) ;
SetScrollRange (hwnd, SB_VERT, 0, iVscrollMax, TRUE) ;
```

假定 NUMLINES 等於 75，并假定特定视窗大小是：50（cyChar 除以 cyClient）。换句话说，我们有 75 行资讯但只有 50 行可以显示在显示区域中。使用上面的两行程式码，把范围设置最小为 0，最大为 25。当卷动列位置等於 0 时，程式显示 0 到 49 行。当卷动列位置等於 1 时，程式显示 1 到 50 行；并且当卷动列位置等於 25（最大值）时，程式显示 25 到 74 行。很明显需要对程式的其他部

分做出修改，但这是可行的。

新卷动列函式的一个好的功能是当使用与卷动列范围一样大的页面时，它已经为您做掉了一大堆杂事。可以像下面的程式码一样使用 SCROLLINFO 结构和 SetScrollInfo：

```
si.cbSize   = sizeof (SCROLLINFO) ;
si.cbMask   = SIF_RANGE | SIF_PAGE ;
si.nMin     = 0 ;
si.nMax     = NUMLINES - 1 ;
si.nPage    = cyClient / cyChar ;
SetScrollInfo (hwnd, SB_VERT, &si, TRUE) ;
```

这样做之後，Windows 会把最大的卷动列位置限制为 si.nMax - si.nPage +1 而不是 si.nMax。像前面那样做出假设：NUMLINES 等於 75 （所以 si.nMax 等於 74），si.nPage 等於 50。这意味著最大的卷动列位置限制为 74 - 50 + 1，即 25。这正是我们想要的。

当页面大小与卷动列范围一样大时，会发生什么情况呢？在这个例子中，就是 nPage 等於 75 或更大的情况。Windows 通常隐藏卷动列，因为它并不需要。如果不想隐藏卷动列，可在呼叫 SetScrollInfo 时使用 SIF_DISABLENOSCROLL，Windows 只是让那个卷动列不能被使用，而不隐藏它。

新 SYSMETS

SYSMETS3——此章中最後的 SYSMETS 程式版本——显示在程式 4-4 中。此版本使用 SetScrollInfo 和 GetScrollInfo 函式，添加左右卷动的水平卷动列，并能更有效地重画显示区域。

程式 4-4　SYSMETS3

```
SYSMETS3.C
/*-------------------------------------------------------------------
   SYSMETS3.C -- System Metrics Display Program No. 3
                 (c) Charles Petzold, 1998
  ------------------------------------------------------------------*/
#include <windows.h>
#include "sysmets.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
           PSTR szCmdLine, int iCmdShow)
{
      static TCHAR szAppName[] = TEXT ("SysMets3") ;
      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 ("Get System Metrics No. 3"),
                WS_OVERLAPPEDWINDOW | WS_VSCROLL | WS_HSCROLL,
                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 int  cxChar, cxCaps, cyChar, cxClient, cyClient, iMaxWidth ;
     HDC   hdc ;
     int   i, x, y, iVertPos, iHorzPos, iPaintBeg, iPaintEnd ;
     PAINTSTRUCT ps ;
     SCROLLINFO  si ;
     TCHAR      szBuffer[10] ;
     TEXTMETRIC  tm ;

     switch (message)
    {
     case WM_CREATE:
           hdc = GetDC (hwnd) ;
```

```
      GetTextMetrics (hdc, &tm) ;
      cxChar = tm.tmAveCharWidth ;
      cxCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2 ;
      cyChar = tm.tmHeight + tm.tmExternalLeading ;

      ReleaseDC (hwnd, hdc) ;
            // Save the width of the three columns
      iMaxWidth = 40 * cxChar + 22 * cxCaps ;
      return 0 ;

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

            // Set vertical scroll bar range and page size
      si.cbSize   = sizeof (si) ;
      si.fMask    = SIF_RANGE | SIF_PAGE ;
      si.nMin     = 0 ;
      si.nMax     = NUMLINES - 1 ;
      si.nPage    = cyClient / cyChar ;
      SetScrollInfo (hwnd, SB_VERT, &si, TRUE) ;
            // Set horizontal scroll bar range and page size
      si.cbSize   = sizeof (si) ;
      si.fMask    = SIF_RANGE | SIF_PAGE ;
      si.nMin     = 0 ;
      si.nMax     = 2 + iMaxWidth / cxChar ;
      si.nPage    = cxClient / cxChar ;
      SetScrollInfo (hwnd, SB_HORZ, &si, TRUE) ;
      return 0 ;

case WM_VSCROLL:
            // Get all the vertical scroll bar information
      si.cbSize   = sizeof (si) ;
      si.fMask    = SIF_ALL ;
      GetScrollInfo (hwnd, SB_VERT, &si) ;
            // Save the position for comparison later on
      iVertPos = si.nPos ;
      switch (LOWORD (wParam))
      {
      case  SB_TOP:
            si.nPos     = si.nMin ;
            break ;

      case  SB_BOTTOM:
                  si.nPos     = si.nMax ;
            break ;

      case SB_LINEUP:
```

```
            si.nPos -   = 1 ;
            break ;

      case  SB_LINEDOWN:
            si.nPos += 1 ;
            break ;

      case  SB_PAGEUP:
            si.nPos -= si.nPage ;
            break ;

      case  SB_PAGEDOWN:
            si.nPos += si.nPage ;
            break ;

      case  SB_THUMBTRACK:
            si.nPos = si.nTrackPos ;
            break ;

      default:
      break ;
      }
            // Set the position and then retrieve it.  Due to adjustments
            //  by Windows it may not be the same as the value set.

      si.fMask = SIF_POS ;
      SetScrollInfo (hwnd, SB_VERT, &si, TRUE) ;
      GetScrollInfo (hwnd, SB_VERT, &si) ;

            // If the position has changed, scroll the window and update it
      if (si.nPos != iVertPos)
      {
            ScrollWindow (   hwnd, 0, cyChar * (iVertPos - si.nPos),
                                    NULL, NULL) ;
            UpdateWindow (hwnd) ;
      }
      return 0 ;
      case WM_HSCROLL:
                  // Get all the vertical scroll bar information
            si.cbSize = sizeof (si) ;
            si.fMask  = SIF_ALL ;

                  // Save the position for comparison later on
            GetScrollInfo (hwnd, SB_HORZ, &si) ;
            iHorzPos = si.nPos ;

            switch (LOWORD (wParam))
            {
```

```
     case  SB_LINELEFT:
           si.nPos -= 1 ;
           break ;

     case  SB_LINERIGHT:
           si.nPos += 1 ;
           break ;

     case  SB_PAGELEFT:
           si.nPos -= si.nPage ;
           break ;

     case  SB_PAGERIGHT:
           si.nPos += si.nPage ;
           break ;

     case  SB_THUMBPOSITION:
           si.nPos = si.nTrackPos ;
           break ;

           default :
           break ;
     }
           // Set the position and then retrieve it.  Due to adjustments
           //   by Windows it may not be the same as the value set.

           si.fMask = SIF_POS ;
           SetScrollInfo (hwnd, SB_HORZ, &si, TRUE) ;
           GetScrollInfo (hwnd, SB_HORZ, &si) ;

           // If the position has changed, scroll the window

     if (si.nPos != iHorzPos)
     {
           ScrollWindow (   hwnd, cxChar * (iHorzPos - si.nPos), 0,
                            NULL, NULL) ;
     }
           return 0 ;
     case WM_PAINT :
           hdc = BeginPaint (hwnd, &ps) ;
           // Get vertical scroll bar position
     si.cbSize = sizeof (si) ;
     si.fMask  = SIF_POS ;
     GetScrollInfo (hwnd, SB_VERT, &si) ;
     iVertPos = si.nPos ;

           // Get horizontal scroll bar position
     GetScrollInfo (hwnd, SB_HORZ, &si) ;
```

```
     iHorzPos = si.nPos ;
          // Find painting limits
     iPaintBeg = max (0, iVertPos + ps.rcPaint.top / cyChar) ;
     iPaintEnd = min (NUMLINES - 1,
                iVertPos + ps.rcPaint.bottom / cyChar) ;

     for (i = iPaintBeg ; i <= iPaintEnd ; i++)
     {
          x = cxChar * (1 - iHorzPos) ;
          y = cyChar * (i - iVertPos) ;

          TextOut (hdc, x, y,
                sysmetrics[i].szLabel,
                lstrlen (sysmetrics[i].szLabel)) ;

          TextOut (hdc, x + 22 * cxCaps, y,
                sysmetrics[i].szDesc,
                lstrlen (sysmetrics[i].szDesc)) ;

          SetTextAlign (hdc, TA_RIGHT | TA_TOP) ;
          TextOut (hdc, x + 22 * cxCaps + 40 * cxChar, y, szBuffer,
                wsprintf (szBuffer, TEXT ("%5d"),
                GetSystemMetrics (sysmetrics[i].iIndex))) ;

          SetTextAlign (hdc, TA_LEFT | TA_TOP) ;
     }

          EndPaint (hwnd, &ps) ;
          return 0 ;

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

这个版本的程式仰赖Windows保存卷动列资讯并做边界检查。在WM_VSCROLL和 WM_HSCROLL 处理的开始，它取得所有的卷动列资讯，根据通知码调整位置，然後呼叫 SetScrollInfo 设置其位置。程式然後呼叫 GetScrollInfo。如果该位置超出了 SetScrollInfo 呼叫的范围，则由 Windows 来纠正该位置并且在GetScrollInfo 呼叫中传回正确的值。

SYSMETS3 使用 ScrollWindow 函式在视窗的显示区域中卷动资讯而不是重画它。虽然该函式很复杂（在新版本的 Windows 中已被更复杂的 ScrollWindowEx 所替代），SYSMETS3 仍以相当简单的方式使用它。函式的第二个参数给出了水

平卷动显示区域的数值，第三个参数是垂直卷动显示区域的数值，单位都是图素。

ScrollWindow 的最後两个参数设定为 NULL,这指出了要卷动整个显示区域。Windows 自动把显示区域中未被卷动操作覆盖的矩形设为无效。这会产生 WM_PAINT 讯息。再也不需要 InvalidateRect 了。注意 ScrollWindow 不是 GDI 函式，因为它不需装置内容代号。它是少数几个非 GDI 的 Windows 函式之一，它可以改变视窗的显示区域外观。很特殊但不方便，它是随卷动列函式一起记载在文件中。

WM_HSCROLL 处理拦截 SB_THUMBPOSITION 通知码并忽略 SB_THUMBTRACK。因而，如果使用者在水平卷动列上拖动卷动方块，在使用者释放滑鼠按钮之前，程式不会水平卷动视窗的内容。

WM_VSCROLL 的方法与之不同：程式拦截 SB_THUMBTRACK 讯息并忽略 SB_THUMBPOSITION。因而，程式随使用者在垂直卷动列上拖动卷动方块而垂直地滚动内容。这种想法很好，但应注意：一旦使用者发现程式会立即回应拖动的卷动方块，他们就会不断地来回拖动卷动方块。幸运的是现在的 PC 快得可以胜任这种严酷的测试。但是在较慢的机器上，可以考虑为 GetSystemMetrics 使用 SB_SLOWMACHINE 参数来替代这种处理。

加快 WM_PAINT 处理的一个方法由 SYSMETS3 展示：WM_PAINT 处理程式确定无效区域中的文字行并仅仅重画这些行。当然，程式码复杂一些，但速度很快。

不用滑鼠怎么办

在 Windows 的早期，有大量的使用者不喜欢使用滑鼠，而且，Windows 自身也不要求必须有滑鼠。虽然，没有滑鼠的 PC 现在走上了单色显示器和点阵印表机的没落之路，但我仍然建议您编写可以使用键盘来产生与滑鼠操作相同效果的程式，尤其对於像卷动列这样的基本操作物件更是如此。因为我们的键盘有一组游标移动键，所以应该实作同样的操作。

在下一章，您将学习使用键盘和在 SYSMETS3 中增加键盘介面的方法。您可能会注意到，SYSMETS3 似乎在通知码等於 SB_TOP 和 SB_BOTTOM 时处理了 WM_VSCROLL 讯息。前面已经提到过，视窗讯息处理程式不从卷动列接收这些讯息，所以，目前这是多余的程式码。当我们在下一章再次回到这个程式时，您将会明白这样做的原因。